MIXIN COMPOSITION SYNTHESIS BASED ON INTERSECTION TYPES

JAN BESSAI [a], TZU-CHUN CHEN [b], ANDREJ DUDENHEFNER [a], BORIS DÜDDER [a],
UGO DE'LIGUORO [c], AND JAKOB REHOF [a]

[a] Technical University of Dortmund, Dortmund, Germany
  e-mail address: jan.bessai@tu-dortmund.de
  e-mail address: andrej.dudenhefner@tu-dortmund.de
  e-mail address: boris.duedder@tu-dortmund.de
  e-mail address: jakob.rehof@tu-dortmund.de

[b] Technical University of Darmstadt, Darmstadt, Germany
  e-mail address: tcchen@rbg.informatik.tu-darmstadt.de

[c] University of Torino, Torino, Italy
  e-mail address: ugo.deliguoro@unito.it

ABSTRACT. We present a method for synthesizing compositions of mixins using type inhabitation in intersection types. First, recursively defined classes and mixins, which are functions over classes, are expressed as terms in a lambda calculus with records. Intersection types with records and record-merge are used to assign meaningful types to these terms without resorting to recursive types. Second, typed terms are translated to a repository of typed combinators. We show a relation between record types with record-merge and intersection types with constructors. This relation is used to prove soundness and partial completeness of the translation with respect to mixin composition synthesis. Furthermore, we demonstrate how a translated repository and goal type can be used as input to an existing framework for composition synthesis in bounded combinatory logic via type inhabitation. The computed result is a class typed by the goal type and generated by a mixin composition applied to an existing class.

1. Introduction

Starting with Cardelli's pioneering work [Car84], various typed λ-calculi extended with records have been thoroughly studied to model sophisticated features of object-oriented programming languages, like recursive objects and classes, object extension, method overriding and inheritance (see e.g. [AC96, Bru02, KL05]).

Here, we focus on the synthesis of mixin application chains. In the object-oriented paradigm, mixins [Moo86, Can79] have been introduced as an alternative construct for code reuse that improves over the limitations of multiple inheritance, e.g. connecting incompatible base classes and semantic ambiguities caused by the diamond problem [HTT87]. Together with abstract classes and traits, mixins (functions over classes) can be considered as an advanced construct to obtain flexible implementations of module libraries and to enhance code reusability; many popular programming languages miss native support for mixins, but they are an object of intensive study and research (e.g. [BPS99, OZ05]). In this setting we aim at synthesizing classes from a library of mixins. Our particular modeling approach is inspired by modern language features (e.g. ECMAScript "`bind`") to preserve contexts in order to prevent programming errors [Ecm11].

We formalize synthesis of classes from a library of mixins as an instance of the relativized type inhabitation problem in combinatory logic. Relativized type inhabitation is the following decision problem: given a combinatory type context Δ and a type τ does there exist an applicative term e such that e has type τ under the type assumptions in Δ? We implicitly include the problem of constructing such a term as the synthesized result.

Although relativized type inhabitation is undecidable even in simple types, it is decidable in k-bounded combinatory logic $\mathsf{BCL}_k(\to, \cap)$ [DMRU12]. $\mathsf{BCL}_k(\to, \cap)$ is the combinatory logic typed with arrow and intersection types, where k bounds the depth of types wrt. $\to$ used to instantiate schematic combinator types. Hence, an algorithm for semi-deciding type inhabitation for $\mathsf{BCL}(\to, \cap) = \bigcup_k \mathsf{BCL}_k(\to, \cap)$ can be obtained by iterative deepening over k and solving the corresponding decision problem in $\mathsf{BCL}_k(\to, \cap)$ [DMRU12]. In the present paper, we enable combinatory synthesis of classes via intersection typed mixin combinators. Intersection types [BCDC83] play an important rôle in combinatory synthesis, because they allow for semantic specification of components and synthesis goals [DMRU12, BDDM14]. They also allow a natural way to type records.

Now, looking at $\{C_1 : \sigma_1, \ldots, C_p : \sigma_p, M_1 : \tau_1, \ldots, M_q : \tau_q\} \subseteq \Delta$ as the abstract specification of a library including classes C_i and mixins M_j with interfaces σ_i and τ_j respectively, and given a type τ specifying a desired class, we may identify the class synthesis problem with the relativized type inhabitation problem of constructing a term e, i.e. an applicative composition of classes and mixins, typed by τ in Δ. To make this feasible, we have to bridge the gap between the expressivity of highly sophisticated type systems used for typing classes and mixins, for instance F-bounded polymorphism used in [CCH$^+$89, CHC90], and the system of intersection types from [BCDC83]. In doing so, we move from the system originally presented in [Lig01], consisting of a type assignment system of intersection and record types, to a λ-calculus which we enrich here with record merge operation (called "**with**" in [CHC90]), to allow for expressive mixin combinators. The type system is modified by reconstructing record types $\langle l_i : \sigma_i \mid i \in I \rangle$ as intersection of unary record types $\langle l_i : \sigma_i \rangle$, and considering a subtype relation extending the one in [BCDC83]. This is however not enough for typing record merge, for which we consider a type-merge operator $+$. The problem of typing extensible records and merge, faced for the first time in [Wan91, Rém92],

is notoriously hard; to circumvent difficulties the theory of record subtyping in [CHC90] (where a similar type-merge operator is considered) allows just for "exact" record typing, which involves subtyping in depth, but not in width. Such a restriction, that has limited effects wrt. a rich and expressive type system like F-bounded polymorphism, would be too severe in our setting. Therefore, we undertake a study of the type algebra of record types with intersection and type-merge, leading to a type assignment system where exact record typing is required only for the right-hand side operand of the term merge operator, which is enough to ensure soundness of typing.

The next challenge is to show that in our system we can type classes and mixins in a meaningful way. Classes are essentially recursive records. Mixins are made of a combination of fixed point combinators and record merge. Such combinators, which usually require recursive types, can be typed in our system by means of an iterative method exploiting the ability of intersection types to represent approximations of the potentially infinite unfolding of recursive definitions.

The final problem we face is the encoding of intersection types with record types and type-merge into the language of $\mathsf{BCL}_k(\to, \cap)$. For this purpose, we consider a conservative extension of bounded combinatory logic, called $\mathsf{BCL}_k(\mathbb{T}_\mathbb{C})$, where we allow unary type constructors that are monotonic and distribute over intersection. We show that the (semi) algorithm solving inhabitation for $\mathsf{BCL}_k(\to, \cap)$ can be adapted to $\mathsf{BCL}_k(\mathbb{T}_\mathbb{C})$, by proving that the key properties necessary to solve the inhabitation problem in $\mathsf{BCL}_k(\to, \cap)$ are preserved in $\mathsf{BCL}_k(\mathbb{T}_\mathbb{C})$ and showing how the type-merge operator can be simulated in $\mathsf{BCL}_k(\mathbb{T}_\mathbb{C})$. In fact, type-merge is not monotonic in its second argument, due to the lack of negative information caused by the combination of + and ∩. Our work culminates in two theorems that ensure soundness and completeness of the so obtained method wrt. synthesis of classes by mixin application.

1.1. **Contributions.** The contributions of this article can be summarized as follows:

- A type system with intersection types and records ($\mathbb{T}_{\langle\rangle}$) for a λ-calculus with records (Λ_R) is designed and its key properties are proven.
- $\mathbb{T}_{\langle\rangle}$ is used as a typed calculus for classes and mixins.
- Bounded combinatory logic and a decision algorithm for relativized type inhabitation are extended with constructors ($\mathsf{BCL}_k(\mathbb{T}_\mathbb{C})$) retaining complexity results.
- A sound and (partially) complete encoding of mixins and classes in $\mathbb{T}_{\langle\rangle}$ as combinators in $\mathsf{BCL}_k(\mathbb{T}_\mathbb{C})$ for synthesis is proven and exemplified.
- Negative information, i.e. the information on which labels are absent, is encoded with polynomial overhead.

1.2. **Organization.** The article is organized as follows: Section 2 discusses the development of traits and mixins as well as program synthesis. In Section 3, intersection and record types for a λ-calculus with records are introduced as a domain specific language for representing mixins and classes. Section 4 adapts bounded combinatory logic to include constructors ($\mathsf{BCL}_k(\mathbb{T}_\mathbb{C})$) as a foundation for mixin synthesis. Section 5 presents the encoding of record types in $\mathsf{BCL}_k(\mathbb{T}_\mathbb{C})$, mixin composition synthesis by type inhabitation and provides detailed examples including the use of semantic types. A conclusion in Section 6 hints at future work.

2. Related Work

This work evolves from the contributions [BDDM14, LC14, BDD⁺15a] to the workshop ITRS'14 and the original submission to TLCA15 [BDD⁺15b]. It combines two research areas, type systems for traits and mixins and program synthesis using type inhabitation.

2.1. **Type Systems for Traits and Mixins.** The concept of mixins goes back to research on Lisp dialects in the late 70's and early 80's [Can79, WM80, Moo86]. The motivation was to extend code organization in object-oriented languages, which was traditionally defined by hierarchical inheritance. To this end Cannon introduced base flavors serving as a starting point later to be extended by mixin flavors [Can79]. Mixin flavors implement particular features and can only be instantiated by application to existing flavors. They can define requirements to be met prior to application. If they use internal state, this state cannot be shared with other flavors.

Type theories for objects and classes were developed shortly after. Languages like Simula [DN66] and Smalltalk-80 [DS84] are object-oriented typed languages, but their type system is geared toward memory management rather than safe reuse. Various techniques have been deployed to enable type safe reuse of implementations. Cardelli's seminal paper "A Semantics of Multiple Inheritance" [Car84] filled this gap. Type inference for products with subtyping was shown to be decidable in [Mit84]. Cardelli improved upon the cumbersome encoding of entities using records with named projections instead of products. Subtyping addresses permutation of record entries as well as multiple inheritance. It ensures compatibility of horizontally extended and vertically specialized records.

A series of later developments dealt with issues introduced by recursion and obtained type inference. To this end Wand designed an ML-inspired system [Wan87] that later inspired the development of System $F_{\leq}$ [CCH⁺89]. A fine grained analysis of the connection between inheritance and subtyping was now possible [CHC90] and came to the surprising negative conclusion, that "inheritance is not subtyping". This analysis by Cook, Hill and Canning is also the first to discuss the difference between early and late binding of the recursion inherently present in objects: delegating recursion to methods using a self parameter (late binding) as done previously by Mitchell [Mit90] unifies classes and objects, whereas early fixpoint computation at the class level is open for later modifications by inheritance and mixins. Subsequently, Bracha and Cook reinterpreted mixins in the light of classes and inheritance, where they can act as abstract subclasses [BC90]. Their main contribution was to shed light on the abstract operation of mixin application in contrast to prior work, which focused on implementation details like class hierarchy linearization. This operation turns out to be powerful enough to encode (multiple) inheritance. From there on, development took two main routes. One direction was the development of advanced object calculi, which is covered in great detail in [AC96]. The other direction was to model the type systems of mainstream programming languages. Featherweight Java by Igarashi, Pierce and Wadler is a prominent example [IPW01] for the latter direction. It inspired a mixin based treatment of virtual classes in [EOC06]. A parallel development focused on traits, which are a restriction of mixins to subtype compatible overwriting [LS08, LSZ12]. Bono et al. suggested a formal type system for a calculus with traits [BDG07, BDG08]. The most recent studies follow this trend to bring together the two directions. Java has been partially formalized by Rowe and van Bakel [BR13, RB14], and completely in K-Java [BR15]. Recently, the Scala compiler

is being reworked to match DOT [AMO12, ARO14, AGO$^+$16], a formal dependently typed core specification.

2.2. **Synthesis by Type Inhabitation.** Synthesis is by now a vast area of computer science, and we can only attempt here to place our approach broadly in relation to major points of comparison.

The synthesis problem can be traced back to Alonzo Church [Chu57, Tho09]. It was first considered for the problem of automatically constructing a finite-state procedure implementing a given input/output relation over infinite bitstreams specified as a logical formula. The automata-theoretic approach was notably advanced by Pnueli and Rosner in the context of linear temporal logic (LTL) [PR89] and by many others since then. Another major branch of work in synthesis is deductive synthesis, as studied by Manna and Waldinger, who rely on proof systems for program properties [MW80], with many more works following in this tradition.

The presented approach to the synthesis problem is proof-theoretic and can best be characterized more specifically as *type-based* and *component-oriented*. This approach was initiated by Rehof et al. [RU11, DMRU12, Reh13] around the idea of using the inhabitation problem in bounded combinatory logics with intersection types [BCDC83] as a foundation for synthesis. In contrast to standard combinatory logic [HS08, DCH92] invented by Schönfinkel in 1924 [Sch24], k-bounded combinatory logic $\mathsf{BCL}_k(\to, \cap)$ [DMRU12] introduces a bound k on the level of types, i.e. depth wrt. $\to$, used to instantiate schematic combinator types. Additionally, rather than considering a *fixed* base of combinators (for example, the base $\mathbf{S}, \mathbf{K}$), the inhabitation problem is *relativized* to an arbitrary set Δ of typed combinators, given as part of the input to the relativized inhabitation problem:

Given Δ and τ, is there an applicative term e such that $\Delta \vdash_{\mathsf{BCL}_k(\to, \cap)} e : \tau$?

In many instances the inhabitation problem with a fixed-base is much easier than in the relativized case, where the base is part of the input. For example, PSPACE-completeness of inhabitation in the simple-typed λ-calculus [Sta79] implies PSPACE-completeness of the equivalent simple-typed $\mathbf{SK}$-calculus. The related problem of term enumeration for simple-typed λ-calculus has been studied by Ben-Yelles [BY79] and, more recently, by Hindley [Hin08].

Linial and Post [LP49] initiated the study of decision problems for arbitrary propositional axiom systems (partial propositional calculi, abbreviated PPC), in reaction to a question posed by Tarski in 1946. In the Linial-Post theorem they show existence of a PPC with an unsolvable decision problem. Later, Singletary showed that every recursively enumerable many-one degree can be represented by the implicational fragment of PPC [Sin74]. This implies that the relativized inhabitation problem is undecidable for combinatory logic with schematism even in simple types. Recent developments [Bok15] show that it is undecidable, whether a given finite set of propositional formulas constitutes an adequate axiom system for a fixed intuitionistic implicational propositional calculus.

In contrast, the main result of [DMRU12] is that the relativized inhabitation problems for $\mathsf{BCL}_k(\to, \cap)$ with intersection types form an infinite hierarchy, being $(k+2)$-EXPTIME-complete for each fixed bound k.

In [Reh13], unbounded relativized inhabitation is seen, already at the level of simple types, to constitute a Turing-complete logic programming language for program composition,

in a way that is related to work on proof-theoretic generalizations of logic programming languages [MNPS91].

Perhaps surprisingly, component-oriented synthesis is a relatively recent development. The approach of combinatory logic synthesis is basically motivated by the idea that type structure provides a natural and code-oriented vehicle for synthesis specifications together with the idea that combinatory logic provides a natural type-theoretic model of components, component interfaces, and component composition. Vardi and Lustig initiated the component-oriented approach within the automata-theoretic tradition, explicitly developing the idea of synthesizing systems from preexisting components [LV09] with the idea of leveraging design intelligence and efficiency from components as building blocks for synthesis. A recent Dagstuhl meeting explored these ideas of design and synthesis from components across different communities [RV14].

Realizing the component-based idea within type-based synthesis requires ways to deal with semantic specification as well as the combinatorial explosion of search spaces inherent in synthesis problems. Rehof et al. [RU12] introduced the idea of using intersection types [BCDC83] to type-based synthesis as a means of addressing the problem of search control and semantic specification at the type level. We refer to types enriched with semantic specifications through intersection types as *semantic types.* In addition to semantic types, Rehof et al. have introduced the idea of *staging* into synthesis via modal types [DMR14]. Simple types were used by Steffen et al. [SMvdB97] in the context of temporal logic synthesis to semantically enrich temporal specifications with taxonomic information.

The notion of composition synthesis using semantic types is related to adaptation synthesis via proof counting [HHSW02, WY05]. In particular, in [HHSW02] typed predicate logic is used for semantic specification at the interface level. We follow this idea, however the type system, underlying logic and algorithmic methods are different.

Refinement types [FP91] externally relate specifications to implementation types. The refinement type scheme structurally constraints how refinement types are formed: this prevents semantic specifications like (Int → Int) ∩ Injective, which are possible in our system. Recently, refinement types have been used for example guided synthesis in [FOWZ16]. Types have also been used in order to synthesize code completions [GKKP13]. Intersection types are not only useful for semantic specification, but also to encode objects. To this end they can be combined with records as proposed in [Pie91]. Their relation to object-oriented inheritance has been studied in [CP96] and serves as an inspiration for our work.

Combinatory logic synthesis has been implemented in a framework, Combinatory Logic Synthesizer **(CL)S**, which is still being further developed [BDD⁺14].

3. Intersection Types for Mixins and Classes

3.1. **Intersection and record types.** We consider a type-free λ-calculus of extensible records, equipped with a merge operator. The term syntax is defined by the following grammar:

$$\Lambda_R \ni M, N, M_i \quad ::= \quad x \mid (\lambda x.M) \mid (MN) \mid (M.l) \mid R \mid (M \oplus R) \quad \text{terms}$$
$$R \quad ::= \quad \langle l_i = M_i \mid i \in I \rangle \qquad\qquad\qquad \text{records}$$

where $x \in \textbf{Var}$ and $l \in \textbf{Label}$ range over denumerably many *term variables* and *labels* respectively, and the sets of indexes I are finite. Free and bound variables are defined

as for the ordinary λ-calculus, and we name Λ_R^0 the set of all closed terms in Λ_R; terms are identified up to renaming of bound variables and $M\{N/x\}$ denotes capture avoiding substitution of N for x in M. We adopt notational conventions from [Bar84]; in particular, application associates to the left and external parentheses are omitted when unnecessary; also the dot notation for record selection takes precedence over λ, so that $\lambda x.\ M.l$ reads as $\lambda x.(M.l)$. If not stated otherwise $\oplus$ also associates to the left, and we avoid external parentheses when unnecessary.

Terms $R \equiv \langle l_i = M_i \mid i \in I \rangle$ (writing $\equiv$ for syntactic identity) represent *records*, with fields l_i and M_i as the respective values; we set $lbl(\langle l_i = M_i \mid i \in I \rangle) = \{l_i \mid i \in I\}$. For records we adopt the usual notation to combine sets, i.e. $\{l_i, l_j \mid i \in I, j \in J\} = \{l_i \mid i \in I \cup J\}$. The term $M.l$ is *field selection* and $M \oplus R$ is record *merge*. In particular, if R_1 and R_2 are records then $R_1 \oplus R_2$ is the record having as fields the union of the fields of R_1 and R_2 and as values those of the original records but in case of ambiguity, where the values in R_2 prevail. The syntactic constraint that R is a record in $M \oplus R$ is justified after Definition 3.11. Note that a variable x is not a record, hence $\lambda x.x \oplus R$ is well-formed for any record R while $\lambda x.R \oplus x$ is not.

The actual meaning of these operations is formalized by the following reduction relation:

Definition 3.1 (Λ_R reduction)**.** Reduction $\longrightarrow\ \subseteq \Lambda_R^2$ is the least compatible relation such that:

$$(\beta) \qquad (\lambda x.M)N \quad\longrightarrow\quad M\{N/x\}$$

$$(sel) \qquad \langle l_i = M_i \mid i \in I \rangle.l_j \quad\longrightarrow\quad M_j \qquad\qquad \text{if } j \in I$$

$$(\oplus) \quad \langle l_i = M_i \mid i \in I \rangle \oplus \langle l_j = N_j \mid j \in J \rangle \quad\longrightarrow\quad \langle l_i = M_i,\ l_j = N_j \mid i \in I \smallsetminus J,\ j \in J \rangle$$

Record merge subsumes field update as considered in [Lig01]: $(M.l := N)$ is exactly $(M \oplus \langle l = N \rangle)$, but merge is not uniformly definable in terms of update as long as labels are not expressions in the calculus, therefore we take merge as primitive operator. The reduction relation $\longrightarrow$ is Church-Rosser namely its transitive closure $\longrightarrow^*$ is confluent.

Theorem 3.2 (Church-Rosser property)**.** *For all M if $M \longrightarrow^* M_1$ and $M \longrightarrow^* M_2$ then there is M_3 such that $M_1 \longrightarrow^* M_3$ and $M_2 \longrightarrow^* M_3$.*

Proof. By adapting Tait and Martin-Löf's proof of Church-Rosser property of β-reduction of λ-calculus, which is based on the so called 1-reduction $\twoheadrightarrow_1$. To the clauses for ordinary λ-calculus (see e.g. [Bar84] Definition 3.2.3) we add:

i) $\forall i \in I.\ M_i \twoheadrightarrow_1 M_i' \Rightarrow \langle l_i = M_i \mid i \in I \rangle \twoheadrightarrow_1 \langle l_i = M_i' \mid i \in I \rangle$

ii) $M \twoheadrightarrow_1 M' \Rightarrow M.l \twoheadrightarrow_1 M'.l$

iii) $M \twoheadrightarrow_1 M'\ \&\ R \twoheadrightarrow_1 R' \Rightarrow M \oplus R \twoheadrightarrow_1 M' \oplus R'$

iv) $\forall i \in I.\ M_i \twoheadrightarrow_1 M_i'\ \&\ j \in I \Rightarrow \langle l_i = M_i \mid i \in I \rangle.l_j \twoheadrightarrow_1 M_j'$

v) $\forall i \in I.\ M_i \twoheadrightarrow_1 M_i'\ \&\ \forall j \in J.\ N_j \twoheadrightarrow_1 N_j' \Rightarrow$
$\langle l_i = M_i \mid i \in I \rangle \oplus \langle l_j = N_j \mid j \in J \rangle \twoheadrightarrow_1 \langle l_i = M_i',\ l_j = N_j' \mid i \in I \smallsetminus J,\ j \in J \rangle$

Now it is easy to check that $\twoheadrightarrow_1^* = \longrightarrow^*$ so that by Lemma 3.2.2 of [Bar84] it suffices to show that $\twoheadrightarrow_1$ satisfies the diamond property:

$$M \twoheadrightarrow_1 M_1\ \&\ M \twoheadrightarrow_1 M_2 \Rightarrow \exists M_3.\ M_1 \twoheadrightarrow_1 M_3\ \&\ M_2 \twoheadrightarrow_1 M_3.$$

The latter is easily established by induction over the definition of $M \twoheadrightarrow_1 M_1$ and by cases of $M \twoheadrightarrow_1 M_2$. $\qquad\qquad\square$

In the spirit of Curry's assignment of polymorphic types and of intersection types in particular, types are introduced as a syntactical tool to capture semantic properties of terms, rather than as constraints to term formation.

Definition 3.3 (Intersection types for Λ_R)**.**

$$\mathbb{T} \ni \sigma, \sigma_i \quad ::= \quad a \mid \omega \mid \sigma_1 \to \sigma_2 \mid \sigma_1 \cap \sigma_2 \mid \rho \quad \text{types}$$
$$\mathbb{T}_{\langle\rangle} \ni \rho, \rho_i \quad ::= \quad \langle\rangle \mid \langle l : \sigma \rangle \mid \rho_1 + \rho_2 \mid \rho_1 \cap \rho_2 \quad \text{record types}$$

where a ranges over *type constants* and l ranges over the denumerable set **Label**.

We use σ, τ, possibly with sub and superscripts, for types in $\mathbb{T}$ and ρ, ρ_i, possibly with superscripts, for record types in $\mathbb{T}_{\langle\rangle}$ only. Note that $\to$ associates to the right, and $\cap$ binds stronger than $\to$. As with intersection type systems for the λ-calculus, the intended meaning of types are sets, provided a set theoretic interpretation of type constants a.

Following [BCDC83], type semantics is given axiomatically by means of the subtyping relation $\leq$, that can be interpreted as subset inclusion: see e.g. [Mit96] §10.4 for extending such interpretation to record types. It is the least pre-order over $\mathbb{T}$ satisfying Definition 3.4 and Definition 3.5.

Definition 3.4 (Type inclusion: arrow and intersection types)**.**

(1) $\sigma \leq \omega$ and $\omega \leq \omega \to \omega$
(2) $\sigma \cap \tau \leq \sigma$ and $\sigma \cap \tau \leq \tau$
(3) $\sigma \leq \tau_1 \ \& \ \sigma \leq \tau_2 \Rightarrow \sigma \leq \tau_1 \cap \tau_2$
(4) $(\sigma \to \tau_1) \cap (\sigma \to \tau_2) \leq \sigma \to \tau_1 \cap \tau_2$
(5) $\sigma_2 \leq \sigma_1 \ \& \ \tau_1 \leq \tau_2 \Rightarrow \sigma_1 \to \tau_1 \leq \sigma_2 \to \tau_2$

By 3.4.2 type ω is the top w.r.t. $\leq$; by 3.4.2 and 3.4.3 $\sigma \cap \tau$ is the meet. Writing $\sigma = \tau$ if $\sigma \leq \tau$ and $\tau \leq \sigma$, from 3.4.1 and 3.4.5 we have $\omega \leq \omega \to \omega \leq \sigma \to \omega \leq \omega$, which are all equalities. Finally from 3.4.4 and 3.4.5 we also have $(\sigma \to \tau_1) \cap (\sigma \to \tau_2) = \sigma \to \tau_1 \cap \tau_2$.

If $\sigma_1 \leq \tau_1$ and $\sigma_2 \leq \tau_2$ then by transitivity we have $\sigma_1 \cap \sigma_2 \leq \sigma_1 \leq \tau_1$ and $\sigma_1 \cap \sigma_2 \leq \sigma_2 \leq \tau_2$ from which $\sigma_1 \cap \sigma_2 \leq \tau_1 \cap \tau_2$ follows by 3.4.3. Hence $\leq$ is a precongruence w.r.t. $\cap$, and hence $=$ is a congruence.

Definition 3.5 (Type inclusion: record types)**.**

(1) $\langle l : \sigma \rangle \leq \langle\rangle$
(2) $\langle l : \sigma \rangle \cap \langle l : \tau \rangle \leq \langle l : \sigma \cap \tau \rangle$
(3) $\sigma \leq \tau \Rightarrow \langle l : \sigma \rangle \leq \langle l : \tau \rangle$
(4) $\rho + \langle\rangle = \langle\rangle + \rho = \rho$
(5) $(\rho_1 + \rho_2) + \rho_3 = \rho_1 + (\rho_2 + \rho_3)$
(6) $(\rho_1 \cap \rho_2) + \rho_3 = (\rho_1 + \rho_3) \cap (\rho_2 + \rho_3)$
(7) $\langle l : \sigma \rangle + (\langle l : \tau \rangle \cap \rho) = \langle l : \tau \rangle \cap \rho$
(8) $\langle l : \sigma \rangle + (\langle l' : \tau \rangle \cap \rho) = \langle l' : \tau \rangle \cap (\langle l : \sigma \rangle + \rho)$ if $l \neq l'$
(9) $\rho_1 \leq \rho_2 \Rightarrow \rho_1 + \rho \leq \rho_2 + \rho$
(10) $\rho_1 = \rho_2 \Rightarrow \rho + \rho_1 = \rho + \rho_2$

While Definition 3.4 is standard after [BCDC83], comments on Definition 3.5 are in order. First observe that from $\langle l : \sigma \rangle \leq \langle\rangle$ we obtain that $\langle l : \sigma \rangle \cap \langle\rangle = \langle l : \sigma \rangle$, so that by putting $\rho = \langle\rangle$ in 3.5.7 and in 3.5.8 we obtain by 3.5.10:

$$\langle l : \sigma \rangle + \langle l : \tau \rangle = \langle l : \tau \rangle, \qquad \langle l : \sigma \rangle + \langle l' : \tau \rangle = \langle l : \sigma \rangle \cap \langle l' : \tau \rangle \qquad (l \neq l'). \tag{3.1}$$

Type $\langle\rangle$ is the type of all records. Type $\langle l : \sigma \rangle$ is a unary record type, whose meaning is the set of records having at least a field labeled by l, with value of type σ; therefore $\langle l : \sigma \rangle \cap \langle l : \tau \rangle$ is the type of records having label l with values both of type σ and τ, that is of type $\sigma \cap \tau$. In fact the following equation is derivable:

$$\langle l : \sigma \rangle \cap \langle l : \tau \rangle = \langle l : \sigma \cap \tau \rangle. \tag{3.2}$$

On the other hand $\langle l : \sigma \rangle \cap \langle l' : \tau \rangle$, with $l \neq l'$, is the type of records having fields labeled by l and l', with values of type σ and τ respectively. It follows that intersection of record types can be used to express properties of records with arbitrary (though finitely) many fields, which justifies the abbreviations $\langle l_i : \sigma_i \mid i \in I \neq \varnothing \rangle = \bigcap_{i \in I} \langle l_i : \sigma_i \rangle$ and $\langle l_i : \sigma_i \mid i \in \varnothing \rangle = \langle \rangle$, where we assume that the l_i are pairwise distinct. Finally, as it will be apparent from Definition 3.11 below, $\rho_1 + \rho_2$ is the type of all records obtained by merging a record of type ρ_1 with a record of type ρ_2, which is intended to type $\oplus$ that is at the same time a record extension and field updating operation. Since this is the distinctive feature of the system introduced here, we comment on this by means of a few lemmas, illustrating its properties.

Lemma 3.6.
(1) $\langle l_{j_0} : \sigma \rangle + \langle l_j : \tau_j \mid j \in J \rangle = \langle l_j : \tau_j \mid j \in J \rangle$ *if* $j_0 \in J$,
(2) $\langle l_{j_0} : \sigma \rangle + \langle l_j : \tau_j \mid j \in J \rangle = \langle l_{j_0} : \sigma \rangle \cap \langle l_j : \tau_j \mid j \in J \rangle$ *if* $j_0 \notin J$
Proof.
(1) By commutativity and associativity of the meet operator $\cap$ and by 3.5.10, when $j_0 \in J$ we may freely assume that

$$\langle l_{j_0} : \sigma \rangle + \langle l_j : \tau_j \mid j \in J \rangle = \langle l_{j_0} : \sigma \rangle + (\langle l_{j_0} : \tau_{j_0} \rangle \cap \langle l_j : \tau_j \mid j \in J \smallsetminus \{j_0\} \rangle)$$

which is equal to $\langle l_{j_0} : \tau_{j_0} \rangle \cap \langle l_j : \tau_j \mid j \in J \smallsetminus \{j_0\} \rangle$ by 3.5.7, namely to $\langle l_j : \tau_j \mid j \in J \rangle$.
(2) By induction over the cardinality of J. If it is 0 then $\langle l_j : \tau_j \mid j \in \varnothing \rangle = \langle \rangle$ and:

$$\langle l_{j_0} : \sigma \rangle + \langle \rangle = \langle l_{j_0} : \sigma \rangle = \langle l_{j_0} : \sigma \rangle \cap \langle \rangle$$

using 3.5.4, 3.5.1 and the fact that $\cap$ is the meet. If $|J| > 0$ then, by reasoning as before we have:

$$\langle l_{j_0} : \sigma \rangle + \langle l_j : \tau_j \mid j \in J \rangle = \langle l_{j_0} : \sigma \rangle + (\langle l_{j_1} : \tau_{j_1} \rangle \cap \langle l_j : \tau_j \mid j \in J \smallsetminus \{j_1\} \rangle).$$

for some $j_1 \in J$. Now $j_0 \notin J$ implies $l_{j_0} \neq l_{j_1}$, and the thesis follows by 3.5.8 and the induction hypothesis. $\qquad\square$

Lemma 3.7.
(1) $\langle l_i : \sigma_i \mid i \in I \rangle \leq \langle l_j : \tau_j \mid j \in J \rangle \Leftrightarrow J \subseteq I \ \& \ \forall j \in J. \ \sigma_j \leq \tau_j$,
(2) $\langle l_i : \sigma_i \mid i \in I \rangle \cap \langle l_j : \tau_j \mid j \in J \rangle = \langle l_i : \sigma_i, l_j : \tau_j, l_k : \sigma_k \cap \tau_k \mid i \in I \smallsetminus J, j \in J \smallsetminus I, k \in I \cap J \rangle$,
(3) $\langle l_i : \sigma_i \mid i \in I \rangle + \langle l_j : \tau_j \mid j \in J \rangle = \langle l_i : \sigma_i, l_j : \tau_j \mid i \in I \smallsetminus J, j \in J \rangle$,
(4) $\forall \rho \in \mathbb{T}_{\langle \rangle}. \ \exists \langle l_i : \sigma_i \mid i \in I \rangle. \ \rho = \langle l_i : \sigma_i \mid i \in I \rangle$.

Proof. Through this proof let $\rho_1 \equiv \langle l_i : \sigma_i \mid i \in I \rangle$ and $\rho_2 \equiv \langle l_j : \tau_j \mid j \in J \rangle$, where $\equiv$ denotes syntactic identity up to commutativity and associativity of $\cap$.
(1) The only if part is proved by induction over the derivation of $\rho_1 \leq \rho_2$. If this equation is an instance of axiom 3.5.1 then $J = \varnothing \subseteq I$ and $\forall j \in J. \ \sigma_j \leq \tau_j$ is vacuously true.
   Axiom 3.5.2 doesn't apply being the labels l_i pairwise distinct.
   In case the derivation ends by rule 3.5.3 we have $I = J = \{l\}$ and $\sigma \leq \tau$ by the premise of the rule.

In case of the axiom instance $\langle l_1 : \sigma \rangle \cap \langle l_2 : \tau \rangle \leq \langle l_1 : \sigma \rangle$ of 3.4.2 we have $J = \{l_1\} \subseteq \{l_1, l_2\} = I$ and obviously $\sigma \leq \sigma$.

Finally in case of deriving $\rho_1 \leq (\rho_3 \cap \rho_4)$ from $\rho_1 \leq \rho_3$ and $\rho_1 \leq \rho_4$, where $\rho_2 \equiv \rho_3 \cap \rho_4$, we have that for some J', J'' such that $J = J' \cup J''$ it is $\rho_3 \equiv \langle l_h : \tau_h \mid h \in J' \rangle$ and $\rho_4 \equiv \langle l_k : \tau_k \mid k \in J'' \rangle$. By induction we know that $J' \subseteq I$ and $\sigma_h \leq \tau_h$ for all $h \in J'$ and $J'' \subseteq I$ and $\sigma_k \leq \tau_k$ for all $k \in J''$. Therefore $J = J' \cup J'' \subseteq I$ and $\sigma_j \leq \tau_j$ for all $j \in J$.

For the if part we reason by induction over the cardinality of J. If $|J| = 0$ then $\rho_2 \equiv \langle \rangle$, hence either $I = \varnothing$, but then $\rho_1 \equiv \langle \rangle$; or $I \neq \varnothing$ so that $\rho_1 \leq \langle l_i : \sigma_i \rangle \leq \langle \rangle$ for any $i \in I$, using axiom 3.5.1. If $|J| > 0$ then let $j_0 \in J$ be any index: then $\rho_2 \equiv \langle l_j : \tau_j \mid j \in J \smallsetminus \{j_0\} \rangle \cap \langle l_{j_0} : \tau_{j_0} \rangle$. Now by induction $\rho_1 \leq \langle l_j : \tau_j \mid j \in J \smallsetminus \{j_0\} \rangle$; by hypothesis $\sigma_{j_0} \leq \tau_{j_0}$ so that $\rho_1 \leq \langle l_{j_0} : \sigma_{j_0} \rangle \leq \langle l_{j_0} : \tau_{j_0} \rangle$ and we conclude being $\cap$ the meet w.r.t. $\leq$.

(2) This is an immediate consequence of commutativity, associativity, idempotency of $\cap$ and equation (3.2).

(3) If $I = \varnothing$ then $\rho_1 = \langle \rangle$ and the thesis follows by 3.5.4. Otherwise we have:

$$\begin{aligned}
\rho_1 + \rho_2 &= \bigcap_{i \in I} (\langle l_i : \sigma_i \rangle + \rho_2) && \text{by repeated applications of 3.5.6} \\
&= \bigcap_{i \in I} \langle l_i : \sigma_i, l_j : \tau_j \mid i \notin J, j \in J \rangle && \text{by Lemma 3.6.1 and 3.6.2} \\
&\equiv \langle l_i : \sigma_i, l_j : \tau_j \mid i \in I \smallsetminus J, j \in J \rangle.
\end{aligned}$$

(4) By induction over ρ. If $\rho = \langle \rangle$ there is nothing to prove.

If $\rho \equiv \rho_1 \cap \rho_2$ then by induction $\rho_1 = \langle l_i^1 : \sigma_i^1 \mid i \in I_1 \rangle$ and $\rho_2 = \langle l_i^2 : \sigma_i^2 \mid i \in I_2 \rangle$ so that the thesis follows by (2) above and the fact that $=$ is a congruence w.r.t. $\cap$.

Similarly if $\rho \equiv \rho_1 + \rho_2$ the thesis follows by induction using (3) and the fact that $=$ is a congruence w.r.t. $+$ by 3.5.9 and 3.5.10. □

Remark 3.8. By 3.5.6 the $+$ distributes to the left over $\cap$; however this doesn't hold to the right, namely $\rho_1 + (\rho_2 \cap \rho_3) \neq (\rho_1 + \rho_2) \cap (\rho_1 + \rho_3)$: take $\rho_1 \equiv \langle l_1 : \sigma_1, l_2 : \sigma_2 \rangle$, $\rho_2 \equiv \langle l_1 : \sigma_1' \rangle$ and $\rho_3 \equiv \langle l_2 : \sigma_2' \rangle$, with $\sigma_1 \neq \sigma_1'$ and $\sigma_2 \neq \sigma_2'$. Then by Lemma 3.7.3 we have: $\rho_1 + (\rho_2 \cap \rho_3) = \rho_1 + \langle l_1 : \sigma_1', l_2 : \sigma_2' \rangle = \langle l_1 : \sigma_1', l_2 : \sigma_2' \rangle$, while $(\rho_1 + \rho_3) \cap (\rho_2 + \rho_3) = \langle l_1 : \sigma_1, l_2 : \sigma_2' \rangle \cap \langle l_1 : \sigma_1', l_2 : \sigma_2 \rangle = \langle l_1 : \sigma_1 \cap \sigma_1', l_2 : \sigma_2 \cap \sigma_2' \rangle$. The last example suggests that $(\rho_1 + \rho_2) \cap (\rho_1 + \rho_3) \leq \rho_1 + (\rho_2 \cap \rho_3)$.

On the other hand by 3.5.9 $+$ is monotonic in its first argument. However we have $\rho_2 \leq \rho_3 \not\Rightarrow \rho_1 + \rho_2 \leq \rho_1 + \rho_3$. Indeed:

$$\begin{aligned}
\langle l_0 : \sigma_1, l_1 : \sigma_2 \rangle + \langle l_1 : \sigma_3, l_2 : \sigma_4 \rangle &= \langle l_0 : \sigma_1, l_1 : \sigma_3, l_2 : \sigma_4 \rangle \\
&\not\leq \langle l_0 : \sigma_0, l_1 : \sigma_1, l_2 : \sigma_4 \rangle && \text{if } \sigma_3 \not\leq \sigma_1 \\
&= \langle l_0 : \sigma_1, l_1 : \sigma_2 \rangle + \langle l_2 : \sigma_4 \rangle
\end{aligned}$$

even if $\langle l_1 : \sigma_1, l_2 : \sigma_2 \rangle \leq \langle l_2 : \sigma_2 \rangle$. From this we conclude that $+$ is not monotonic in its second argument. Comparing this with 3.5.10 we see that $\leq$ is not a precongruence w.r.t. $+$, while $=$, namely the symmetric closure of $\leq$, is a congruence.

Finally if one assumes (1)-(3) of Lemma 3.7 as axioms then (pre)-congruence axioms in Definition 3.5 become derivable. The opposite is hardly provable and possibly false[1].

Lemma 3.7.1 states that subtyping among intersection of unary record types subsumes subtyping in width and depth of ordinary record types from the literature. Lemma 3.7.3 shows that the $+$ type constructor reflects at the level of types the operational behavior of the merge operator $\oplus$. Lemma 3.7.4 says that any record type is equivalent to an intersection of unary record types; this implies that types of the form $\rho_1 + \rho_2$ are eliminable in principle.

---

[1] Remark of an anonymous referee about our former attempt to derive 3.5.9 and 3.5.10 from the other axioms.

However, they play a key role in typing mixins, motivating the issue of control of negative information in the synthesis process: see Section 5. More properties of subtyping record types w.r.t. $+$ and $\cap$ are listed in the next two lemmas.

Lemma 3.9. $\rho_1 + \rho_2 = \rho_1 \cap \rho_2 \Leftrightarrow \rho_1 + \rho_2 \leq \rho_1$.

Proof. W.l.o.g. by 3.7.4 let us assume that $\rho_1 = \langle l_i : \sigma_{1,i} \mid i \in I \rangle$, $\rho_2 = \langle l_j : \sigma_{2,j} \mid j \in J \rangle$ and $\rho_3 = \langle l_k : \sigma_{3,k} \mid k \in K \rangle$. If $\rho_1 + \rho_2 = \rho_1 \cap \rho_2$ then by 3.7.2 and 3.7.3:

$$\langle l_i : \sigma_{1,i}, l_j : \sigma_{2,j} \mid i \in I \smallsetminus J, j \in J \rangle$$
$$= \langle l_i : \sigma_{1,i}, l_j : \sigma_{2,j}, l_k : \sigma_{1,k} \cap \sigma_{2,k} \mid i \in I \smallsetminus J, j \in J \smallsetminus I, k \in I \cap I \rangle$$

Obviously $J = (J \smallsetminus I) \cup (I \cap J)$ and from 3.7.1 we deduce that $\sigma_{2,j} = \sigma_{1,j} \cap \sigma_{2,j}$ for all $j \in J$, hence $\sigma_{2,j} \leq \sigma_{1,j}$ so we conclude that $\rho_1 + \rho_2 \leq \rho_1$ again by 3.7.1. Since all these implications can be reverted, we conclude. $\square$

Let us define the map $lbl : \mathbb{T}_{\langle \rangle} \to \wp(\mathbf{Label})$ (where $\wp(\mathbf{Label})$ is the powerset of $\mathbf{Label}$) by:

$$lbl(\langle l : \sigma \rangle) = \{l\}, \quad lbl(\rho_1 \cap \rho_2) = lbl(\rho_1 + \rho_2) = lbl(\rho_1) \cup lbl(\rho_2).$$

Lemma 3.10.
(1) $\rho_1 = \rho_2 \Rightarrow lbl(\rho_1) = lbl(\rho_2)$,
(2) $lbl(\rho_1) \cap lbl(\rho_2) = \varnothing \Rightarrow \rho_1 + \rho_2 = \rho_1 \cap \rho_2$,
(3) $lbl(\rho_1) \subseteq lbl(\rho_2) \Rightarrow \rho_1 + \rho_2 = \rho_2$,
(4) $lbl(\rho_2) = lbl(\rho_3) \Rightarrow (\rho_1 + \rho_2) \cap (\rho_1 + \rho_3) = \rho_1 + (\rho_2 \cap \rho_3)$.

Proof. Part (1) follows by 3.7.1, and (2) is a consequence of the fact that if $lbl(\rho_1) \cap lbl(\rho_2) = \varnothing$ then equation (3.1) $\langle l : \sigma \rangle + \langle l' : \tau \rangle = \langle l : \sigma \rangle \cap \langle l' : \tau \rangle$ repeatedly applies. Part (3) follows by 3.7.3.

To see part (4) we can assume by 3.7.4 that $\rho_1 = \langle l_i : \sigma_{1,i} \mid i \in I \rangle$, $\rho_2 = \langle l_j : \sigma_{2,j} \mid j \in J \rangle$ and $\rho_3 = \langle l_k : \sigma_{3,k} \mid k \in K \rangle$. Since $lbl(\rho_2) = lbl(\rho_3)$ we have that $J = K$, so that:

$$
\begin{array}{rcll}
\rho_1 + \rho_2 & = & \langle l_i : \sigma_{1,i}, l_j : \sigma_{2,j} \mid i \in I \smallsetminus J, j \in J \rangle & \text{by 3.7.3} \\
\rho_1 + \rho_3 & = & \langle l_i : \sigma_{1,i}, l_j : \sigma_{3,j} \mid i \in I \smallsetminus J, j \in J \rangle & \text{by 3.7.3} \\
\rho_2 \cap \rho_3 & = & \langle l_j : \sigma_{2,j} \cap \sigma_{3,j} \mid j \in J \rangle & \text{by 3.7.2}
\end{array}
$$

Again by 3.7.2, 3.7.3 and the fact that $\sigma_{1,i} \cap \sigma_{1,i} = \sigma_{1,i}$ we conclude that $(\rho_1 + \rho_2) \cap (\rho_1 + \rho_3)$ and $\rho_1 + (\rho_2 \cap \rho_3)$ are both equal to

$$\langle l_i : \sigma_{1,i}, l_j : \sigma_{2,j} \cap \sigma_{3,j} \mid i \in I \smallsetminus J, j \in J \rangle. \qquad \square$$

About Lemma 3.10.2 above note that condition $lbl(\rho_1) \cap lbl(\rho_2) = \varnothing$ is essential, since $\cap$ is commutative while $\rho_1 + \rho_2 \neq \rho_2 + \rho_1$ in general, as it immediately follows by Lemma 3.7.3.

We come now to the type assignment system. A *basis* (also called a context in the literature) is a finite set $\Gamma = \{x_1 : \sigma_n, \dots, x_n : \sigma_n\}$, where the variables x_i are pairwise distinct; we set $dom(\Gamma) = \{x \mid \exists \sigma. \; x : \sigma \in \Gamma\}$ and we write $\Gamma, x : \sigma$ for $\Gamma \cup \{x : \sigma\}$ where $x \notin dom(\Gamma)$. Then we consider the following extension of the system in [BCDC83], also called **BCD** in the literature.

Definition 3.11 (Type Assignment)**.** The rules of the assignment system are:

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \, (Ax) \qquad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau} \, (\to I) \qquad \frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau} \, (\to E)$$

$$\frac{\Gamma \vdash M : \sigma \quad \Gamma \vdash M : \tau}{\Gamma \vdash M : \sigma \cap \tau} \, (\cap) \qquad \frac{}{\Gamma \vdash M : \omega} \, (\omega) \qquad \frac{\Gamma \vdash M : \sigma \quad \sigma \leq \tau}{\Gamma \vdash M : \tau} \, (\leq)$$

$$\frac{}{\Gamma \vdash \langle l_i = M_i \mid i \in I \rangle : \langle \rangle} \, (\langle \rangle) \qquad \frac{\Gamma \vdash M_k : \sigma \quad k \in I}{\Gamma \vdash \langle l_i = M_i \mid i \in I \rangle : \langle l_k : \sigma \rangle} \, (rec) \qquad \frac{\Gamma \vdash M : \langle l : \sigma \rangle}{\Gamma \vdash M.l : \sigma} \, (sel)$$

$$\frac{\Gamma \vdash M : \rho_1 \quad \Gamma \vdash R : \rho_2 \quad (*)}{\Gamma \vdash M \oplus R : \rho_1 + \rho_2} \, (+)$$

where $(*)$ in rule $(+)$ is the side condition: $lbl(R) = lbl(\rho_2)$.

Using Lemma 3.7.1, the following rule is easily shown to be admissible:

$$\frac{\Gamma \vdash M_j : \sigma_j \qquad \forall j \in J \subseteq I}{\Gamma \vdash \langle l_i = M_i \mid i \in I \rangle : \langle l_j : \sigma_i \mid j \in J \rangle} \, (rec')$$

Contrary to this, the side condition $(*)$ of rule $(+)$ is equivalent to "exact" record typing in [BC90], disallowing record subtyping in width. Such a condition is necessary for soundness of typing. Indeed, suppose that $\Gamma \vdash M_0 : \sigma$ and $\Gamma \vdash M_0' : \sigma_0'$ but $\Gamma \not\vdash M_0' : \sigma_0$; then without $(*)$ we could derive:

$$\frac{\Gamma \vdash \langle l_0 = M_0 \rangle : \langle l_0 : \sigma_0 \rangle \qquad \Gamma \vdash \langle l_0 = M_0', l_1 : \sigma_1 \rangle : \langle l_1 : \sigma_1 \rangle}{\Gamma \vdash \langle l_0 = M_0 \rangle \oplus \langle l_0 = M_0', l :_1: \sigma_1 \rangle : \langle l_0 : \sigma_0, l_1 : \sigma_1 \rangle}$$

from which we obtain that $\Gamma \vdash (\langle l_0 = M_0 \rangle \oplus \langle l_0 = M_0', l :_1: \sigma_1 \rangle).l_0 : \sigma_0$ breaking subject reduction, since $(\langle l_0 = M_0 \rangle \oplus \langle l_0 = M_0', l :_1: \sigma_1 \rangle).l_0 \longrightarrow^* M_0'$. The essential point is that proving that $\Gamma \vdash N : \langle l : \sigma \rangle$ doesn't imply that $l' \notin lbl(R')$ for any $l' \neq l$, which follows only by the uncomputable (not even recursively enumerable) statement that $\Gamma \not\vdash N : \langle l' : \omega \rangle$, a negative information.

This explains the restriction to record terms as the second argument of $\oplus$: in fact, allowing $M \oplus N$ to be well formed for an arbitrary N we might have $N \equiv x$ in $\lambda x. (M \oplus x)$. But extending lbl to all terms in Λ_R is not possible without severely limiting the expressiveness of the assignment system. In fact to say that $lbl(N) = lbl(R)$ if $N \longrightarrow^* R$ would make the lbl function non computable; on the other hand putting $lbl(x) = \varnothing$, which is the only reasonable and conservative choice as we do not know possible substitutions for x in $\lambda x. (M \oplus x)$, implies that the latter term has type $\omega \to \omega = \omega$ at best.

As a final remark, let us observe that we do not adopt exact typing of records in general, but only for typing the right-hand side of $\oplus$-terms, a feature that will be essential when typing mixins.

The following two lemmas are standard after [BCDC83].

Lemma 3.12.
$$\bigcap_{i \in I}(\sigma_i \to \tau_i) \leq \sigma \to \tau \Rightarrow \exists J \subseteq I. \, \sigma \leq \bigcap_{j \in J} \sigma_j \,\, \& \,\, \bigcap_{j \in J} \tau_j \leq \tau$$

Proof. By induction over the proof of $\bigcap_{i \in I}(\sigma_i \to \tau_i) \leq \sigma \to \tau$.   □

Lemma 3.13. *The following rule is admissible:*

$$\frac{\Gamma, x : \tau \vdash M : \sigma \quad \tau' \leq \tau}{\Gamma, x : \tau' \vdash M : \sigma}$$

Proof. By induction over the derivation of $\Gamma, x : \tau \vdash M : \sigma$. The only non-trivial case is when $\Gamma, x : \tau \vdash M : \sigma$ is an instance of (Ax) and $M \equiv x$. In this case $\sigma = \tau$ and we replace the axiom by the inference:

$$\frac{\Gamma, x : \tau' \vdash x : \tau' \quad \tau' \leq \tau}{\Gamma, x : \tau' \vdash x : \tau} \ (\leq)$$

$\square$

Lemma 3.14 (Generation). *Let $\sigma \neq \omega$:*

(1) $\Gamma \vdash x : \sigma \Leftrightarrow \exists \tau.\ x : \tau \in \Gamma\ \&\ \tau \leq \sigma,$
(2) $\Gamma \vdash \lambda x.M : \sigma \Leftrightarrow \exists I, (\sigma_i)_{i \in I}, (\tau_i)_{i \in I}.\ \Gamma, x : \sigma_i \vdash M : \tau_i\ \&\ \bigcap_{i \in I}(\sigma_i \to \tau_i) \leq \sigma,$
(3) $\Gamma \vdash MN : \sigma \Leftrightarrow \exists \tau.\ \Gamma \vdash M : \tau \to \sigma\ \&\ \Gamma \vdash N : \tau,$
(4) $\Gamma \vdash \langle l_i = M_i \mid i \in I \rangle : \sigma \Leftrightarrow \forall i \in I\ \exists \sigma_i.\ \Gamma \vdash M_i : \sigma_i\ \&\ \langle l_i : \sigma_i \mid i \in I \rangle \leq \sigma,$
(5) $\Gamma \vdash M.l : \sigma \Leftrightarrow \Gamma \vdash M : \langle l : \sigma \rangle,$
(6) $\Gamma \vdash M \oplus R : \sigma \Leftrightarrow \exists \rho_1, \rho_2.\ \Gamma \vdash M : \rho_1\ \&\ \Gamma \vdash R : \rho_2\ \&\ lbl(R) = lbl(\rho_2)\ \&\ \rho_1 + \rho_2 \leq \sigma.$

Proof. All the if parts are obvious. For the only if parts first observe that there is a one-to-one correspondence between the term constructors and the rules in the type system but in case of rules (ω), $(\cap)$ and $(\leq)$. Indeed, rule $(\langle\rangle)$ is no exception as the typing $\Gamma \vdash \langle l_i = M_i \mid i \in I \rangle : \langle\rangle$ is derivable using (rec) and $(\leq)$ in all cases but when $I = \varnothing$, in which case $(\langle\rangle)$ becomes the axiom $\Gamma \vdash \langle\rangle : \langle\rangle$ and $\langle\rangle$ is a nullary operator.

Disregarding rule (ω) because of the hypothesis $\sigma \neq \omega$, all the statements in this lemma depend on the remark that any derivation $\mathcal{D}$ of $\Gamma \vdash M : \sigma$ consists of a finite set of subderivations $\mathcal{D}_i$ of judgments $\Gamma \vdash M : \sigma_i$ such that each $\mathcal{D}_i$ ends by the rule corresponding to the main term constructor of M, and that in $\mathcal{D}$ all the inferences below the $\mathcal{D}_i$ are instances of either rule $(\cap)$ or $(\leq)$. By noting that inferences using $(\cap)$ and $(\leq)$ commute that is:

$$\frac{\dfrac{\Gamma \vdash M : \sigma' \quad \sigma' \leq \sigma}{\Gamma \vdash M : \sigma} \ (\leq) \quad \Gamma \vdash M : \tau}{\Gamma \vdash M : \sigma \cap \tau} \ (\cap) \qquad \text{becomes} \qquad \frac{\dfrac{\Gamma \vdash M : \sigma' \quad \Gamma \vdash M : \tau}{\Gamma \vdash M : \sigma' \cap \tau} \ (\cap) \quad \sigma' \cap \tau \leq \sigma \cap \tau}{\Gamma \vdash M : \sigma \cap \tau} \ (\leq)$$

we can freely assume that all $(\cap)$ inferences precede $(\leq)$ rules, concluding that $\bigcap_i \sigma_i \leq \sigma$.

Given that all the statements of this lemma are deduced by reading backward rules (Ax), $(\to I)$, $(\to E)$, $(\langle\rangle)$, (rec), (sel) and $(+)$. All cases are either standard from [BCDC83] or are easy extensions thereof, but that of rule $(+)$ in (6) above. In this case we know that in the derivation of $\Gamma \vdash M \oplus R : \sigma$ there are subderivations ending by the inference:

$$\frac{\Gamma \vdash M : \rho_{i,1} \quad \Gamma \vdash R : \rho_{i,2} \quad lbl(\rho_{i,2}) = lbl(R)}{\Gamma \vdash M \oplus R : \rho_{i,1} + \rho_{i,2}} \ (+)$$

where the side condition $lbl(\rho_{i,2}) = lbl(R)$ holds for all $i \in I$; then by the remark above we have that $\bigcap_{i \in I}(\rho_{i,1} + \rho_{i,2}) \leq \sigma$. Now taking $\rho_1 = \bigcap_{i \in I} \rho_{i,1}$ and $\rho_2 = \bigcap_{i \in I} \rho_{i,2}$ we have that:

$$\begin{aligned}
\sigma\ &\geq\ \bigcap_{i \in I}(\rho_{i,1} + \rho_{i,2}) \\
&\geq\ \bigcap_{i \in I}(\rho_1 + \rho_{i,2}) &&\text{by Def. 3.5.9, since } \rho_{i,1} \geq \rho_1 \text{ for all } i \in I \\
&=\ \rho_1 + \rho_2 &&\text{by 3.10.4 since } lbl(\rho_2) = lbl(\rho_{i,2}) \text{ for all } i \in I.
\end{aligned}$$

$\square$

Lemma 3.15 (Substitution).

$$\Gamma, x : \sigma \vdash M : \tau \ \& \ \Gamma \vdash N : \sigma \Rightarrow \Gamma \vdash M\{N/x\} : \tau.$$

Proof. By induction over the derivation of $\Gamma, x : \sigma \vdash M : \tau$. Observe that the writing $\Gamma, x : \sigma$ implies that $x \notin \Gamma$; on the other hand if $\Gamma \vdash N : \sigma$ is derivable then $fv(N) \subseteq \Gamma$, hence $x \notin fv(N)$. ◻

Theorem 3.16 (Subject reduction). $\Gamma \vdash M : \sigma \ \& \ M \longrightarrow N \Rightarrow \Gamma \vdash N : \sigma$.

Proof. The proof is by cases of reduction rules, using Lemma 3.14.

(1) Case $M \equiv (\lambda x.M')N'$. Then $N \equiv M'\{N'/x\}$ by rule (β). By Lemma 3.14.3, there exists τ such that $\Gamma \vdash (\lambda x.M') : \tau \to \sigma$ and $\Gamma \vdash N' : \tau$. By Lemma 3.14.2, there exist I, σ_i, τ_i such that $\Gamma, x : \tau_i \vdash M' : \sigma_i$ for all $i \in I$ and $\bigcap_{i \in I}(\tau_i \to \sigma_i) \leq \tau \to \sigma$. By Lemma 3.12 this implies that there is $J \subseteq I$ such that $\tau \leq \bigcap_{j \in J} \tau_j$ and $\bigcap_{j \in J} \sigma_j \leq \sigma$. By Lemma 3.13 this implies that $\Gamma, x : \tau \vdash M' : \sigma_j$ for all $j \in J$, so that by rule $(\cap)$ we have $\Gamma, x : \tau \vdash M' : \bigcap_{j \in J} \sigma_j$ and hence $\Gamma, x : \tau \vdash M' : \sigma$ by $(\leq)$. From this and $\Gamma \vdash N' : \tau$ we get $\Gamma \vdash M'\{N'/x\} : \sigma$ by Lemma 3.15.

(2) Case $M \equiv \langle l_i = M_i \mid i \in I \rangle.l_j$. Then $N \equiv M_j$ for some $j \in I$ by rule (sel). By Lemma 3.14.5 $\Gamma \vdash M : \langle l_j : \sigma \rangle$, hence by 3.14.4 for all $i \in I$ there exists σ_i such that $\Gamma \vdash M_i : \sigma_i$ and $\langle l_i : \sigma_i \mid i \in I \rangle \leq \langle l_j : \sigma \rangle$. Now by the fact that $j \in I$ and by Lemma 3.7.1 it follows that $\Gamma \vdash M_j : \sigma_j$ and $\sigma_j \leq \sigma$, hence $\Gamma \vdash M_j : \sigma$ by rule $(\leq)$.

(3) Case $M \equiv \langle l_i = M_i \mid i \in I \rangle \oplus \langle l_j = N_j \mid j \in J \rangle$. Then by rule $(\oplus)$:

$$N \equiv \langle l_i = M_i, \ l_j = N_j \mid i \in I \smallsetminus J, \ j \in J \rangle.$$

By Lemma 3.14.6 there exist the record types ρ_1, ρ_2 such that:
   (a) $\Gamma \vdash \langle l_i = M_i \mid i \in I \rangle : \rho_1$,
   (b) $\Gamma \vdash \langle l_j = N_j \mid j \in J \rangle : \rho_2$,
   (c) $lbl(\rho_2) = lbl(\langle l_j = N_j \mid j \in J \rangle) = \{l_j \mid j \in J\}$,
   (d) $\rho_1 + \rho_2 \leq \sigma$.

By (3a) and (3b) and by Lemma 3.14.4 we have that for all $i \in I$ there exist σ_i such that $\Gamma \vdash M_i : \sigma_i$, with $\langle l_i : \sigma_i \mid i \in I \rangle \leq \rho_1$, and similarly for all $j \in J$ there are τ_j such that $\Gamma \vdash N_j : \tau_j$ and $\langle l_j : \tau_j \mid j \in J \rangle \leq \rho_2$.

Since $\Gamma \vdash M_i : \sigma_i$ for all $i \in I$, a fortiori it holds for all $i \in I \smallsetminus J$. On the other hand, by Lemma 3.7.4 we know that $\rho_2 = \langle l_k : \tau'_k \mid k \in K \rangle$ for some K and τ'_k, so that $\langle l_j : \tau_j \mid j \in J \rangle \leq \rho_2$ implies $J \supseteq K$ and $\tau_k \leq \tau'_k$ for all $k \in K$ by Lemma 3.7.1. It follows that $J = K$ by (3c), and $\Gamma \vdash N_j : \tau'_j$ for all $j \in J$ by rule $(\leq)$. Then by rule (rec) and $(\cap)$ we conclude that

$$\Gamma \vdash N : \langle l_i : \sigma_i, \ l_j : \tau'_j \mid i \in I \smallsetminus J, \ j \in J \rangle.$$

Now $\langle l_i : \sigma_i, \ l_j : \tau'_j \mid i \in I \smallsetminus J, \ j \in J \rangle = \langle l_i : \sigma_i \mid i \in I \rangle + \rho_2$ by Lemma 3.7.3, but since

$$\langle l_i : \sigma_i \mid i \in I \rangle \leq \rho_1$$

we have by Definition 3.5.9 that $\langle l_i : \sigma_i \mid i \in I \rangle + \rho_2 \leq \rho_1 + \rho_2$; then we conclude that $\Gamma \vdash N : \sigma$ by (3d) and rule $(\leq)$. ◻

3.2. **Class and Mixin combinators.** The following definition of classes and mixins is inspired by [CHC90] and [BC90] respectively, though with some departures to be discussed below. To make the description more concrete, in the examples we add constants to Λ_R.

Recall that a *combinator* is a term in Λ_R^0, namely a closed term. Let $\mathbf{Y}$ be Curry's fixed point combinator: $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ (the actual definition of $\mathbf{Y}$ is immaterial however, since all fixed point combinators have the same Böhm tree as a consequence of [Bar84] Lemma 6.5.3, and hence have the same types in **BCD** by the approximation theorem: see e.g. [BDS13] Theorem 12.1.17).

Definition 3.17. Let myClass, state and argClass be (term) variables and $\mathbf{Y}$ be a fixed point combinator; then we define the following sets of combinators:

Class: C ::= $\mathbf{Y}(\lambda\,\mathsf{myClass}\ \lambda\,\mathsf{state}.\,\langle l_i = N_i \mid i \in I\rangle)$

Mixin: M ::= $\lambda\,\mathsf{argClass}.\,\mathbf{Y}(\lambda\,\mathsf{myClass}\ \lambda\,\mathsf{state}.\,(\mathsf{argClass}\ \mathsf{state}) \oplus \langle l_i = N_i \mid i \in I\rangle)$

We define $\mathcal{C}$ and $\mathcal{M}$ as the sets of classes and mixins respectively.

To illustrate this definition let us use the abbreviation `let` $x = N$ `in` M for $M\{N/x\}$. Then a class combinator $C \in \mathcal{C}$ can be written in a more perspicuous way as follows:

$$C \equiv \mathbf{Y}(\lambda\,\mathsf{myClass}\ \lambda\,\mathsf{state}.\ \mathtt{let}\ \mathsf{self} = (\mathsf{myClass}\ \mathsf{state})\ \mathtt{in}\ \langle l_i = N_i \mid i \in I\rangle). \qquad (3.3)$$

A *class* is the fixed point of a function, the *class definition*, mapping a recursive definition of the class itself and a state S, that is the value or a record of values in general, for the instance variables of the class, into a record $\langle l_i = N_i \mid i \in I\rangle$ of *methods*. A class C is instantiated to an *object* $O \equiv C\,S$ by applying the class C to a state S. Hence we have:

$$O \equiv C\,S \longrightarrow^* \mathtt{let}\ \mathsf{self} = (C\,S)\ \mathtt{in}\ \langle l_i = N_i \mid i \in I\rangle,$$

where the variable self is used in the method bodies N_i to call other methods from the same object. Note that the recursive parameter myClass might occur in the N_i in subterms other than (myClass state), and in particular $N_i\{C/\mathsf{myClass}\}$ might contain a subterm $C\,S'$, where S' is a state possibly different than S; even C itself might be returned as the value of a method. Classes are the same as in [CHC90] §4, but for the explicit identification of self with (myClass state).

We come now to typing of classes. Let $R = \langle l_i = N_i \mid i \in I\rangle$, and suppose that $C \equiv \mathbf{Y}(\lambda\,\mathsf{myClass}\ \lambda\,\mathsf{state}.\,R) \in \mathcal{C}$. To type C we must find a type σ (a type of its state) and a sequence of types $\rho_1, \ldots, \rho_n \in \mathbb{T}_{\langle\rangle}$ such that for all $0 < i < n$:

$$\mathsf{myClass} : \sigma \to \rho_i, \mathsf{state} : \sigma \vdash R : \rho_{i+1}.$$

Note that this is always possible for any n: in the worst case, we can take $\rho_i = \langle l_i : \omega \mid i \in I\rangle$ for all $0 < i \le n$. In general one has more expressive types, depending on the typings of the N_i in R (see example 3.18 below). It follows that:

$$\vdash \lambda\,\mathsf{myClass}\ \lambda\,\mathsf{state}.\,R : (\omega \to (\sigma \to \rho_1)) \cap \bigcap_{i=1}^{n-1}((\sigma \to \rho_i) \to (\sigma \to \rho_{i+1})),$$

and therefore, by using the fact that $\vdash \mathbf{Y} : (\omega \to \tau_1) \cap \cdots \cap (\tau_{n-1} \to \tau_n) \to \tau_n$ for arbitrary types $\tau_1, \ldots, \tau_n$ (see e.g. [BDS13], p. 586), we conclude that the typing of classes has the following shape (where $\rho = \rho_n$):

$$\vdash C \equiv \mathbf{Y}(\lambda\,\mathsf{myClass}\ \lambda\,\mathsf{state}.\,\langle l_i = N_i \mid i \in I\rangle) : \sigma \to \rho \qquad (3.4)$$

In conclusion the type of a class C is the arrow from the type of the state σ to a type ρ of its instances.

Example 3.18. The class Num has a method *get* returning the current state and a method *succ* returning the state of the instance object incremented by one. Beside there is a third method *set* whose body is the identity function:

$$\text{Num} = \mathbf{Y}(\lambda\mathsf{myClass}.\lambda\mathsf{state}.$$

$$\mathtt{let}\ \mathsf{self} = \mathsf{myClass\ state}\ \mathtt{in}$$

$$\langle get = \mathsf{state}, set = \lambda\mathsf{state}'.\mathsf{state}', succ = \mathsf{self}.set(\mathsf{self}.get+1)\rangle$$

There are several ways in which functional state update can be implemented. One possibility for a method to update the current state of the object functionally is to return an updated instance of the object. As will become clear later on, in the context of mixin applications it is unclear whether the updated object has to be an instance of the early-bound inner class (Num in the above example) or an instance of the class modified by mixin applications. The former option loses information whenever an object that is an instance of a class modified by mixin applications is updated. The latter option breaks compatibility whenever a mixin incompatibly overwrites a method using record merge, which will be discussed later on. In conclusion, the decision regarding the instantiation of the updated object cannot be made a-priori, but depends on the context in which the method that updates the object is called. Since in the functional setting two object instances of one class are distinguished only by the underlying state, a method, such as *set*, that modifies the underlying object state returns the updated state and leaves instantiation to the caller.

To Type Num, let us call

$$R \equiv \langle get = \mathsf{state}, set = \lambda\mathsf{state}'.\mathsf{state}', succ = \mathsf{self}.set(\mathsf{self}.get+1)\rangle$$

where $\mathsf{self} \equiv (\mathsf{myClass\ state})$. Now we can deduce:

$$\mathsf{myClass} : \omega, \mathsf{state} : \mathrm{Int} \vdash R : \langle get : \mathrm{Int}, set : \mathrm{Int} \to \mathrm{Int}, succ : \omega\rangle = \rho_1.$$

since $\mathsf{myClass} : \mathrm{Int} \to \omega, \mathsf{state} : \mathrm{Int} \vdash \mathsf{myClass\ state} : \omega$ and therefore $\mathsf{self}.set(\mathsf{self}.get+1)$ is just typable by ω; on the other hand the type of *set* is any type of the identity, so that $\mathrm{Int} \to \mathrm{Int}$ is a possibility.

Given ρ_1 we consider the new assumption $\mathsf{myClass} : \mathrm{Int} \to \rho_1$ by which we obtain the typing $\mathsf{self} \equiv \mathsf{myClass\ state} : \rho_1$ that is enough to get $\mathsf{self}.set(\mathsf{self}.get+1) : \mathrm{Int}$ and hence:

$$\mathsf{myClass} : \mathrm{Int} \to \rho_1, \mathsf{state} : \mathrm{Int} \vdash R : \langle get : \mathrm{Int}, set : \mathrm{Int} \to \mathrm{Int}, succ : \mathrm{Int}\rangle = \rho_2.$$

Eventually by (3.4) we conclude that $\text{Num} : \mathrm{Int} \to \rho_2$.

In case of Num we have reached the type ρ_2 that doesn't contain any occurrence of ω in a finite number of steps. This is because method *succ* of class Num returns a number; let us consider a variant Num' of Num having a method *succ* that returns a new instance of the class, whose state has been incremented:

$$\mathrm{Num}' = \mathbf{Y}(\lambda\mathsf{myClass}.\lambda\mathsf{state}.$$
$$\quad\quad \mathtt{let}\ \mathsf{self} = \mathsf{myClass}\ \mathsf{state}\ \mathtt{in}$$
$$\quad\quad\quad \langle get = \mathsf{state}, succ = \mathsf{myClass}(\mathsf{self}.get + 1)\rangle$$

Let us call $R' \equiv \langle get = \mathsf{state}, succ = \mathsf{myClass}(\mathsf{self}.get + 1)\rangle$, where $\mathsf{self} \equiv (\mathsf{myClass}\ \mathsf{state})$ as above. Then

$$\mathsf{myClass} : \mathrm{Int} \to \omega, \mathsf{state} : \mathrm{Int} \vdash R' : \langle get : \mathrm{Int}, succ : \omega\rangle = \rho_1'.$$

By assuming $\mathsf{myClass} : \mathrm{Int} \to \rho_1'$ and $\mathsf{state} : \mathrm{Int}$, we have that $\mathsf{self} : \rho_1'$ and hence $\mathsf{self}.get : \mathrm{Int}$. It follows that $\mathsf{myClass}(\mathsf{self}.get + 1) : \rho_1'$ so that

$$\mathsf{myClass} : \mathrm{Int} \to \rho_1', \mathsf{state} : \mathrm{Int} \vdash R' : \langle get : \mathrm{Int}, succ : \rho_1'\rangle = \rho_2'.$$

In general, putting $\rho_0' = \omega$ and $\rho_{i+1}' = \langle get : \mathrm{Int}, succ : \rho_i'\rangle$ we have for all i:

$$\mathsf{myClass} : \mathrm{Int} \to \rho_i', \mathsf{state} : \mathrm{Int} \vdash R' : \langle get : \mathrm{Int}, succ : \rho_i'\rangle = \rho_{i+1}'.$$

From this by (3.4) we conclude that $\mathrm{Num}' : \mathrm{Int} \to \rho_i'$ for all i. We note that this time, differently than in case of Num, we cannot get rid of occurrences ω in the ρ_i'.

A variant of this typing uses the type constants Odd and Even called semantic types in [Reh13] where they are used in the intersection types $\mathrm{Int} \cap \mathsf{Odd}$ or $\mathrm{Int} \cap \mathsf{Even}$ (see also below Section 5.3). Let us suppose that we have axioms $x : \mathrm{Int} \cap \mathsf{Odd} \vdash x + 1 : \mathrm{Int} \cap \mathsf{Even}$ and $x : \mathrm{Int} \cap \mathsf{Even} \vdash x + 1 : \mathrm{Int} \cap \mathsf{Odd}$ added to the typing system. Then a more interesting type for Num' is

$$\mathrm{Num}' : \quad (\mathrm{Int} \cap \mathsf{Odd} \to \langle get : \mathrm{Int} \cap \mathsf{Odd}, succ : \langle get : \mathrm{Int} \cap \mathsf{Even}\rangle\rangle) \cap$$
$$(\mathrm{Int} \cap \mathsf{Even} \to \langle get : \mathrm{Int} \cap \mathsf{Even}, succ : \langle get : \mathrm{Int} \cap \mathsf{Odd}\rangle\rangle).$$

Although Num' is closer to what is done with object-oriented programming, in the context of mixin application, introduced in the remainder of this section, we shall not consider this kind of recursive definition in this work, as previously explained.

A *mixin* $M \in \mathcal{M}$ is a combinator such that, if $C \in \mathcal{C}$ then $M\,C$ reduces to a new class $C' \in \mathcal{C}$. Writing M in a more explicit way we obtain:

$$M \equiv \lambda\,\mathsf{argClass}.\,\mathbf{Y}(\lambda\,\mathsf{myClass}\,\lambda\,\mathsf{state}.\,\mathtt{let}\ \mathsf{super} = (\mathsf{argClass}\ \mathsf{state})\ \mathtt{in}$$
$$\mathtt{let}\ \mathsf{self} = (\mathsf{myClass}\ \mathsf{state})\ \mathtt{in}$$
$$\mathsf{super} \oplus \langle l_i = N_i \mid i \in I\rangle)$$

In words, a mixin merges an instance $C\,S$ of the input class C with a new state S together with a *difference* record $R \equiv \langle l_i = N_i \mid i \in I\rangle$, that would be written $\Delta(C\,S)$ in terms of [BC90]. Note that our mixins are not the same as class modificators (also called wrappers e.g. in [Bra92]). Wrappers bind super to the unmodified class definition applied to self *without* taking the fixed point. In our case, super is simply an instance of the unmodified class. The effect is that we have a static (or early) binding instead of dynamic (or late) binding of self.

Let $M \equiv \lambda\,\mathsf{argClass}.\,\mathbf{Y}(\lambda\,\mathsf{myClass}\,\lambda\,\mathsf{state}.\,(\mathsf{argClass}\ \mathsf{state}) \oplus R) \in \mathcal{M}$; to type M we have to find types $\sigma^1, \sigma^2, \rho^1$ and a sequence $\rho_1^2, \dots, \rho_n^2 \in \mathbb{T}_{\langle\rangle}$ of record types such that for all

$1 \leq i < n$ it is true that $lbl(R) = lbl(\rho_i^2)$ and setting

$\Gamma_0 = \{\text{argClass} : \sigma^1 \to \rho^1, \text{myClass} : \omega, \text{state} : \sigma^1 \cap \sigma^2\}$

$\Gamma_i = \{\text{argClass} : \sigma^1 \to \rho^1, \text{myClass} : (\sigma^1 \cap \sigma^2) \to \rho^1 + \rho_i^2, \text{state} : \sigma^1 \cap \sigma^2\}$ for all $1 \leq i < n$

we may deduce for all $0 \leq i < n$:

$$\cfrac{\cfrac{\cfrac{\Gamma_i \vdash \text{state} : \sigma^1 \cap \sigma^2}{\Gamma_i \vdash \text{state} : \sigma^1}\ (\leq)}{\Gamma_i \vdash \text{argClass state} : \rho^1}\ (\to E) \qquad \Gamma_i \vdash R : \rho_{i+1}^2 \qquad lbl(R) = lbl(\rho_{i+1}^2)}{\Gamma_i \vdash (\text{argClass state}) \oplus R : \rho^1 + \rho_{i+1}^2}\ (+)$$

Hence for all $0 \leq i < n$ we can derive the typing judgment:

$\text{argClass} : \sigma^1 \to \rho^1 \vdash \lambda\,\text{myClass}\,\lambda\,\text{state}.\,(\text{argClass state}) \oplus R :$

$$((\sigma^1 \cap \sigma^2) \to (\rho^1 + \rho_i^2)) \to (\sigma^1 \cap \sigma^2) \to (\rho^1 + \rho_{i+1}^2)$$

and therefore, by reasoning as for classes, we get (setting $\rho^2 = \rho_n^2$):

$\vdash M \equiv \lambda\,\text{argClass}.\,\mathbf{Y}(\lambda\,\text{myClass}\,\lambda\,\text{state}.\,(\text{argClass state}) \oplus R) :$

$$(\sigma^1 \to \rho^1) \to (\sigma^1 \cap \sigma^2) \to (\rho^1 + \rho^2) \quad (3.5)$$

Spelling out this type, we can say that σ^1 is a type of the state of the argument-class of M; $\sigma^1 \cap \sigma^2$ is the type of the state of the resulting class, that refines σ^1. Type ρ^1 expresses the requirements of M about the methods of the argument-class, i.e. what is assumed to hold for the usages of super and argClass in R to be properly typed; $\rho^1 + \rho^2$ is a type of the record of methods of the refined class, resulting from the merge of the methods of the argument-class with those of the difference R; since in general there will be overridden methods, whose types might be incompatible, the + type constructor cannot be replaced by intersection. On the other hand, M preserves typings of any label that is not in R.

Lemma 3.19. *For any σ and $\rho = \langle l : \tau \rangle$ such that $l \notin lbl(R)$ we have*

$\vdash M \equiv \lambda\,\text{argClass}.\,\mathbf{Y}(\lambda\,\text{myClass}\,\lambda\,\text{state}.\,(\text{argClass state}) \oplus R) : (\sigma \to \rho) \to (\sigma \to \rho)$

Proof. Let $\{l_1, \ldots, l_n\} = lbl(R)$. We have $\vdash R : \langle l_1 : \omega, \ldots, l_n : \omega \rangle$. Since $l \notin lbl(R)$ we obtain $\rho + \langle l_1 : \omega, \ldots, l_n : \omega \rangle = \langle l : \tau, l_1 : \omega, \ldots, l_n : \omega \rangle \leq \rho$. Using the above general type derivation with $\sigma^1 = \sigma^2 = \sigma$, $\rho^1 = \rho$, $\rho^2 = \langle l_1 : \omega, \ldots, l_n : \omega \rangle$ and $\vdash \mathbf{Y} : (\omega \to (\sigma \to \rho)) \to (\sigma \to \rho)$ we obtain $\vdash M : (\sigma \to \rho) \to (\sigma \to \rho + \rho^2) \leq (\sigma \to \rho) \to (\sigma \to \rho)$. $\qquad \square$

Example 3.20. Let us consider the mixin Comparable adding a method *compare* to its argument class, which is supposed to have a method *get*:

Comparable $= \lambda\text{argClass}.\mathbf{Y}(\lambda\text{myClass}.\lambda\text{state}.$

```
let super = argClass state in
let self = myClass state in
```
$\quad\quad$ super $\oplus \langle compare = \lambda o.(o.get == \text{self}.get) \rangle)$

We write == for a suitable equality operator, which is distinct from the symbol = used in our calculus to associate labels to their values in a record.

For the sake of readability, we take the types σ^1 and σ^2 of the respective states of the argument class and the resulting class to be just Int and simply write Int for Int $\cap$ Int which is the type we assume for **state**. If we set

$$\Gamma_0 = \{\mathsf{argClass} : \mathrm{Int} \to \langle get : \mathrm{Int}\rangle, \mathsf{myClass} : \mathrm{Int} \to \omega, \mathsf{state} : \mathrm{Int}\}$$

we have $\Gamma_0 \vdash \mathsf{super} : \langle get : \mathrm{Int}\rangle$ but $\Gamma_0 \vdash \mathsf{self} : \omega$ so that $\Gamma_0 \vdash \lambda o.(o.get == \mathsf{self}.get) : \tau \to \omega$ for any type τ given to o; but $\tau \to \omega = \omega$ by Definition 3.4, so that we conclude

$$\Gamma_0 \vdash \mathsf{super} \oplus \langle compare = \lambda o.(o.get == \mathsf{self}.get)\rangle : \langle get : \mathrm{Int}\rangle + \langle compare : \omega\rangle.$$

Now, taking

$$\Gamma_1 = \{\mathsf{argClass} : \mathrm{Int} \to \langle get : \mathrm{Int}\rangle, \mathsf{myClass} : \mathrm{Int} \to \langle get : \mathrm{Int}\rangle + \langle compare : \omega\rangle, \mathsf{state} : \mathrm{Int}\}$$

and using the fact that $\langle get : \mathrm{Int}\rangle + \langle compare : \omega\rangle = \langle get : \mathrm{Int}\rangle \cap \langle compare : \omega\rangle \leq \langle get : \mathrm{Int}\rangle$ we have $\Gamma_1 \vdash \mathsf{self} : \langle get : \mathrm{Int}\rangle$, that implies $\Gamma_1 \vdash \mathsf{self}.get : \mathrm{Int}$ and therefore $\Gamma_1 \vdash \lambda o.(o.get == \mathsf{self}.get) : \langle get : \mathrm{Int}\rangle \to \mathrm{Bool}$. By (3.5) we conclude that Comparable has type

$$(\mathrm{Int} \to \langle get : \mathrm{Int}\rangle) \to \mathrm{Int} \to \langle get : \mathrm{Int}\rangle + \langle compare : \langle get : \mathrm{Int}\rangle \to \mathrm{Bool}\rangle.$$

We finally observe that if the argument class C has type $\mathrm{Int} \to \langle get : \mathrm{Int}, compare : \tau\rangle$ for some type τ, then we have that $\mathrm{Int} \to \langle get : \mathrm{Int}, compare : \tau\rangle \leq \mathrm{Int} \to \langle get : \mathrm{Int}\rangle$, so that the class $C' \equiv \mathsf{Comparable}\,C$ has the correct type $\mathrm{Int} \to \langle get : \mathrm{Int}\rangle + \langle compare : \langle get : \mathrm{Int}\rangle \to \mathrm{Bool}\rangle$ as the method $\mathsf{super}.compare$ is overridden in $\mathsf{super}\oplus\langle compare = \lambda o.(o.get == \mathsf{self}.get)\rangle$. This is because, even by keeping the typing $\mathsf{super} : \langle get : \mathrm{Int}, compare : \tau\rangle$ through the derivation, we obtain that $\mathsf{super} \oplus \langle compare = \lambda o.(o.get == \mathsf{self}.get)\rangle$ has type

$$\langle get : \mathrm{Int}, compare : \tau\rangle + \langle compare : \langle get : \mathrm{Int}\rangle \to \mathrm{Bool}\rangle = \langle get : \mathrm{Int}, compare : \langle get : \mathrm{Int}\rangle \to \mathrm{Bool}\rangle$$

which is the same as $\langle get : \mathrm{Int}\rangle + \langle compare : \langle get : \mathrm{Int}\rangle \to \mathrm{Bool}\rangle$. For a more general typing of Comparable see Appendix B.

4. Bounded Combinatory Logic

Our main goal is synthesize meaningful mixin compositions. For this purpose, we use the logical programming language given by inhabitation in $\mathsf{BCL}_k(\mathbb{T}_\mathbb{C})$ (bounded combinatory logic with constructors). Conveniently, necessary type information can be inferred from intersection types for Λ_R. $\mathsf{BCL}_k(\mathbb{T}_\mathbb{C})$ is an extension of bounded combinatory logic $\mathsf{BCL}_k(\to, \cap)$ [DMRU12] by covariant constructors. Constructors can be utilized to encode record types and useful features of $+$ while extending the existing synthesis framework (CL)S [BDD+14]. In this section, we present $\mathsf{BCL}_k(\mathbb{T}_\mathbb{C})$ along with its fundamental properties, in particular decidability of inhabitation.

4.1. $\mathsf{BCL}_k(\mathbb{T}_\mathbb{C})$. *Combinatory terms* are formed by application of combinators from a *repository* (combinatory logic context) Δ.

Definition 4.1 (Combinatory Term). $E, E' ::= C \mid (E\,E')$ where $C \in dom(\Delta)$

We create repositories of typed combinators that can be considered logic programs for the existing synthesis framework (CL)S [BDD+14] to reason about semantics of such compositions. The underlying type system of (CL)S is the intersection type system **BCD** [BCDC83], which we extend by covariant constructors. The extended type system $\mathbb{T}_\mathbb{C}$, while suited for synthesis, is flexible enough to encode record types and features of $+$.

Definition 4.2 (Intersection Types with Constructors $\mathbb{T}_\mathbb{C}$). The set $\mathbb{T}_\mathbb{C}$ is given by:

$$\mathbb{T}_\mathbb{C} \ni \sigma, \tau, \tau_1, \tau_2 \ ::= a \mid \alpha \mid \omega \mid \tau_1 \to \tau_2 \mid \tau_1 \cap \tau_2 \mid c(\tau)$$

where a ranges over constants, α over type variables and c over unary constructors $\mathbb{C}$.

$\mathbb{T}_\mathbb{C}$ adds the following two subtyping axioms to the **BCD** system (cf. Definition 3.4)

$$\tau_1 \le \tau_2 \Rightarrow c(\tau_1) \le c(\tau_2) \qquad c(\tau_1) \cap c(\tau_2) \le c(\tau_1 \cap \tau_2)$$

The additional axioms ensure *constructor distributivity*, i.e., $c(\tau_1) \cap c(\tau_2) = c(\tau_1 \cap \tau_2)$.

The notion of *level* of a type is used as a bound to ensure decidability of inhabitation, or equivalently termination of logic programs in (CL)S.

Definition 4.3 (Level). We define the *level* of a type as follows:

$$\text{level}(\omega) = \text{level}(a) = \text{level}(\alpha) = 0 \qquad\qquad \text{level}(c(\tau)) = 1 + \text{level}(\tau)$$

$$\text{level}(\sigma \to \tau) = 1 + \max(\text{level}(\sigma), \text{level}(\tau)) \qquad \text{level}(\sigma \cap \tau) = \max(\text{level}(\sigma), \text{level}(\tau))$$

We define the level of a substitution S as $\text{level}(S) = \max\{\text{level}(S(\alpha)) \mid \alpha \in dom(S)\}$.

Definition 4.4 (Type Assignment $\mathsf{BCL}_k(\mathbb{T}_\mathbb{C})$).

$$\frac{C : \tau \in \Delta \qquad \text{level}(S) \le k}{\Delta \vdash_k C : S(\tau)} \text{ (Var)} \qquad \frac{\Delta \vdash_k E : \sigma \to \tau \qquad \Delta \vdash_k E' : \sigma}{\Delta \vdash_k EE' : \tau} \text{ } (\to E)$$

$$\frac{\Delta \vdash_k E : \sigma \quad \Delta \vdash_k E : \tau}{\Delta \vdash_k E : \sigma \cap \tau} \text{ } (\cap) \qquad\qquad \frac{\Delta \vdash_k E : \sigma \qquad \sigma \le \tau}{\Delta \vdash_k E : \tau} \text{ } (\le)$$

For a set of typed combinators Δ and a type $\tau \in \mathbb{T}_\mathbb{C}$ we say τ is inhabited in Δ, if there exists a combinatory term E and a $k \in \mathbb{N}$ such that $\Delta \vdash_k E : \tau$.

4.2. **$\mathsf{BCL}_k(\mathbb{T}_\mathbb{C})$ Inhabitation.** In this section we extend the understanding of $\mathsf{BCL}(\to, \cap)$ inhabitation [DMRU12] to constructors. First, we extend the necessary notions of *paths* and *organized types* to constructors in the following way.

Definition 4.5 (Path). A *path* π is a type of the form: $\pi ::= a \mid \alpha \mid \sigma \to \pi \mid c(\omega) \mid c(\pi)$, where α is a variable, τ is a type, c is a constructor and a is a constant.

Definition 4.6 (Paths in τ). Given a type $\tau \in \mathbb{T}_\mathbb{C}$, the set $\mathbb{P}(\tau)$ of paths in τ is defined as

$$\mathbb{P}(a) = \{a\} \qquad\qquad \mathbb{P}(\sigma \to \tau) = \{\sigma \to \pi \mid \pi \in \mathbb{P}(\tau)\}$$

$$\mathbb{P}(\alpha) = \{\alpha\} \qquad\qquad \mathbb{P}(\sigma \cap \tau) = \mathbb{P}(\sigma) \cup \mathbb{P}(\tau)$$

$$\mathbb{P}(\omega) = \varnothing \qquad\qquad \mathbb{P}(c(\tau)) = \begin{cases} \{c(\omega)\} & \text{if } \mathbb{P}(\tau) = \varnothing \\ \{c(\pi) \mid \pi \in \mathbb{P}(\tau)\} & \text{else} \end{cases}$$

Observe that $\mathbb{P}(\tau) = \varnothing$ iff $\tau = \omega$.

Definition 4.7 (Organized Type). A type τ is called organized, if $\tau \equiv \bigcap_{i \in I} \pi_i$, where π_i for $i \in I$ are paths.

For brevity, we sometimes write $\bigcap \mathbb{P}(\tau)$ for $\pi_1 \cap \ldots \cap \pi_n$ where $\mathbb{P}(\tau) = \{\pi_1, \ldots, \pi_n\}$. If $\mathbb{P}(\tau) = \varnothing$, then we set $\bigcap \mathbb{P}(\tau) \equiv \omega$.

Lemma 4.8. *Given a type $\tau \in \mathbb{T}_\mathbb{C}$, the type $\bigcap \mathbb{P}(\tau)$ is organized with $\bigcap \mathbb{P}(\tau) = \tau$.*

The detailed proof by induction of Lemma 4.8 is in the appendix.

Lemma 4.9. *Given a type $\tau \in \mathbb{T}_\mathbb{C}$ we have $|\bigcap \mathbb{P}(\tau)| \leq |\tau|^2$, where $|\cdot|$ denotes the number of nodes in the syntax tree of a given type.*

Proof. By induction using Definition 4.6 we have $|\mathbb{P}(\tau)| \leq |\tau|$. The only non-trivial cases for the inductive proof of the main statement are (assuming $\mathbb{P}(\tau) \neq \varnothing$)

$$|\bigcap \mathbb{P}(c(\tau))| = \sum_{\pi \in \mathbb{P}(\tau)} (|\pi| + 2) - 1 = |\mathbb{P}(\tau)| + |\bigcap \mathbb{P}(\tau)| \leq |\tau| + |\tau|^2 \leq |c(\tau)|^2$$

$$|\bigcap \mathbb{P}(\sigma \to \tau)| = \sum_{\pi \in \mathbb{P}(\tau)} (|\sigma| + |\pi| + 2) - 1 = (|\sigma| + 1) \cdot |\mathbb{P}(\tau)| + |\bigcap \mathbb{P}(\tau)| \leq |\sigma \to \tau|^2 \qquad \square$$

Due to Lemma 4.9, for any intersection type there exists an equivalent organized intersection type computable in polynomial time. Note that organized types are not necessarily normalized [Hin82] or strict [Bak11]. However, organized types have the following property known from normalized types.

Lemma 4.10. *Given two types $\sigma, \tau \in \mathbb{T}_\mathbb{C}$, we have $\sigma \leq \tau$ iff for each path $\pi \in \mathbb{P}(\tau)$ there exists a path $\pi' \in \mathbb{P}(\sigma)$ such that $\pi' \leq \pi$ and*

- *If $\pi \equiv \alpha$ (resp. a), then $\pi' \equiv \alpha$ (resp. a).*
- *If $\pi \equiv \sigma_2 \to \tau_2$, then $\pi' \equiv \sigma_1 \to \tau_1$ such that $\sigma_2 \leq \sigma_1$ and $\tau_1 \leq \tau_2$.*
- *If $\pi \equiv c(\tau_1)$, then $\pi' \equiv c(\sigma_1)$ such that $\sigma_1 \leq \tau_1$.*

The detailed proof by induction of Lemma 4.10 is in the appendix.

We say a path π has the *arity* of at least m if $\pi \equiv \sigma_1 \to \ldots \to \sigma_m \to \tau$ for some $\sigma_1, \ldots \sigma_m, \tau \in \mathbb{T}_\mathbb{C}$. Additionally, for such a path π we define $\arg_i(\pi) = \sigma_i$ for $1 \leq i \leq m$ and $\mathrm{tgt}_m(\pi) = \tau$. We define $\mathbb{P}_m(\tau)$ as the set of paths in τ having arities of at least m.

$\mathsf{BCL}_k(\to, \cap)$ inhabitation is $(k+2)$-ExpTime complete [DMRU12]. The upper bound is derived by constructing an alternating Turing machine based on the *path lemma* [DMRU12, Lemma 11]. Accordingly, we formulate a path lemma for $\mathsf{BCL}_k(\mathbb{T}_\mathbb{C})$. Let $\mathrm{atoms}(\tau)$ be the set of constants, variables and constructor names occurring in τ. Additionally, for a substitution S let $\mathrm{atoms}(S) = \bigcup \{\mathrm{atoms}(S(\alpha)) \mid \alpha \in dom(S)\}$, and for a repository Δ let $\mathrm{atoms}(\Delta) = \bigcup \{\mathrm{atoms}(\tau) \mid C : \tau \in \Delta\}$.

Lemma 4.11 (Path Lemma for $\mathsf{BCL}_k(\mathbb{T}_\mathbb{C})$). *The following are equivalent conditions:*

(1) $\Delta \vdash_k C\, E_1 \ldots E_m : \tau$
(2) *There exists a set of paths*
$P \subseteq \mathbb{P}_m(\bigcap \{S(\Delta(C)) \mid \mathrm{level}(S) \leq k, \mathrm{atoms}(S) \subseteq \mathrm{atoms}(\Delta) \cup \mathrm{atoms}(\tau)\})$ *such that*
   (a) $\bigcap_{\pi \in P} \mathrm{tgt}_m(\pi) \leq \tau$
   (b) $\Delta \vdash_k E_i : \bigcap_{\pi \in P} \arg_i(\pi)$ *for $1 \leq i \leq m$*

The detailed proof of Lemma 4.11, which is a slight extension of [DMRU12, Lemma 11], is in the appendix.

In the above Lemma 4.11 we can bound the size of the set P of paths of level at most k. Let $\exp_k$ be the iterated exponential function, i.e. $\exp_0(n) = n$ and $\exp_{k+1}(n) = 2^{\exp_k(n)}$.

Lemma 4.12. *There exists a polynomial p such that (modulo $=$) for any $k \in \mathbb{N}$ the number of level-k types over n atoms is at most $\exp_{k+1}(p(n))$, and the size of such types is at most $\exp_k(p(n))$.*

Proof. Since normalizing (i.e. recursively organizing) a type does not increase its level, we only need to consider normalized types. Fix the set of atoms $\mathbb{A}$ with $|\mathbb{A}| = n$. Let $\mathbb{P}^k$ denote the set of paths of level at most k, and $\mathbb{T}^k$ the set of normalized types of level at most k. We have

$$|\mathbb{A}| = n \le n|\mathbb{T}^k|^2 \text{ and}$$

$$|\{\sigma \to \pi \mid \sigma \in \mathbb{T}^k, \pi \in \mathbb{P}^k\}| \le |\mathbb{T}^k| \cdot |\mathbb{P}^k| \le n|\mathbb{T}^k|^2 \text{ and}$$

$$|\{c(\pi) \mid c \in \mathbb{A}, \pi \in \mathbb{P}^k\}| \le |\mathbb{A}| \cdot |\mathbb{P}^k| \text{ therefore}$$

$$|\mathbb{P}^{k+1}| \le |\mathbb{A}| + |\{\sigma \to \pi \mid \sigma \in \mathbb{T}^k, \pi \in \mathbb{P}^k\}| + |\{c(\pi) \mid c \in \mathbb{A}, \pi \in \mathbb{P}^k\}| \le 3n|\mathbb{T}^k|^2 \text{ and}$$

$$|\mathbb{T}^{k+1}| \le |\{\bigcap P \mid P \subseteq \mathbb{P}^{k+1}\}| \le 2^{3n|\mathbb{T}^k|^2}$$

By induction on k we have that there exists a polynomial p such that

$$|\mathbb{T}^k| \le \exp_{k+1}(p(n) + \sum_{i=0}^{k-1} \frac{p(n)}{2^i}) \le \exp_{k+1}(3p(n))$$

Let s_k be the maximal size of a type of level k with atoms in $\mathbb{A}$. We have

$$s_{k+1} \le |\mathbb{P}^{k+1}| \cdot (2s_k + 2) \le 3n|\mathbb{T}^k|^2 \cdot (2s_k + 2)$$

By induction on k we have that there exists a polynomial p such that $s_k \le \exp_k(p(n))$. $\square$

For the interested reader, only the rank of a given type influences the height of the exponentiation tower.

Using Lemma 4.11 we can decide $\mathsf{BCL}_k(\mathbb{T}_{\mathbb{C}})$ inhabitation by the alternating Turing machine shown in Figure 1.

Figure 1: Alternating Turing machine deciding inhabitation in $\mathsf{BCL}_k(\mathbb{T}_{\mathbb{C}})$

```
      Input :   Δ, τ, k
      loop :
  1     CHOOSE (C : σ) ∈ Δ
  2     σ' := ⋂{S(σ) | level(S) ≤ k, atoms(S) ⊆ atoms(Δ) ∪ atoms(τ)}
  3     CHOOSE m ∈ {0, . . . , maximal arity of paths in σ'}
  4     CHOOSE P ⊆ ℙ_m(σ')

  5     IF (⋂_{π∈P} tgt_m(π) ≤ τ) THEN
  6       IF (m = 0) THEN ACCEPT
  7     ELSE
  8       FORALL(i = 1 . . . m)
  9         τ := ⋂_{π∈P} arg_i(π)
  10        GOTO loop
```

Theorem 4.13. $\mathsf{BCL}_k(\mathbb{T}_{\mathbb{C}})$ *inhabitation in* $(k+2)$-ExpTime.

Proof. The alternating Turing machine in Figure 1 directly implements Lemma 4.11 and is therefore sound and complete. To show that the machine operates in alternating $(k+1)$-ExpSpace, we need to bound the size of (organized) σ', which is $(n \cdot \exp_{k+1}(p(n)) \cdot$

$\exp_k(p(n)))^2$ due to Lemmas 4.12 and 4.9. By the identity $\text{ASPACE}(f(n)) = \text{DTIME}(2^{\mathcal{O}(f(n))})$ [CKS81], $\mathsf{BCL}_k(\mathbb{T}_\mathbb{C})$ inhabitation is in $(k+2)$-EXPTIME. $\qquad\square$

5. MIXIN COMPOSITION SYNTHESIS BY TYPE INHABITATION

In this section, we present an encoding of record types by $\mathbb{T}_\mathbb{C}$ types that capture mixin semantics. We use the encoding to define a repository of typed combinators from classes and mixins that can be used to synthesize meaningful mixin compositions by means of $\mathsf{BCL}_k(\mathbb{T}_\mathbb{C})$ inhabitation. In the following we fix a finite set of labels $\mathcal{L} \subseteq \mathbf{Label}$ that are used in the particular domain of interest for mixin composition synthesis.

5.1. **Records as Unary Covariant Distributing Constructors.** First, we need to encode record types as $\mathbb{T}_\mathbb{C}$ types. We define constructors $\langle\!\langle \cdot \rangle\!\rangle$ and $l(\cdot)$ for $l \in \mathcal{L}$ to represent record types using the following partial translation function $[\![\cdot]\!] : \mathbb{T} \to \mathbb{T}_\mathbb{C}$ as follows:

$$[\![\omega]\!] = \omega \qquad\qquad\qquad [\![a]\!] = a$$
$$[\![\sigma \to \tau]\!] = [\![\sigma]\!] \to [\![\tau]\!] \qquad\qquad [\![\sigma \cap \tau]\!] = [\![\sigma]\!] \cap [\![\tau]\!]$$
$$[\![\langle l : \tau \rangle]\!] = \langle\!\langle l([\![\tau]\!]) \rangle\!\rangle \qquad\qquad [\![\langle \rangle]\!] = \langle\!\langle \omega \rangle\!\rangle$$

By definition, we have $[\![\langle l_i : \tau_i \mid i \in I \rangle]\!] = [\![\bigcap_{i \in I} \langle l_i : \tau_i \rangle]\!] = \bigcap_{i \in I} \langle\!\langle l_i([\![\tau_i]\!]) \rangle\!\rangle$ if $I \neq \varnothing$.

Lemma 5.1. *For any $\sigma, \tau \in \mathbb{T}$ such that $[\![\sigma]\!]$ and $[\![\tau]\!]$ are defined we have $\sigma = \tau$ iff $[\![\sigma]\!] = [\![\tau]\!]$.*

Proof. Routine induction on $\leq$ derivation observing that σ, τ do not contain $+$, $[\![\cdot]\!]$ is homomorphic wrt. $\to$ and $\cap$, and atomic records are covariant and distribute over $\cap$. $\qquad\square$

The translation function $[\![\cdot]\!]$ is not defined for types containing $+$. Since $+$ has non-monotonic properties, it cannot be immediately represented by a covariant type constructor. Simply applying Lemma 3.7.4 would require to consider all arguments a mixin could possibly be applied to. Such an unwieldy specification would not be adequate for synthesis. There are two possibilities to deal with this problem. The first option is extending the type-system used for inhabitation. Here, the main difficulty is that existing inhabitation algorithms rely on the separation of intersections into paths [DMRU12]. As demonstrated in the remark accompanying Lemma 3.9, it becomes unclear how to perform such a separation in the presence of the non-monotonic $+$ operation. The second option, pursued in the rest of this section, is to use the expressiveness of schematism provided by $\mathsf{BCL}_k(\mathbb{T}_\mathbb{C})$. Specifically, encoding particular $\mathbb{T}$ types containing $+$ that capture mixin semantics as $\mathbb{T}_\mathbb{C}$ types suited for $\mathsf{BCL}_k(\mathbb{T}_\mathbb{C})$ inhabitation. Ultimately, we are able to achieve completeness (cf. Theorem 5.5) wrt. particular mixin typings (cf. Property $(\star)$) which restrict the general shape (3.5). This restriction allows for a concise schematic mixin specification suited for synthesis by $\mathsf{BCL}_k(\mathbb{T}_\mathbb{C})$ inhabitation.

Let $M \equiv \lambda\,\mathsf{argClass}.\,\mathbf{Y}(\lambda\,\mathsf{myClass}\,\lambda\,\mathsf{state}.\,(\mathsf{argClass}\,\mathsf{state}) \oplus R) \in \mathcal{M}$ be such that for some σ, ρ_1, ρ_2, which do not contain $+$, with $lbl(\rho_2) = lbl(R)$ and all $\rho \in \mathbb{T}_{\langle\rangle}$ we have

$$\vdash M : (\sigma \to \rho \cap \rho_1) \to (\sigma \to \rho + \rho_2) \qquad\qquad (\star)$$

We define the $\mathbb{T}_\mathbb{C}$ type

$$\tau_M \equiv \big(([\![\sigma]\!] \to [\![\rho_1]\!]) \to ([\![\sigma]\!] \to [\![\rho_2]\!])\big) \cap \bigcap_{l \in \mathcal{L} \smallsetminus lbl(R)} \big(([\![\sigma]\!] \to \langle\!\langle l(\alpha_l) \rangle\!\rangle) \to ([\![\sigma]\!] \to \langle\!\langle l(\alpha_l) \rangle\!\rangle)\big)$$

Note that τ_M contains only labels $l \in \mathcal{L}$ and type variables α_l where $l \in \mathcal{L}$ because σ, ρ_1, ρ_2 do not contain type variables (cf. Definition 3.3).

Lemma 5.2 (Translation Soundness). *Assume* $(\star)$.
For any substitution S and type $\tau \in \mathbb{T}$ such that $[\![\tau]\!] = S(\tau_M)$ we have $\vdash M : \tau$.

Proof. We necessarily have
$S(\tau_M) = [\![((\sigma \to \rho_1) \to (\sigma \to \rho_2)) \cap \bigcap_{l \in \mathcal{L} \smallsetminus lbl(R)} ((\sigma \to \langle l : \tau_l \rangle) \to (\sigma \to \langle l : \tau_l \rangle))]\!]$ for some τ_l for
$l \in \mathcal{L} \smallsetminus lbl(R)$. Due to $(\star)$ with $\rho = \langle \rangle$ we have $\vdash M : (\sigma \to \rho_1) \to (\sigma \to \rho_2)$, and by Lemma 3.19 we have $\vdash M : (\sigma \to \langle l : \tau_l \rangle) \to (\sigma \to \langle l : \tau_l \rangle)$ for any $l \in \mathcal{L} \smallsetminus lbl(R)$, thus showing the claim. $\square$

Translation Soundness forces instantiations of τ_M to remain within typings of M in Λ_R. *Negative information*, i.e. information about labels absent in R, is encoded by explicitly capturing all positive information, excluding labels in $lbl(R)$, in instances of

$$\bigcap_{l \in \mathcal{L} \smallsetminus lbl(R)} (([\![\sigma]\!] \to \langle\!\langle l(\alpha_l) \rangle\!\rangle) \to ([\![\sigma]\!] \to \langle\!\langle l(\alpha_l) \rangle\!\rangle))$$

Schematism, i.e. the possibility to instantiate type variables α_l, is essential to capture mixin behavior. This encoding is possible under the assumption of a finite set of labels $\mathcal{L}$, which is valid because synthesis does not introduce new labels. The encoding overhead is polynomially bounded by product of the number of mixin combinators and the number of labels.

Lemma 5.3 (Translation Completeness). *Assume* $(\star)$.
For any $\rho \in \mathbb{T}_{\langle\rangle}$ with $lbl(\rho) \subseteq \mathcal{L}$ there exists a substitution S and a type $\tau \in \mathbb{T}$ such that $S(\tau_M) \leq [\![\tau]\!]$ and $\tau = (\sigma \to \rho \cap \rho_1) \to (\sigma \to \rho + \rho_2)$.

Proof. Let $\rho = \bigcap_{l \in L} \langle l : \tau_l \rangle$ for some $L \subseteq \mathcal{L}$. By Lemma 3.7.4 we may assume for each $l \in L$ that τ_l does not contain +. We define

$$S(\alpha_l) = \begin{cases} [\![\tau_l]\!] & \text{if } l \in L \smallsetminus lbl(R) \\ \omega & \text{else} \end{cases}$$

and successively applying Lemma 5.1 obtain

$$S(\tau_M) \leq [\![((\sigma \to \rho_1) \to (\sigma \to \rho_2)) \cap \bigcap_{l \in L \smallsetminus lbl(R)} ((\sigma \to \langle l : \tau_l \rangle) \to (\sigma \to \langle l : \tau_l \rangle))]\!]$$

$$\leq [\![((\sigma \to \bigcap_{l \in L \smallsetminus lbl(R)} \langle l : \tau_l \rangle \cap \rho_1) \to (\sigma \to \bigcap_{l \in L \smallsetminus lbl(R)} \langle l : \tau_l \rangle \cap \rho_2))]\!]$$

$$\leq [\![((\sigma \to \rho \cap \rho_1) \to (\sigma \to \underbrace{\bigcap_{l \in L \smallsetminus lbl(R)} \langle l : \tau_l \rangle \cap \rho_2}_{= \rho + \rho_2 \text{ by Lem. 3.10.2}}))]\!] \qquad\qquad \square$$

Translation Completeness ensures that each type Λ_R of M according to $(\star)$ is captured by some instance of τ_M.

Using the above translation properties, we construct a repository of typed combinators representing classes and mixins.

5.2. **Mixin Composition.** In this section we denote type assignment in Λ_R by $\vdash_{\langle\rangle}$ and fix the following ingredients:

- A finite set of classes $\mathcal{C}$.
- For each $C \in \mathcal{C}$ types $\sigma_C \in \mathbb{T}, \rho_C \in \mathbb{T}_{\langle\rangle}$ such that $[\![\sigma_C \to \rho_C]\!]$ is defined and
  $\vdash_{\langle\rangle} C : \sigma_C \to \rho_C$.
- A finite set of mixins $\mathcal{M}$.
- For each $M \in \mathcal{M}$ types $\sigma_M \in \mathbb{T}$ and $\rho_M^1, \rho_M^2 \in \mathbb{T}_{\langle\rangle}$ such that $[\![\sigma_M]\!], [\![\rho_M^1]\!], [\![\rho_M^2]\!]$ are defined and for all types $\rho \in \mathbb{T}_{\langle\rangle}$ we have $\vdash_{\langle\rangle} M : (\sigma_M \to \rho \cap \rho_M^1) \to (\sigma_M \to \rho + \rho_M^2)$.
- For each $M \in \mathcal{M}$ the non-empty set of labels $L_M = lbl(\rho_M^2) \subseteq \mathcal{L}$ defined by M.

We translate given classes and mixins to the following repository $\Delta_{\mathcal{L}}^{\mathcal{C},\mathcal{M}}$ of combinators

$$
\begin{aligned}
\Delta_{\mathcal{L}}^{\mathcal{C},\mathcal{M}} = &\{C : [\![\sigma_C \to \rho_C]\!] \mid C \in \mathcal{C}\} \\
&\cup \{M : \big(([\![\sigma_M]\!] \to [\![\rho_M^1]\!]) \to ([\![\sigma_M]\!] \to [\![\rho_M^2]\!])\big) \\
&\quad \cap \bigcap_{l \in \mathcal{L} \smallsetminus L_M} \big(([\![\sigma_M]\!] \to \langle\!\langle l(\alpha_l)\rangle\!\rangle) \to ([\![\sigma_M]\!] \to \langle\!\langle l(\alpha_l)\rangle\!\rangle)\big) \mid M \in \mathcal{M}\}
\end{aligned}
$$

Note that we use identifiers C for classes and M for mixins just as symbolic names in the repository, while they are also typable closed terms in Λ_R.

To simplify notation, we introduce the infix metaoperator $\gg$ such that $x \gg f = f\ x$. It is left associative and has the lowest precedence. Accordingly, $x \gg f \gg g = g\ (f\ x)$.

Although types in $\Delta_{\mathcal{L}}^{\mathcal{C},\mathcal{M}}$ do not contain record-merge, types of mixin compositions in $\mathsf{BCL}_k(\mathbb{T}_\mathbb{C})$ are sound, which is shown in the following Theorem 5.4.

Theorem 5.4 (Soundness)**.** *Let $M_1, \ldots, M_n \in \mathcal{M}$ be mixins, let $C \in \mathcal{C}$ be a class, let $\sigma \in \mathbb{T}, \rho \in \mathbb{T}_{\langle\rangle}$ be types such that $[\![\sigma \to \rho]\!]$ is defined, and let $k \in \mathbb{N}$.*
If $\Delta_{\mathcal{L}}^{\mathcal{C},\mathcal{M}} \vdash_k C \gg M_1 \gg \ldots \gg M_n : [\![\sigma \to \rho]\!]$, then $\vdash_{\langle\rangle} C \gg M_1 \gg M_2 \gg \ldots \gg M_n : \sigma \to \rho$.

Proof. Induction on n using Lemma 5.2 and $(\to E)$. $\qquad\square$

Complementarily, types of mixin compositions in $\mathsf{BCL}_k(\mathbb{T}_\mathbb{C})$ are complete wrt. Λ_R type assumptions for classes and mixins listed above. Specifically, we show in the following Theorem 5.5 that, assuming arguably natural typings of classes and mixins, we can find corresponding $\mathsf{BCL}_k(\mathbb{T}_\mathbb{C})$-counterparts of Λ_R type derivations.

Theorem 5.5 (Partial Completeness)**.** *Let $\Gamma \subseteq \{x_C : \sigma_C \to \rho_C \mid C \in \mathcal{C}\} \cup \{x_M^\rho : (\sigma_M \to \rho \cap \rho_M^1) \to (\sigma_M \to \rho + \rho_M^2) \mid M \in \mathcal{M}, \rho \in \mathbb{T}_{\langle\rangle}, [\![\rho]\!]$ is defined$\}$ be a finite context and let $\sigma \in \mathbb{T}, \rho \in \mathbb{T}_{\langle\rangle}$ be types such that $[\![\sigma \to \rho]\!]$ is defined. Let $k = \max(\{\text{level}([\![\rho_C]\!]) \mid C \in \mathcal{C}\} \cup \{\text{level}([\![\rho_M^2]\!]) \mid M \in \mathcal{M}\} \cup \{\text{level}([\![\rho]\!])\})$. Let $M_1, \ldots, M_n \in \mathcal{M}$ be mixins.*
If $\Gamma \vdash_{\langle\rangle} x_C \gg x_{M_1}^{\rho_1} \gg \ldots \gg x_{M_n}^{\rho_n} : \sigma \to \rho$, then $\Delta_{\mathcal{L}}^{\mathcal{C},\mathcal{M}} \vdash_k C \gg M_1 \gg \ldots \gg M_n : [\![\sigma \to \rho]\!]$.

Proof. Induction on n choosing for each x_M^ρ where $\rho = \bigcap_{l \in L} \langle l : \tau_l \rangle$ the substitution $\alpha_l \mapsto [\![\tau_l]\!]$ for $l \in L \smallsetminus L_M$ and $\alpha_l \mapsto \omega$ otherwise to type $M \in \Delta_{\mathcal{L}}^{\mathcal{C},\mathcal{M}}$ and using $(\to E), (\leq)$, Lemma 5.3. $\square$

Note that Theorem 5.5 defines a bound k based on the input. In the following, we use this bound ensuring that the provided examples are in fact computable. We extend the running

example by the following mixin Succ2.

$\text{Succ2} = \lambda\text{argClass}.\mathbf{Y}(\lambda\text{myClass}.\lambda\text{state}.$

    $\text{let super} = \text{argClass state in}$

    $\text{let self} = \text{myClass state in}$

        $\text{super} \oplus \langle succ2 = \text{let super}' = \text{argClass}(\text{super}.succ) \text{ in super}'.succ\rangle)$

Succ2 adds the method *succ2* that is the twofold application of *succ*. Note that the object **super** is updated functionally in *succ2*. Using the above translation, we obtain

$\Delta^{\{\text{Num}\},\{\text{Succ2},\text{Comparable}\}}_{\{get,set,succ,succ2,compare\}} = \{$

  $\text{Num} : \text{Int} \to \langle\!\langle get(\text{Int}) \cap set(\text{Int} \to \text{Int}) \cap succ(\text{Int})\rangle\!\rangle,$

  $\text{Comparable} : \big((\text{Int} \to \langle\!\langle get(\text{Int})\rangle\!\rangle) \to (\text{Int} \to \langle\!\langle compare(\langle\!\langle get(\text{Int})\rangle\!\rangle \to \text{Bool})\rangle\!\rangle)\big)$

    $\cap \big((\text{Int} \to \langle\!\langle get(\alpha_{get})\rangle\!\rangle) \to (\text{Int} \to \langle\!\langle get(\alpha_{get})\rangle\!\rangle)\big)$

    $\cap \big((\text{Int} \to \langle\!\langle set(\alpha_{set})\rangle\!\rangle) \to (\text{Int} \to \langle\!\langle set(\alpha_{set})\rangle\!\rangle)\big)$

    $\cap \big((\text{Int} \to \langle\!\langle succ(\alpha_{succ})\rangle\!\rangle) \to (\text{Int} \to \langle\!\langle succ(\alpha_{succ})\rangle\!\rangle)\big)$

    $\cap \big((\text{Int} \to \langle\!\langle succ2(\alpha_{succ2})\rangle\!\rangle) \to (\text{Int} \to \langle\!\langle succ2(\alpha_{succ2})\rangle\!\rangle)\big),$

  $\text{Succ2} : \big((\text{Int} \to \langle\!\langle succ(\text{Int})\rangle\!\rangle) \to (\text{Int} \to \langle\!\langle succ2(\text{Int})\rangle\!\rangle)\big)$

    $\cap \big((\text{Int} \to \langle\!\langle get(\alpha_{get})\rangle\!\rangle) \to (\text{Int} \to \langle\!\langle get(\alpha_{get})\rangle\!\rangle)\big)$

    $\cap \big((\text{Int} \to \langle\!\langle set(\alpha_{set})\rangle\!\rangle) \to (\text{Int} \to \langle\!\langle set(\alpha_{set})\rangle\!\rangle)\big)$

    $\cap \big((\text{Int} \to \langle\!\langle succ(\alpha_{succ})\rangle\!\rangle) \to (\text{Int} \to \langle\!\langle succ(\alpha_{succ})\rangle\!\rangle)\big)$

    $\cap \big((\text{Int} \to \langle\!\langle compare(\alpha_{compare})\rangle\!\rangle) \to (\text{Int} \to \langle\!\langle compare(\alpha_{compare})\rangle\!\rangle)\big)\}$

We may ask inhabitation questions such as

$\Delta^{\{\text{Num}\},\{\text{Succ2},\text{Comparable}\}}_{\{get,set,succ,succ2,compare\}} \vdash_k ? : [\![\text{Int} \to \langle succ : \text{Int}, compare : \langle get : \text{Int}\rangle \to \text{Bool}, succ2 : \text{Int}\rangle]\!]$

and obtain the combinatory term "Num $\gg$ Comparable $\gg$ Succ2" as the synthesized result. From Theorem 5.4 we know

$\vdash_{\langle\rangle} \text{Num} \gg \text{Comparable} \gg \text{Succ2} : \text{Int} \to \langle succ : \text{Int}, compare : \langle get : \text{Int}\rangle \to \text{Bool}, succ2 : \text{Int}\rangle$

    The presented encoding has several benefits with respect to scalability. First, the size of the presented repositories is polynomial in $|\mathcal{L}| * |\mathcal{C}| * |\mathcal{M}|$. Second, expanding the label set $\mathcal{L}$ can be performed automatically in polynomial time by adding additional components $([\![\sigma_M]\!] \to \langle\!\langle l(\alpha_l)\rangle\!\rangle) \to ([\![\sigma_M]\!] \to \langle\!\langle l(\alpha_l)\rangle\!\rangle)$ to each mixin M for each new label l. Third, adding a class/mixin to an existing repository is as simple as adding one typed combinator for the class/mixin. Existing combinators in the repository remain untouched. As an example, we add the following mixin SuccDelta to $\Delta^{\{\text{Num}\},\{\text{Succ2},\text{Comparable}\}}_{\{get,set,succ,succ2,compare\}}$.

  $\text{SuccDelta} = \lambda\text{argClass}.\mathbf{Y}(\lambda\text{myClass}.\lambda\text{state}.$

    $\text{let super} = \text{argClass state in}$

    $\text{let self} = \text{myClass state in super} \oplus \langle succ = \lambda d.(\text{super}.set(\text{super}.get + d))\rangle)$

  In Λ_R for all types $\rho \in \mathbb{T}_{\langle\rangle}$ we have

  $\vdash_{\langle\rangle} \text{SuccDelta} : (\text{Int} \to \rho \cap \langle get : \text{Int}, set : \text{Int} \to \text{Int}\rangle) \to (\text{Int} \to \rho + \langle succ : \text{Int} \to \text{Int}\rangle)$

We obtain the following extended repository

$$\Delta_{\{get,set,succ,succ2,compare\}}^{\{Num\},\{Succ2,Comparable,SuccDelta\}} = \Delta_{\{get,set,succ,succ2,compare\}}^{\{Num\},\{Succ2,Comparable\}}$$

$$\cup \{SuccDelta : \big((Int \to \langle\!\langle get(Int) \cap set(Int \to Int)\rangle\!\rangle) \to (Int \to \langle\!\langle succ(Int \to Int)\rangle\!\rangle)\big)$$

$$\cap \big((Int \to \langle\!\langle get(\alpha_{get})\rangle\!\rangle) \to (Int \to \langle\!\langle get(\alpha_{get})\rangle\!\rangle)\big)$$

$$\cap \big((Int \to \langle\!\langle set(\alpha_{set})\rangle\!\rangle) \to (Int \to \langle\!\langle set(\alpha_{set})\rangle\!\rangle)\big)$$

$$\cap \big((Int \to \langle\!\langle succ2(\alpha_{succ2})\rangle\!\rangle) \to (Int \to \langle\!\langle succ2(\alpha_{succ2})\rangle\!\rangle)\big)$$

$$\cap \big((Int \to \langle\!\langle compare(\alpha_{compare})\rangle\!\rangle) \to (Int \to \langle\!\langle compare(\alpha_{compare})\rangle\!\rangle)\big)\}$$

Asking the inhabitation question

$$\Delta_{\{get,set,succ,succ2,compare\}}^{\{Num\},\{Succ2,Comparable,SuccDelta\}} \vdash_k ? : [\![Int \to \langle succ : Int \to Int, succ2 : Int\rangle]\!]$$

synthesizes "Num $\gg$ Succ2 $\gg$ SuccDelta". Note that even in such a simplistic scenario the order in which mixins are applied can be crucial mainly because $\oplus$ is not commutative. Moreover, the early binding of self and the associated preservation of overwritten methods allows for destructive overwrite of *succ* by SuccDelta without invalidating *succ2* that is previously added by Succ2. This also may make multiple applications of a single mixin meaningful.

In order to improve the reading experience, the running example is coherently arranged in the appendix.

5.3. **Semantic Types.** An additional benefit of using intersection types and, in particular, the (CL)S framework is the availability of the so called *semantic types* [Reh13]. Semantic types can be used to further specify the semantics of typed combinators in a repository and restrict/guide the inhabitant search. Semantic types consist of additional constants a and constructors c($\cdot$) that arise in the current domain of interest (semantic types are well suited to capture taxonomies [Reh13]). Native types are augmented by semantic types using intersection. For example, consider the native type Int. We may be interested in whether the current value is Even or Odd. Therefore, we may augment the native type by this information resulting in types of the form Int $\cap$ Even representing even integers or Int $\cap$ Odd representing odd integers. This becomes increasingly interesting when we have knowledge about functional dependencies between semantic types in the domain of interest. In particular, knowing that successors of even integers are odd and vice versa, we may augment the native type of Num to represent this domain knowledge

$$Num : (Int \cap Even \to \langle\!\langle get(Int \cap Even)\rangle\!\rangle)$$

$$\cap (Int \cap Odd \to \langle\!\langle get(Int \cap Odd)\rangle\!\rangle)$$

$$\cap (Int \to \langle\!\langle set((Int \to Int) \cap (Even \to Even) \cap (Odd \to Odd))\rangle\!\rangle)$$

$$\cap (Int \cap Even \to \langle\!\langle succ(Int \cap Odd)\rangle\!\rangle)$$

$$\cap (Int \cap Odd \to \langle\!\langle succ(Int \cap Even)\rangle\!\rangle)$$

The above type of Num expresses that the *get* method returns the semantic type of the underlying state of the object, the *set* method does not take the underlying state of the object into account, and the *succ* method returns the opposed semantic type. Note that, if all semantic components are erased, the remaining type of Num is exactly the original native type.

In the setting of mixin composition synthesis, semantic types enhance the expressiveness of the type system. The mixin Succ2 from our running example can be typed in the following way

$$\text{Succ2} : (\text{Int} \cap \textsf{Even} \to \langle\!\langle succ(\text{Int} \cap \textsf{Odd})\rangle\!\rangle) \cap (\text{Int} \cap \textsf{Odd} \to \langle\!\langle succ(\text{Int} \cap \textsf{Even})\rangle\!\rangle)$$
$$\to (\text{Int} \cap \textsf{Even} \to \langle\!\langle succ2(\text{Int} \cap \textsf{Even})\rangle\!\rangle) \cap (\text{Int} \cap \textsf{Odd} \to \langle\!\langle succ2(\text{Int} \cap \textsf{Odd})\rangle\!\rangle)$$

The above type of Succ2 expresses that given the proper semantic types of $succ$, the semantic type of $succ2$ corresponds to the semantic type of the underlying state of the object, i.e. the twofold successor of an even (resp. odd) integer is even (resp. odd).

This allows to distinguish methods with different semantics in the domain of interest but exhibiting identical native types. Consider the following mixin Parity that overwrites the method $succ$ to be the twofold successor such that the parity of the underlying state remains the same.

$$\text{Parity} = \lambda\textsf{argClass}.\mathbf{Y}(\lambda\textsf{myClass}.\lambda\textsf{state}.$$
$$\texttt{let } \textsf{super} = \textsf{argClass state } \texttt{in}$$
$$\texttt{let } \textsf{self} = \textsf{myClass state } \texttt{in } \textsf{super} \oplus \langle succ = \textsf{super}.succ2\rangle)$$

Parity can be typed in the following way

$$\text{Parity} : (\text{Int} \cap \textsf{Even} \to \langle\!\langle succ2(\text{Int} \cap \textsf{Even})\rangle\!\rangle) \cap (\text{Int} \cap \textsf{Odd} \to \langle\!\langle succ2(\text{Int} \cap \textsf{Odd})\rangle\!\rangle)$$
$$\to (\text{Int} \cap \textsf{Even} \to \langle\!\langle succ(\text{Int} \cap \textsf{Even})\rangle\!\rangle) \cap (\text{Int} \cap \textsf{Odd} \to \langle\!\langle succ(\text{Int} \cap \textsf{Odd})\rangle\!\rangle)$$

Asking the inhabitation question

$$\Delta_{\{get,set,succ,succ2,compare\}}^{\{\text{Num}\},\{\text{Succ2,Comparable,SuccDelta,Parity}\}} \vdash_k \text{Int} \cap \textsf{Even} \to \langle\!\langle succ : \text{Int} \cap \textsf{Even}\rangle\!\rangle$$

results in "Num ≫ Succ2 ≫ Parity". Note that due to the early binding, $succ2$ uses the old $succ$ method which has the required semantics in order for Succ2 to be applied.

Let us explore the descriptive capabilities of semantic types using a more illustrative example. Consider the domain of cryptography containing algorithms for encrypting and signing data. Describing abstract properties such as "encrypted" or "signed" at the lowest level, e.g. using Hoare logic, requires an enormous amount of work. In practice, such properties are described textually while the native type, e.g. String, does not capture particular semantics. Consider the following repository

$$\Delta_{\text{native}} = \{\text{Reader} : \quad \text{String} \to \langle\!\langle get(\text{String})\rangle\!\rangle,$$
$$\text{Enc} : \quad (\text{String} \to \langle\!\langle get(\text{String})\rangle\!\rangle) \to \text{String} \to \langle\!\langle get(\text{String})\rangle\!\rangle,$$
$$\text{Sign} : \quad (\text{String} \to \langle\!\langle get(\text{String})\rangle\!\rangle) \to \text{String} \to \langle\!\langle get(\text{String})\rangle\!\rangle,$$
$$\text{Time} : \quad (\text{String} \to \langle\!\langle get(\text{String})\rangle\!\rangle) \to \text{String} \to \langle\!\langle get(\text{String})\rangle\!\rangle\}$$

with the following textual description

- The Reader class provides a get method that returns some plain text data.
- The Enc mixin replaces the get method of a given class with a new get method that returns the encrypted result of the overwritten get method.
- The Sign mixin replaces the get method appending the signature to the result of the overwritten get method.
- The Time mixin replaces the get method appending a time-stamp to the result of the overwritten get method.

Unfortunately, the types in Δ are too general to be used for synthesis of meaningful compositions. However, we can use semantic types to embed the textual description of the particular semantics in our domain of interest into Δ_{native} resulting in the following repository

$$\Delta = \{\text{Reader}: \quad \text{String} \to \langle\!\langle get(\text{String} \cap \mathsf{Plain})\rangle\!\rangle,$$
$$\text{Enc}: \quad (\text{String} \to \langle\!\langle get(\text{String} \cap \alpha)\rangle\!\rangle) \to (\text{String} \to \langle\!\langle get(\text{String} \cap \mathsf{Enc}(\alpha))\rangle\!\rangle),$$
$$\text{Sign}: \quad (\text{String} \to \langle\!\langle get(\text{String} \cap \alpha)\rangle\!\rangle) \to (\text{String} \to \langle\!\langle get(\text{String} \cap \alpha \cap \mathsf{Sign}(\alpha))\rangle\!\rangle),$$
$$\text{Time}: \quad (\text{String} \to \langle\!\langle get(\text{String} \cap \alpha)\rangle\!\rangle) \to (\text{String} \to \langle\!\langle get(\text{String} \cap \alpha \cap \mathsf{Time})\rangle\!\rangle)\}$$

In the above repository Δ, the mixin Enc replaces any semantic information α of the *get* method by $\mathsf{Enc}(\alpha)$. The mixin Sign adds semantic information $\mathsf{Sign}(\alpha)$ to any previous semantic information α while also preserving α. The mixin Time adds semantic information Time while preserving the old semantic information.

If we are interested in a composition that has a *get* method returning an encrypted plain text with a time-stamp and a signature, we may ask the following inhabitation question

$$\Delta \vdash_k ? : \text{String} \to \langle\!\langle get(\text{String} \cap \mathsf{Enc}(\mathsf{Plain} \cap \mathsf{Time} \cap \mathsf{Sign}(\mathsf{Plain} \cap \mathsf{Time})))\rangle\!\rangle$$

The above question is answered by "Reader $\gg$ Time $\gg$ Sign $\gg$ Enc". If we are interested in a composition that encrypts the plain-text thrice, we may ask the following inhabitation question

$$\Delta \vdash_k ? : \text{String} \to \langle\!\langle get(\text{String} \cap \mathsf{Enc}(\mathsf{Enc}(\mathsf{Enc}(\mathsf{Plain}))))\rangle\!\rangle$$

The above question is answered by "Reader $\gg$ Enc $\gg$ Enc $\gg$ Enc".

6. Conclusion and Future Work

We presented a theory for automatic compositional construction of object-oriented classes by combinatory synthesis. This theory is based on the λ-calculus with records and record merge $\oplus$ typed by intersection types with records and +. It is capable of modeling classes as functions from states to records (i.e. objects), and mixins as functions from classes to classes. Mixins can be assigned meaningful types using + expressing their compositional character. However, non-monotonic properties of + are incompatible with the existing theory of $\mathsf{BCL}(\to, \cap)$ synthesis. Therefore, we designed a translation to repositories of combinators typed in $\mathsf{BCL}(\mathbb{T}_\mathbb{C})$. We have proven this translation to be sound (Theorem 5.4) and partially complete (Theorem 5.5). A notable feature is the encoding of negative information (the absence of labels). The original approach [BDD+15b] exploited the logic programming capabilities of inhabitation by adding sets of combinators serving as witnesses for the non-presence of labels. In this work we further refined the encoding by embedding the distinction between labels that are accessed, modified or untouched by a mixin into the combinator type directly. In Section 5 we also showed that this encoding scales wrt. extension of repositories.

Future work includes further studies on the possibilities to encode predicates exploiting patterns similar to the negative information encoding. Another direction of future work is to extend types of mixins and classes by semantic as well as modal types [DMR14], a development initiated in [BDDM14]. In particular, the expressiveness of semantic types can be used to assign meaning to multiple applications of a single mixin and allow to reason

about object-oriented code on a higher abstraction level as well as with higher semantic accuracy.

References

[AC96]   Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.

[AGO⁺16]   Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. The Essence of Dependent Object Types. In Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella, editors, *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, pages 249–272. Springer International Publishing, 2016.

[AMO12]   Nada Amin, Adriaan Moors, and Martin Odersky. Dependent Object Types. In *19th International Workshop on Foundations of Object-Oriented Languages*, 2012.

[ARO14]   Nada Amin, Tiark Rompf, and Martin Odersky. Foundations of Path-dependent Types. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 233–249, New York, NY, USA, 2014. ACM.

[Bak11]   Steffen van Bakel. Strict Intersection Types for the Lambda Calculus. *ACM Computing Survey*, 43(3):20:1–20:49, April 2011.

[Bar84]   Henk Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.

[BC90]   Gilad Bracha and William R. Cook. Mixin-based Inheritance. In *OOPSLA/ECOOP*, pages 303–311, 1990.

[BCDC83]   Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A Filter Lambda Model and the Completeness of Type Assignment. *Journal of Symbolic Logic*, 48(4):931–940, 1983.

[BDD⁺14]   Jan Bessai, Andrej Dudenhefner, Boris Düdder, Moritz Martens, and Jakob Rehof. Combinatory Logic Synthesizer. In Tiziana Margaria and Bernhard Steffen, editors, *ISoLA'14*, volume 8802 of *LNCS*, pages 26–40. Springer, 2014.

[BDD⁺15a] Jan Bessai, Boris Düdder, Andrej Dudenhefner, Tzu-Chun Chen, and Ugo de' Liguoro. Typing Classes and Mixins with Intersection Types. In *Proceedings Seventh Workshop on Intersection Types and Related Systems, ITRS 2014, Vienna, Austria, 18 July 2014.*, volume 177 of *EPTCS*, pages 79–93, 2015.

[BDD⁺15b] Jan Bessai, Andrej Dudenhefner, Boris Düdder, Tzu-Chun Chen, Ugo de' Liguoro, and Jakob Rehof. Mixin Composition Synthesis Based on Intersection Types. In Thorsten Altenkirch, editor, *13th International Conference on Typed Lambda Calculi and Applications, TLCA 2015, July 1-3, 2015, Warsaw, Poland*, volume 38 of *LIPIcs*, pages 76–91. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[BDDM14]   Jan Bessai, Boris Düdder, Andrej Dudenhefer, and Moritz Martens. Delegation-based Mixin Composition Synthesis, 2014. URL: `http://www-seal.cs.tu-dortmund.de/seal/downloads/papers/paper-ITRS2014.pdf`. Last accessed 2016-04-12.

[BDG07]   Viviana Bono, Ferruccio Damiani, and Elena Giachino. Separating Type, Behavior, and State to Achieve Very Fine-grained Reuse. *Electronic proceedings of FTfJP*, 7, 2007.

[BDG08]   Viviana Bono, Ferruccio Damiani, and Elena Giachino. On Traits and Types in a Java-like Setting. In *Fifth Ifip International Conference On Theoretical Computer Science–Tcs 2008*, pages 367–382. Springer, 2008.

[BDS13]   Henk Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types*. Perspectives in Logic, Cambridge University Press, 2013.

[Bok15]   Grigoriy V. Bokov. Undecidability of the problem of recognizing axiomatizations for propositional calculi with implication. *Logic Journal of IGPL*, 23(2):341–353, 2015.

[BPS99]   Viviana Bono, Amit Patel, and Vitaly Shmatikov. A Core Calculus of Classes and Mixins. In *ECOOP*, volume 1628 of *Lecture Notes in Computer Science*, pages 43–66, 1999.

[BR13]   Steffen van Bakel and Reuben Rowe. Functional Type Assignment for Featherweight Java. In *The Beauty of Functional Code*, pages 27–46. Springer, 2013.

[BR15]   Denis Bogdanas and Grigore Roşu. K-Java: A Complete Semantics of Java. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 445–456, New York, NY, USA, 2015. ACM.

[Bra92]   Gilad Bracha. *The Programming Language JIGSAW: Mixins, Modularity and Multiple Inheritance*. PhD thesis, University of Utah, 1992.

[Bru02]   Kim B. Bruce. *Foundations of Object-Oriented Languages - Types and Semantics*. MIT Press, 2002.

[BY79]   Choukri-Bey Ben-Yelles. *Type Assignment in the Lambda-Calculus: Syntax and Semantics*. PhD thesis, University College of Swansea, 1979.

[Can79]   Howard I. Cannon. Flavors: A Non-hierarchical Approach to Object-Oriented Programming, 1979. URL: `http://www.softwarepreservation.org/projects/LISP/MIT/nnnfla1-20040122.pdf`. Last accessed 2016-04-07.

[Car84]   Luca Cardelli. A Semantics of Multiple Inheritance. In *Semantics of Data Types*, volume 173, pages 51–67, 1984.

[CCH+89]   Peter S. Canning, William R. Cook, Walter L. Hill, Walter G. Olthoff, and John C. Mitchell. F-Bounded Polymorphism for Object-Oriented Programming. In *FPCA*, pages 273–280, 1989.

[CHC90]   William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance Is Not Subtyping. In *POPL'90*, pages 125–135. ACM Press, 1990.

[Chu57]   Alonzo Church. Application of Recursive Arithmetic to the Problem of Circuit Synthesis. *Summaries of the Summer Institute of Symbolic Logic*, I:3–50, 1957.

[CKS81]   Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer. Alternation. *Journal of the ACM (JACM)*, 28(1):114–133, 1981.

[CP96]   Adriana B. Compagnoni and Benjamin C. Pierce. Higher-Order Intersection Types and Multiple Inheritance. *Mathematical Structures in Computer Science*, 6(5):469–501, 1996.

[DCH92]   Mariangiola Dezani-Ciancaglini and Roger Hindley. Intersection Types for Combinatory Logic. *Theoretical Computer Science*, 100(2):303–324, 1992.

[DMR14]   Boris Düdder, Moritz Martens, and Jakob Rehof. Staged Composition Synthesis. In *ESOP*, volume 8410 of *Lecture Notes in Computer Science*, pages 67–86, 2014.

[DMRU12]   Boris Düdder, Moritz Martens, Jakob Rehof, and Paweł Urzyczyn. Bounded Combinatory Logic. In *Proceedings of CSL'12*, volume 16, pages 243–258. Schloss Dagstuhl, 2012.

[DN66]   Ole-Johan Dahl and Kristen Nygaard. SIMULA: An ALGOL-based Simulation Language. *Commun. ACM*, 9(9):671–678, September 1966.

[DS84]   L P. Deutsch and Allan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 297–302. ACM, 1984.

[Ecm11]   Standard Ecma. ECMA-262 ECMAScript Language Specification, 2011.

[EOC06]   Erik Ernst, Klaus Ostermann, and William R. Cook. A Virtual Class Calculus. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 270–282, New York, NY, USA, 2006. ACM.

[FOWZ16]   Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. Example-Directed Synthesis: A Type-Theoretic Interpretation. In *POPL'16*, pages 802–815. ACM, 2016.

[FP91]   Tim Freeman and Frank Pfenning. Refinement Types for ML. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, volume 26 of *PLDI '91*, pages 268–277, New York, NY, USA, June 1991. ACM.

[GKKP13]   Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete Completion using Types and Weights. In *SIGPLAN Notices*, volume 48, pages 27–38. ACM, 2013.

[HHSW02]   Christian Haack, Brian Howard, Allen Stoughton, and Joe B. Wells. Fully Automatic Adaptation of Software Components Based on Semantic Specifications. In *AMAST*, volume 2422 of *LNCS*, pages 83–98. Springer, 2002.

[Hin82]   Roger Hindley. The Simple Semantics for Coppo-Dezani-Sallé Types. In *International Symposium on Programming*, volume 137 of *LNCS*, pages 212–226. Springer, 1982.

[Hin08]   Roger Hindley. *Basic Simple Type Theory*. Cambridge Tracts in Theoretical Computer Science, vol. 42, 2008.

[HS08]   Roger Hindley and Jonathan P. Seldin. *Lambda-calculus and Combinators, an Introduction*. Cambridge University Press, 2008.

[HTT87]   John F. Horty, David S. Touretzky, and Richmond H. Thomason. A Clash of Intuitions: The Current State of Nonmonotonic Multiple Inheritance Systems. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 476–482, 1987.

[IPW01]   Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A Minimal Core
           Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.
[KL05]   Oleg Kiselyov and Ralf Lämmel. Haskell's Overlooked Object System. *CoRR*, abs/cs/0509027,
           2005.
[LC14]   Ugo de' Liguoro and Tzu-chun Chen. Semantic Types for Classes and Mixins, 2014. URL:
           `http://www.di.unito.it/~deligu/papers/UdLTC14.pdf`. Last accessed 2016-04-12.
[Lig01]   Ugo de' Liguoro. Characterizing Convergent Terms in Object Calculi via Intersection Types. In
           TLCA, pages 315–328, 2001.
[LP49]   Samuel Linial and Emil L. Post. Recursive Unsolvability of the Deducibility, Tarski's Completeness
           and Independence of Axioms Problems of Propositional Calculus. *Bulletin of the American
           Mathematical Society*, 55:50, 1949.
[LS08]   Luigi Liquori and Arnaud Spiwack. FeatherTrait: A Modest Extension of Featherweight Java.
           ACM Transactions on Programming Languages and Systems (TOPLAS), 30(2):11, 2008.
[LSZ12]   Giovanni Lagorio, Marco Servetto, and Elena Zucca. Featherweight Jigsaw - Replacing Inheritance
           by Composition in Java-like Languages. *Information and Computation*, 214:86–111, 2012.
[LV09]   Yoad Lustig and Moshe Y. Vardi. Synthesis from Component Libraries. In *FOSSACS*, volume
           5504 of *LNCS*, pages 395–409. Springer, 2009.
[Mit84]   John C. Mitchell. Coercion and Type Inference. In *Proceedings of the 11th ACM SIGACT-
           SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, pages 175–185, New
           York, NY, USA, 1984. ACM.
[Mit90]   John C. Mitchell. Toward a Typed Foundation for Method Specialization and Inheritance. In
           *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming
           Languages*, POPL '90, pages 109–124, New York, NY, USA, 1990. ACM.
[Mit96]   John C. Mitchell. *Foundations for Programming Languages.* Foundation of computing series.
           MIT Press, 1996.
[MNPS91]   Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform Proofs as a
           Foundation for Logic Programming. *Ann. Pure Appl. Logic*, 51(1–2):125–157, 1991.
[Moo86]   David A. Moon. Object-oriented Programming with Flavors. In *Conference Proceedings on
           Object-oriented Programming Systems, Languages and Applications*, OOPLSA '86, pages 1–8,
           New York, NY, USA, 1986. ACM.
[MW80]   Zohar Manna and Richard Waldinger. A Deductive Approach to Program Synthesis. *ACM
           Transactions on Programming Languages and Systems (TOPLAS)*, 2(1):90–121, 1980.
[OZ05]   Martin Odersky and Matthias Zenger. Scalable Component Abstractions. In *OOPSLA*, pages
           41–57. ACM, 2005.
[Pie91]   Benjamin C. Pierce. *Programming with Intersection Types and Bounded Quantification.* PhD
           thesis, PhD thesis, Carnegie Mellon University, 1991.
[PR89]   Amir Pnueli and Roni Rosner. On the Synthesis of a Reactive Module. In *Proceedings of the
           16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89,
           pages 179–190, New York, NY, USA, 1989. ACM.
[RB14]   Reuben Rowe and Steffen van Bakel. Semantic Types and Approximation for Featherweight Java.
           Theor. Comput. Sci., 517:34–74, 2014.
[Reh13]   Jakob Rehof. Towards Combinatory Logic Synthesis. In *BEAT'13, 1st International Workshop
           on Behavioural Types*. ACM, January 22 2013.
[Rém92]   Didier Rémy. Typing Record Concatenation for Free. In *POPL'92*, pages 166–176, 1992.
[RU11]   Jakob Rehof and Paweł Urzyczyn. Finite Combinatory Logic with Intersection Types. In *Pro-
           ceedings of TLCA'11*, volume 6690 of *LNCS*, pages 169–183. Springer, 2011.
[RU12]   Jakob Rehof and Paweł Urzyczyn. The Complexity of Inhabitation with Explicit Intersection. In
           Kozen Festschrift, volume 7230 of *LNCS*, pages 256–270. Springer, 2012.
[RV14]   Jakob Rehof and Moshe Y. Vardi. Design and Synthesis from Components (Dagstuhl Seminar
           14232). *Dagstuhl Reports*, 4(6):29–47, 2014. `http://drops.dagstuhl.de/opus/volltexte/2014/
           4683`.
[Sch24]   Moses I. Schönfinkel. Über die Bausteine der mathematischen Logik. *Mathematische Annalen*,
           92(3):305–316, 1924.
[Sin74]   Wilson E. Singletary. Many-one Degrees Associated with Partial Propositional Calculi. *Notre
           Dame Journal of Formal Logic*, XV(2):335–343, 1974.

[SMvdB97] Bernhard Steffen, Tiziana Margaria, and Michael von der Beeck. Automatic Synthesis of Linear
           Process Models from Temporal Constraints: An Incremental Approach. In *AAS97*, 1997.
[Sta79]   Richard Statman. Intuitionistic Propositional Logic is Polynomial-space Complete. *Theoretical
           Computer Science*, 9:67–72, 1979.
[Tho09]   Wolfgang Thomas. Facets of synthesis: Revisiting Churchs problem. In *Foundations of Software
           Science and Computational Structures*, pages 1–14. Springer, 2009.
[Wan87]   Mitchell Wand. Complete Type Inference for Simple Objects. In *LICS*, volume 87, pages 37–44,
           1987.
[Wan91]   Mitchell Wand. Type Inference for Record Concatenation and Multiple Inheritance. *Inf. Comput.*,
           93(1):1–15, 1991.
[WM80]   Daniel Weinreb and David A. Moon. Flavors: Message Passing in the Lisp Machine. Technical
           Report AI-M-602, Massachusetts Institute of Technology, Cambridge Artificial Intelligence Lab,
           1980.
[WY05]   Joe B. Wells and Boris Yakobowski. Graph-Based Proof Counting and Enumeration with
           Applications for Program Fragment Synthesis. In *LOPSTR 2004*, volume 3573 of *LNCS*, pages
           262–277. Springer, 2005.

Appendix A.

Proof of Lemma 4.8.

Proof. By definition, $\bigcap \mathbb{P}(\tau)$ is ω or an intersection of paths, therefore it is organized. Due to the identities $\sigma \to \omega = \omega$ and $\omega \cap \omega = \omega$ we have that if $\mathbb{P}(\tau) = \varnothing$ then $\tau = \omega$. We show by induction on the depth of the syntax tree of τ that $\bigcap \mathbb{P}(\tau) = \tau$.

Basis Step: If $\tau \equiv \omega$ or $\tau \equiv \alpha$ or $\tau \equiv a$, then $\bigcap \mathbb{P}(\tau) = \tau$ by definition.

Inductive Step:

    Case 1.1: $\tau \equiv \tau_1 \to \tau_2$ and $\mathbb{P}(\tau_2) = \varnothing$.

        Since $\mathbb{P}(\tau_2) = \varnothing$, we have $\tau_2 = \omega$, therefore $\tau = \omega = \bigcap \varnothing = \bigcap \mathbb{P}(\tau)$.

    Case 1.2: $\tau \equiv \tau_1 \to \tau_2$ and $\mathbb{P}(\tau_2) \neq \varnothing$.

$$\bigcap \mathbb{P}(\tau) = \bigcap \{\tau_1 \to \pi \mid \pi \in \mathbb{P}(\tau_2)\} = \tau_1 \to \bigcap \mathbb{P}(\tau_2) \stackrel{\text{IH}}{=} \tau_1 \to \tau_2 = \tau.$$

    Case 2: $\tau \equiv \tau_1 \cap \tau_2$.

$$\bigcap \mathbb{P}(\tau) = \bigcap \mathbb{P}(\tau_1) \cap \bigcap \mathbb{P}(\tau_2) \stackrel{\text{IH}}{=} \tau_1 \cap \tau_2 = \tau.$$

    Case 3.1: $\tau \equiv c(\tau_1)$ and $\mathbb{P}(\tau_1) = \varnothing$.

$$\bigcap \mathbb{P}(\tau) = c(\omega) = \tau.$$

    Case 3.2: $\tau \equiv c(\tau_1)$ and $\mathbb{P}(\tau_1) \neq \varnothing$.

$$\bigcap \mathbb{P}(\tau) = c(\bigcap \mathbb{P}(\tau_1)) \stackrel{\text{IH}}{=} c(\tau_1) = \tau$$

$\square$

    Proof of Lemma 4.10.

Proof. For "$\Leftarrow$" we have $\sigma \stackrel{\text{Lem. 4.8}}{=} \bigcap \mathbb{P}(\sigma) \leq \bigcap \mathbb{P}(\tau) \stackrel{\text{Lem. 4.8}}{=} \tau$ using the fact that $\sigma_1 \leq \tau_1$ and $\sigma_2 \leq \tau_2$ implies $\sigma_1 \cap \sigma_2 \leq \tau_1 \cap \tau_2$. For "$\Rightarrow$" we show by induction on the derivation of $\sigma \leq \tau$ that for each path $\pi \in \mathbb{P}(\tau)$ there exists a path $\pi' \in \mathbb{P}(\sigma)$ such that $\pi' \leq \pi$ and the desired properties hold.

Basis Step: In the following cases we have $\mathbb{P}(\sigma) \supseteq \mathbb{P}(\tau)$, therefore the desired properties hold via syntactical identity

- $\sigma \equiv \sigma \leq \sigma \equiv \tau$
- $\sigma \equiv \sigma \leq \omega \equiv \tau$
- $\sigma \equiv \omega \leq \omega \to \omega \equiv \tau$
- $\sigma \equiv \sigma' \cap \tau' \leq \sigma' \equiv \tau$ for some $\sigma', \tau' \in \mathbb{T}_{\mathbb{C}}$
- $\sigma \equiv \sigma' \cap \tau' \leq \tau' \equiv \tau$ for some $\sigma', \tau' \in \mathbb{T}_{\mathbb{C}}$
- $\sigma \equiv (\sigma' \to \tau_1') \cap (\sigma' \to \tau_2') \leq \sigma' \to \tau_1' \cap \tau_2' \equiv \tau$ for some $\sigma', \tau_1', \tau_2' \in \mathbb{T}_{\mathbb{C}}$
- $\sigma \equiv c(\sigma_1) \cap c(\tau_1) \leq c(\sigma_1 \cap \tau_1) \equiv \tau$ for some $\sigma_i, \tau_i \in \mathbb{T}_{\mathbb{C}}$ for $1 \leq i \leq n$

Inductive Step: Let $\pi \in \mathbb{P}(\tau)$.

    Case 1: $\sigma \equiv \sigma_1 \to \tau_1 \leq \sigma_2 \to \tau_2 \equiv \tau$ with $\sigma_2 \leq \sigma_1$ and $\tau_1 \leq \tau_2$.

        We have $\mathbb{P}(\sigma_2 \to \tau_2) \ni \pi \equiv \sigma_2 \to \pi_2$ for some path $\pi_2 \in \mathbb{P}(\tau_2)$. By the induction hypothesis there exists a path $\pi_1 \in \mathbb{P}(\tau_1)$ such that $\pi_1 \leq \pi_2$. Therefore,

$$\mathbb{P}(\sigma) \ni \sigma_1 \to \pi_1 \leq \sigma_2 \to \pi_2 \equiv \pi \text{ with } \sigma_2 \leq \sigma_1 \text{ and } \pi_1 \leq \pi_2$$

    Case 2: $\sigma \equiv \sigma \leq \tau_1 \cap \tau_2 \equiv \tau$ with $\sigma \leq \tau_1$ and $\sigma \leq \tau_2$.

        Since $\pi \in \mathbb{P}(\tau) = \mathbb{P}(\tau_1) \cup \mathbb{P}(\tau_2)$, we have $\pi \in \mathbb{P}(\tau_1)$ or $\pi \in \mathbb{P}(\tau_2)$. By the induction hypothesis in both cases there exists a path $\pi' \in \mathbb{P}(\sigma)$ such that $\pi' \leq \pi$ satisfying the desired properties.

Case 3: $\sigma \equiv c(\sigma_1) \le c(\tau_1) \equiv \tau$ with $\mathbb{P}(\tau_i) = \varnothing$.

Since $\pi \equiv c(\omega)$, we have $\pi' \le \pi$ satisfying the desired properties for any $\pi' \in \mathbb{P}(\sigma)$, noting that $\mathbb{P}(\sigma) \ne \varnothing$.

Case 4: $\sigma \equiv c(\sigma_1) \le c(\tau_1) \equiv \tau$ with $\sigma_1 \le \tau_1$ and $\mathbb{P}(\tau_1) \ne \varnothing$.

By Definition 4.6, we have $\pi \equiv c(\pi_\tau)$ for some $\pi_\tau \in \mathbb{P}(\tau_1)$. By the induction hypothesis there exists a $\pi_\sigma \in \mathbb{P}(\sigma_1)$ such that $\pi_\sigma \le \pi_\tau$. Therefore, $\mathbb{P}(\sigma) \ni c(\pi_\sigma) \le c(\pi_\tau) \equiv \pi$.

Case 5: $\sigma \le \sigma_1 \le \tau$.

By the induction hypothesis there exists a path $\pi_1 \in \mathbb{P}(\sigma_1)$ such that $\pi_1 \le \pi$. Again by the induction hypothesis there exists a path $\pi' \in \mathbb{P}(\sigma)$ such that $\pi' \le \pi_1$. Therefore, we have $\mathbb{P}(\sigma) \ni \pi' \le \pi_1 \le \pi$.

$\square$

Proof of Lemma 4.11.

Proof. $(2) \Rightarrow (1)$ is trivial using the rules $(\to E), (\le)$ and the property $(\sigma_1 \to \tau_1) \cap (\sigma_2 \to \tau_2) \le \sigma_1 \cap \sigma_2 \to \tau_1 \cap \tau_2$. For $(1) \Rightarrow (2)$, we use induction on the derivation of $\Delta \vdash_k C E_1 \dots E_m : \tau$. Wlog. $\tau \ne \omega$ and (similar to [DMRU12, Proposition 8]) the derivation $\Delta \vdash_k C E_1 \dots E_m : \tau$ does not contain atoms other than $\mathrm{atoms}(\Delta) \cup \mathrm{atoms}(\tau)$.

Basis Step: The last rule is $\dfrac{C : \sigma \in \Delta \qquad \mathrm{level}(S) \le k}{\Delta \vdash_k C : S(\sigma) \equiv \tau}$ (Var). For $P = \mathbb{P}_0(S(\sigma))$ we have

$\bigcap_{\pi \in P} \mathrm{tgt}_0(\pi) = \bigcap P = \bigcap \mathbb{P}_0(S(\sigma)) = \bigcap \mathbb{P}(S(\sigma)) \overset{\text{Lem. } 4.8}{=} S(\sigma) = \tau$.

Inductive Step:

Case 1: The last rule is $\dfrac{\Delta \vdash_k C E_1 \dots E_m : \sigma \qquad \sigma \le \tau}{\Delta \vdash_k C E_1 \dots E_m : \tau}$ $(\le)$.

Follows immediately from the induction hypothesis with $\bigcap_{\pi \in P} \mathrm{tgt}_m(\pi) \le \sigma \le \tau$.

Case 2: The last rule is $\dfrac{\Delta \vdash_k C E_1 \dots E_m : \tau_1 \qquad \Delta \vdash_k C E_1 \dots E_m : \tau_2}{\Delta \vdash_k C E_1 \dots E_m : \tau_1 \cap \tau_2 \equiv \tau}$ $(\cap)$.

By the induction hypothesis, there exist sets
$P_1, P_2 \subseteq \mathbb{P}_m(\bigcap\{S(\Delta(C)) \mid \mathrm{level}(S) \le k, \mathrm{atoms}(S) \subseteq \mathrm{atoms}(\Delta) \cup \mathrm{atoms}(\tau)\})$
such that $\bigcap_{\pi \in P_j} \mathrm{tgt}_m(\pi) \le \tau_j$ and $\Delta \vdash_k E_i : \bigcap_{\pi \in P_j} \mathrm{arg}_i(\pi)$ for $1 \le i \le m$ and $j \in \{1, 2\}$.
For $P = P_1 \cup P_2$ we obtain
(1) $\bigcap_{\pi \in P} \mathrm{tgt}_m(\pi) = \bigcap_{\pi \in P_1} \mathrm{tgt}_m(\pi) \cap \bigcap_{\pi \in P_2} \mathrm{tgt}_m(\pi) \le \tau_1 \cap \tau_2 = \tau$
(2) $\Delta \vdash_k E_i : \bigcap_{\pi \in P} \mathrm{arg}_i(\pi) = \bigcap_{\pi \in P_1} \mathrm{arg}_i(\pi) \cap \bigcap_{\pi \in P_2} \mathrm{arg}_i(\pi)$ for $1 \le i \le m$ using the induction hypothesis and the rule $(\cap)$.

Case 3: The last rule is $\dfrac{\Delta \vdash_k C E_1 \dots E_{m-1} : \sigma \to \tau \qquad \Delta \vdash_k E_m : \sigma}{\Delta \vdash_k C E_1 \dots E_m : \tau}$ $(\to E)$.

By the induction hypothesis, there exists a set
$P' \subseteq \mathbb{P}_{m-1}(\bigcap\{S(\Delta(C)) \mid \mathrm{level}(S) \le k, \mathrm{atoms}(S) \subseteq \mathrm{atoms}(\Delta) \cup \mathrm{atoms}(\tau)\})$
such that $\bigcap_{\pi \in P'} \mathrm{tgt}_{m-1}(\pi) \le \sigma \to \tau$. Let $P = \{\pi \in P' \mid \sigma \le \mathrm{arg}_m(\pi)\}$. By Lemma 4.10 for each path $\sigma \to \pi \in \mathbb{P}(\sigma \to \tau)$ there exists a path $\pi' \in P'$ such that $\mathrm{tgt}_{m-1}(\pi') \le \sigma \to \pi$ and $\mathrm{tgt}_{m-1}(\pi') = \mathrm{arg}_m(\pi') \to \mathrm{tgt}_m(\pi')$ with $\sigma \le \mathrm{arg}_m(\pi')$ and $\mathrm{tgt}_m(\pi') \le \pi$, therefore $\pi' \in P$. By Lemma 4.10 we obtain $\bigcap_{\pi \in P} \mathrm{tgt}_m(\pi) \le \tau$. Additionally, by the induction hypothesis $\Delta \vdash_k E_i : \bigcap_{\pi \in P'} \mathrm{arg}_i(\pi) \le \bigcap_{\pi \in P} \mathrm{arg}_i(\pi)$ for $1 \le i \le m - 1$, and $\Delta \vdash_k E_m : \sigma \le \bigcap_{\pi \in P} \mathrm{arg}_m(\pi)$.

$\square$

Appendix B. Running Example

Overview of terms

$\mathrm{Num} = \mathbf{Y}(\lambda\mathsf{myClass}.\lambda\mathsf{state}.$
>> `let` $\mathsf{self} = \mathsf{myClass}\ \mathsf{state}$ `in`
>>> $\langle get = \mathsf{state}, set = \lambda\mathsf{state}'.\mathsf{state}', succ = \mathsf{self}.set(\mathsf{self}.get + 1)\rangle$

$\mathrm{Comparable} = \lambda\mathsf{argClass}.\mathbf{Y}(\lambda\mathsf{myClass}.\lambda\mathsf{state}.$
>> `let` $\mathsf{super} = \mathsf{argClass}\ \mathsf{state}$ `in`
>> `let` $\mathsf{self} = \mathsf{myClass}\ \mathsf{state}$ `in`
>>> $\mathsf{super} \oplus \langle compare = \lambda o.(o.get == \mathsf{self}.get)\rangle)$

$\mathrm{Succ2} = \lambda\mathsf{argClass}.\mathbf{Y}(\lambda\mathsf{myClass}.\lambda\mathsf{state}.$
>> `let` $\mathsf{super} = \mathsf{argClass}\ \mathsf{state}$ `in`
>> `let` $\mathsf{self} = \mathsf{myClass}\ \mathsf{state}$ `in`
>>> $\mathsf{super} \oplus \langle succ2 = $ `let` $\mathsf{super}' = \mathsf{argClass}(\mathsf{super}.succ)$ `in` $\mathsf{super}'.succ\rangle)$

$\mathrm{SuccDelta} = \lambda\mathsf{argClass}.\mathbf{Y}(\lambda\mathsf{myClass}.\lambda\mathsf{state}.$
>> `let` $\mathsf{super} = \mathsf{argClass}\ \mathsf{state}$ `in`
>> `let` $\mathsf{self} = \mathsf{myClass}\ \mathsf{state}$ `in`
>>> $\mathsf{super} \oplus \langle succ = \lambda d.(\mathsf{super}.set(\mathsf{super}.get + d)\rangle))$

$\mathrm{Parity} = \lambda\mathsf{argClass}.\mathbf{Y}(\lambda\mathsf{myClass}.\lambda\mathsf{state}.$
>> `let` $\mathsf{super} = \mathsf{argClass}\ \mathsf{state}$ `in`
>> `let` $\mathsf{self} = \mathsf{myClass}\ \mathsf{state}$ `in`
>>> $\mathsf{super} \oplus \langle succ = \mathsf{super}.succ2\rangle)$

Overview of Λ_R types for any record type $\rho \in \mathbb{T}_{\langle\rangle}$

$\mathrm{Num} : \mathrm{Int} \rightarrow \langle get : \mathrm{Int}, set : \mathrm{Int} \rightarrow \mathrm{Int}, succ : \mathrm{Int}\rangle$

$\mathrm{Comparable} : (\mathrm{Int} \rightarrow \rho \cap \langle get : \mathrm{Int}\rangle) \rightarrow (\mathrm{Int} \rightarrow \rho + \langle compare : \langle get : \mathrm{Int}\rangle \rightarrow \mathrm{Bool}\rangle)$

$\mathrm{Succ2} : (\mathrm{Int} \rightarrow \rho \cap \langle succ : \mathrm{Int}\rangle) \rightarrow (\mathrm{Int} \rightarrow \rho + \langle succ2 : \mathrm{Int}\rangle)$

$\mathrm{SuccDelta} : (\mathrm{Int} \rightarrow \rho \cap \langle get : \mathrm{Int}, set : \mathrm{Int} \rightarrow \mathrm{Int}\rangle) \rightarrow (\mathrm{Int} \rightarrow \rho + \langle succ : \mathrm{Int} \rightarrow \mathrm{Int}\rangle)$

$\mathrm{Parity} : (\mathrm{Int} \rightarrow \rho \cap \langle succ2 : \mathrm{Int}\rangle) \rightarrow (\mathrm{Int} \rightarrow \rho + \langle succ : \mathrm{Int}\rangle)$

Overview of $\mathbb{T}_{\mathbb{C}}$ types

$\text{Num} : \text{Int} \to \langle\!\langle get(\text{Int}) \cap set(\text{Int} \to \text{Int}) \cap succ(\text{Int}) \rangle\!\rangle,$

$\text{Comparable} : \big((\text{Int} \to \langle\!\langle get(\text{Int}) \rangle\!\rangle) \to (\text{Int} \to \langle\!\langle compare(\langle\!\langle get(\text{Int}) \rangle\!\rangle \to \text{Bool}) \rangle\!\rangle)\big)$

$\qquad \cap \big((\text{Int} \to \langle\!\langle get(\alpha_{get}) \rangle\!\rangle) \to (\text{Int} \to \langle\!\langle get(\alpha_{get}) \rangle\!\rangle)\big)$

$\qquad \cap \big((\text{Int} \to \langle\!\langle set(\alpha_{set}) \rangle\!\rangle) \to (\text{Int} \to \langle\!\langle set(\alpha_{set}) \rangle\!\rangle)\big)$

$\qquad \cap \big((\text{Int} \to \langle\!\langle succ(\alpha_{succ}) \rangle\!\rangle) \to (\text{Int} \to \langle\!\langle succ(\alpha_{succ}) \rangle\!\rangle)\big)$

$\qquad \cap \big((\text{Int} \to \langle\!\langle succ2(\alpha_{succ2}) \rangle\!\rangle) \to (\text{Int} \to \langle\!\langle succ2(\alpha_{succ2}) \rangle\!\rangle)\big)$

$\text{Succ2} : \big((\text{Int} \to \langle\!\langle succ(\text{Int}) \rangle\!\rangle) \to (\text{Int} \to \langle\!\langle succ2(\text{Int}) \rangle\!\rangle)\big)$

$\qquad \cap \big((\text{Int} \to \langle\!\langle get(\alpha_{get}) \rangle\!\rangle) \to (\text{Int} \to \langle\!\langle get(\alpha_{get}) \rangle\!\rangle)\big)$

$\qquad \cap \big((\text{Int} \to \langle\!\langle set(\alpha_{set}) \rangle\!\rangle) \to (\text{Int} \to \langle\!\langle set(\alpha_{set}) \rangle\!\rangle)\big)$

$\qquad \cap \big((\text{Int} \to \langle\!\langle succ(\alpha_{succ}) \rangle\!\rangle) \to (\text{Int} \to \langle\!\langle succ(\alpha_{succ}) \rangle\!\rangle)\big)$

$\qquad \cap \big((\text{Int} \to \langle\!\langle compare(\alpha_{compare}) \rangle\!\rangle) \to (\text{Int} \to \langle\!\langle compare(\alpha_{compare}) \rangle\!\rangle)\big)$

$\text{SuccDelta} : \big((\text{Int} \to \langle\!\langle get(\text{Int}) \cap set(\text{Int} \to \text{Int}) \rangle\!\rangle) \to (\text{Int} \to \langle\!\langle succ(\text{Int} \to \text{Int}) \rangle\!\rangle)\big)$

$\qquad \cap \big((\text{Int} \to \langle\!\langle get(\alpha_{get}) \rangle\!\rangle) \to (\text{Int} \to \langle\!\langle get(\alpha_{get}) \rangle\!\rangle)\big)$

$\qquad \cap \big((\text{Int} \to \langle\!\langle set(\alpha_{set}) \rangle\!\rangle) \to (\text{Int} \to \langle\!\langle set(\alpha_{set}) \rangle\!\rangle)\big)$

$\qquad \cap \big((\text{Int} \to \langle\!\langle succ2(\alpha_{succ2}) \rangle\!\rangle) \to (\text{Int} \to \langle\!\langle succ2(\alpha_{succ2}) \rangle\!\rangle)\big)$

$\qquad \cap \big((\text{Int} \to \langle\!\langle compare(\alpha_{compare}) \rangle\!\rangle) \to (\text{Int} \to \langle\!\langle compare(\alpha_{compare}) \rangle\!\rangle)\big)$

$\text{Parity} : \big((\text{Int} \to \langle\!\langle succ2(\text{Int}) \rangle\!\rangle) \to (\text{Int} \to \langle\!\langle succ(\text{Int}) \rangle\!\rangle)\big)$

$\qquad \cap \big((\text{Int} \to \langle\!\langle get(\alpha_{get}) \rangle\!\rangle) \to (\text{Int} \to \langle\!\langle get(\alpha_{get}) \rangle\!\rangle)\big)$

$\qquad \cap \big((\text{Int} \to \langle\!\langle set(\alpha_{set}) \rangle\!\rangle) \to (\text{Int} \to \langle\!\langle set(\alpha_{set}) \rangle\!\rangle)\big)$

$\qquad \cap \big((\text{Int} \to \langle\!\langle succ2(\alpha_{succ2}) \rangle\!\rangle) \to (\text{Int} \to \langle\!\langle succ2(\alpha_{succ2}) \rangle\!\rangle)\big)$

$\qquad \cap \big((\text{Int} \to \langle\!\langle compare(\alpha_{compare}) \rangle\!\rangle) \to (\text{Int} \to \langle\!\langle compare(\alpha_{compare}) \rangle\!\rangle)\big)$