# Mixin Composition Synthesis Based on Intersection Types*

**Jan Bessai[1], Andrej Dudenhefner[1], Boris Düdder[1], Tzu-Chun Chen[2], Ugo de'Liguoro[3], and Jakob Rehof[1]**

1 **Technical University of Dortmund, Dortmund, Germany**
{jan.bessai, boris.duedder, andrej.dudenhefner, jakob.rehof}@cs.tu-dortmund.de
2 **Technical University of Darmstadt, Darmstadt, Germany**
tcchen@rbg.informatik.tu-darmstadt.de
3 **University of Torino, Torino, Italy**
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 corresponds to a mixin composition typed by the goal type.

**1998 ACM Subject Classification** F.4.1 Mathematical Logic – $\lambda$ Calculus and Related Systems

**Keywords and phrases** Record Calculus, Combinatory Logic, Type Inhabitation, Mixin, Intersection Type

**Digital Object Identifier** 10.4230/LIPIcs.TLCA.2015.76

## 1 Introduction

Starting with Cardelli's pioneering work [13], various typed $\lambda$-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. [1, 11, 23]).

Here, we focus on the synthesis of mixin compositions. In the object-oriented paradigm, mixins [9, 10] 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. 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

* This work was partially supported by EU COST Action IC1201: BETTY and MIUR PRIN CINA Prot. 2010LHT4KM, San Paolo Project SALT.

13th International Conference on Typed Lambda Calculi and Applications (TLCA'15).
Editor: Thorsten Altenkirch; pp. 76–91

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

programming languages miss native support for mixins, but they are an object of intensive study and research (e.g. [8, 24]). In this setting we aim at synthesizing classes from a library of mixins that can be used in programming languages like Java, which do not natively support mixins. Our particular modeling approach is inspired by modern language features (e.g. ECMAScript "`bind`") to preserve contexts in order to prevent programming errors [21].

We formalize synthesis of classes from a library of mixins as an instance of the relativized type inhabitation problem in bounded combinatory logic with intersection types [20]. Relativized type inhabitation is the decision problem: given a combinatory type context $\Delta$ and a type $\tau$ does there exist an applicative term $e$ such that $e$ has type $\tau$ under the type assumptions in $\Delta$? We denote type inhabitation by $\Delta \vdash_{\mathrm{BCL}} ? : \tau$ and implicitly include the problem of constructing a term inhabiting $\tau$.

Relativized type inhabitation, which is undecidable in general [20], is decidable in $k$-bounded combinatory logic $\mathsf{BCL}_k(\to,\cap)$, that is, combinatory logic typed with arrow and intersection types of depth at most $k$, and 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 $\mathsf{BCL}_k(\to,\cap)$ [20]. In the present paper, we enable combinatory synthesis of classes via intersection typed mixin combinators. Intersection types [4] play an important rôle in combinatory synthesis, because they allow for semantic specification of components and synthesis goals [20, 5].

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 $\sigma_i$ and $\tau_j$ respectively, and given a type $\tau$ specifying an unknown class, we may identify the class synthesis problem with the type inhabitation problem $\Delta \vdash_{\mathrm{BCL}} ? : \tau$. 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 [12, 15], and the system of intersection types from [4]. In doing so, we move from the system originally presented in [16], consisting of a type assignment system of intersection and record types to a $\lambda$-calculus which we enrich here with record merge operation (called "**with**" in [15]), 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 [4]. 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 [27, 26], is notoriously hard; to circumvent difficulties the theory of record subtyping in [15] (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 w.r.t. 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 we can type in a meaningful way in our system classes and mixins, where the former are essentially recursive records and the latter 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 $\mathsf{BCL}(\to,\cap)$. For this purpose we consider a conservative extension

of bounded combinatory logic, called $\mathsf{BCL}(\mathbb{T}_\mathbb{C})$, where we allow unary type constructors that are monotonic and distribute over intersection. We show that the (semi) algorithm solving inhabitation for $\mathsf{BCL}(\to,\cap)$ can be adapted to $\mathsf{BCL}(\mathbb{T}_\mathbb{C})$, by proving that the key properties necessary to solve the inhabitation problem in $\mathsf{BCL}(\to,\cap)$ are preserved in $\mathsf{BCL}(\mathbb{T}_\mathbb{C})$ and showing how the type-merge operator can be simulated in $\mathsf{BCL}(\mathbb{T}_\mathbb{C})$. In fact, type-merge is not monotonic in its second argument, due to the lack of negative information caused by the combination of + and $\cap$. Our work culminates in two theorems that ensure soundness and completeness of the so obtained method w.r.t. synthesis of classes by mixins composition.

**Related works.** This work evolves from the contributions [5, 17, 6] to the workshop ITRS'14. The papers that have inspired our work, mainly by Cook and others, have been cited above. The theme of using intersection types and bounded-polymorphism for typing object-oriented languages and inheritance has been treated in [14, 25]. Type inhabitation has been recently used for synthesis of object oriented code [22, 18], but to our best knowledge the present paper provides, for the first time, a theory of type-safe mixin composition synthesis based on the component-oriented approach of combinatory logic synthesis.

## 2 Intersection Types for Mixins and Classes

### 2.1 Intersection and record types

We consider a type-free $\lambda$-calculus of extensible records, equipped with a merge operator. The term syntax is defined by the following grammar:

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

where $x \in \mathbf{Var}$ and $l \in \mathbf{Label}$ range over denumerably many *term variables* and *labels* respectively, and the sets of indexes $I$ are finite. Free and bound variables are defined as usual for ordinary $\lambda$-calculus, and we name $\Lambda^0_R$ the set of all closed terms in $\Lambda_R$; 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 [3]; in particular application associates to the left and external parentheses are omitted when unnecessary; also the dot notation for record selection takes precedence over $\lambda$, 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 $lbl(\langle l_i = M_i \mid i \in I\rangle) = \{l_i \mid i \in I\}$. 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 with 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 8.

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

▶ **Definition 1** ($\Lambda_R$ reduction)**.** Reduction $\longrightarrow \subseteq \Lambda^2_R$ is the least compatible relation such that:

$$\begin{array}{lrcll} (\beta) & (\lambda x.M)N & \longrightarrow & M\{N/x\} & \\ (sel) & \langle l_i = M_i \mid i \in I\rangle.l_j & \longrightarrow & M_j & \text{if } j \in I \\ (\oplus) & \langle l_i = M_i \mid i \in I\rangle \oplus \langle l_j = N_j \mid\ j \in J\rangle & \longrightarrow & \langle l_i = M_i,\ l_j = N_j \mid i \in I \setminus J,\ j \in J\rangle & \end{array}$$

We claim that $\longrightarrow^*$ is Church-Rosser, and that $(M \oplus R_1) \oplus R_2$ is equivalent to $M \oplus (R_1 \oplus R_2)$ under any reasonable observational semantics (e.g. by extending to $\Lambda_R$ applicative bisimulation from the lazy $\lambda$-calculus).

Record merge subsumes field update: $M.l := N \equiv M \oplus \langle l = N \rangle$, but merge is not uniformly definable in terms of update as long as labels are not expressions in the calculus.

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 2** (Intersection types for $\Lambda_R$).

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

where $a$ ranges over *type constants*, $l \in \mathbf{Label}$.

We use $\sigma, \tau$, possibly with sub and superscripts, for types in $\mathbb{T}$ and $\rho, \rho_i$, 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 $\lambda$-calculus, the intended meaning of types are sets, provided a set theoretic interpretation of constants $a$.

Following [4], type semantics is given axiomatically by means of the subtyping relation $\leq$, that can be interpreted as subset inclusion. It is the least pre-order over $\mathbb{T}$ such that:

▶ **Definition 3** (Type inclusion: arrow and intersection types).

$$\begin{array}{lll} \sigma \leq \omega, & \omega \leq \omega \to \omega, & \\ \sigma \cap \tau \leq \sigma, & \sigma \cap \tau \leq \tau, & \sigma \leq \tau_1 \ \& \ \sigma \leq \tau_2 \Rightarrow \sigma \leq \tau_1 \cap \tau_2, \\ (\sigma \to \tau_1) \cap (\sigma \to \tau_2) \leq \sigma \to \tau_1 \cap \tau_2 & & \sigma_2 \leq \sigma_1 \ \& \ \tau_1 \leq \tau_2 \Rightarrow \sigma_1 \to \tau_1 \leq \sigma_2 \to \tau_2 \end{array}$$

We write $\sigma = \tau$ for $\sigma \leq \tau$ and $\tau \leq \sigma$.

▶ **Definition 4** (Type inclusion: record types).

$$\begin{array}{lll} \langle l : \sigma \rangle \cap \langle l : \tau \rangle \leq \langle l : \sigma \cap \tau \rangle, & \sigma \leq \tau \Rightarrow \langle l : \sigma \rangle \leq \langle l : \tau \rangle, & \\ \langle l : \sigma \rangle \leq \langle\rangle, & \langle l : \sigma \rangle + \langle\rangle = \langle l : \sigma \rangle = \langle\rangle + \langle l : \sigma \rangle, & \\ \langle l : \sigma \rangle + \langle l : \tau \rangle = \langle l : \tau \rangle, & \langle l : \sigma \rangle + \langle l' : \tau \rangle = \langle l : \sigma \rangle \cap \langle l' : \tau \rangle & (l \neq l'), \\ \langle l : \sigma \rangle + (\langle l : \tau \rangle \cap \rho) = \langle l : \tau \rangle \cap \rho, & \langle l : \sigma \rangle + (\langle l' : \tau \rangle \cap \rho) = \langle l' : \tau \rangle \cap (\langle l : \sigma \rangle + \rho) & (l \neq l'). \end{array}$$

While Definition 3 is standard after [4], comments on Definition 4 are in order. 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 $\sigma$; therefore $\langle l : \sigma \rangle \cap \langle l : \tau \rangle$ is the type of records having label $l$ with values both of type $\sigma$ and $\tau$, that is of type $\sigma \cap \tau$. In fact the equation $\langle l : \sigma \rangle \cap \langle l : \tau \rangle = \langle l : \sigma \cap \tau \rangle$ is derivable. 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 $\sigma$ and $\tau$ 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 abbreviation $\langle l_i : \sigma_i \mid i \in I \neq \varnothing \rangle = \bigcap_{i \in I} \langle l_i : \sigma_i \rangle$ 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 8 below, $\rho_1 + \rho_2$ is the type of all records obtained by merging a record of type $\rho_1$ with a record of type $\rho_2$, 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 5.**
1. $(\forall j \in J \subseteq I.\ \sigma_j \leq \tau_j) \Rightarrow \langle l_i : \sigma_i \mid i \in I\rangle \leq \langle l_j : \tau_j \mid j \in J\rangle$,
2. $\langle l_i : \sigma_i \mid i \in I\rangle + \langle l_j : \tau_j \mid j \in J\rangle = \langle l_i : \sigma_i, m_j : \tau_j \mid i \in I \setminus J, j \in J\rangle$,
3. $\forall \rho \in \mathbb{T}_{\langle\rangle}.\ \exists\ \langle l_i : \sigma_i \mid i \in I\rangle.\ \rho = \langle l_i : \sigma_i \mid i \in I\rangle$.

Part (1) of Lemma 5 states that subtyping among intersection of unary record types subsumes subtyping in width and depth of ordinary record types from the literature. Part (2) shows that the + type constructor reflects at the level of types the operational behavior of the merge operator $\oplus$. Part (3) 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. However they play a key role in typing mixins, motivating the issue of control of negative information in the synthesis process: see sections 2.2 and 4. More properties of subtyping record types w.r.t. + and $\cap$ are listed in the next lemma. Let us preliminary define the map $lbl : \mathbb{T}_{\langle\rangle} \to \wp(\mathbf{Label})$ (where $\wp(\mathbf{Label})$ is the powerset of **Label**) by:

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

Then we immediately have:

▶ **Lemma 6.**
1. $\rho_1 = \rho_2 \Rightarrow lbl(\rho_1) = lbl(\rho_2)$,
2. $lbl(\rho_1) \cap lbl(\rho_2) = \varnothing \Rightarrow \rho_1 + \rho_2 = \rho_1 \cap \rho_2$.

In (1) above $\rho_1 = \rho_2$ is $\rho_1 \leq \rho_2 \leq \rho_1$. About (2) note that condition $lbl(\rho_1) \cap lbl(\rho_2) = \varnothing$ is essential, since $\rho_1 + \rho_2 \neq \rho_2 + \rho_1$ in general, as it immediately follows by Lemma 5.2.

▶ **Lemma 7.**
1. $(\rho_1 + \rho_2) + \rho_3 = \rho_1 + (\rho_2 + \rho_3)$,
2. $(\rho_1 \cap \rho_2) + \rho_3 = (\rho_1 + \rho_3) \cap (\rho_2 + \rho_3)$,
3. $\rho_1 \leq \rho_2 \Rightarrow \rho_1 + \rho_3 \leq \rho_2 + \rho_3$,
4. $\rho_1 + \rho_2 = \rho_1 \cap \rho_2 \Leftrightarrow \rho_1 + \rho_2 \leq \rho_1$.

▶ Remark. In general $\rho_1 + (\rho_2 \cap \rho_3) \neq (\rho_1 + \rho_3) \cap (\rho_2 + \rho_3)$: 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 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 $(\rho_1 + \rho_3) \cap (\rho_2 + \rho_3) \leq \rho_1 + (\rho_2 \cap \rho_3)$. On the other hand $\rho_2 \leq \rho_3 \not\Rightarrow \rho_1 + \rho_2 \leq \rho_1 + \rho_3$. Indeed:

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

even if $\langle l_1 : \sigma_1, l_2 : \sigma_2\rangle \leq \langle l_2 : \sigma_2\rangle$. From this and (3) of Lemma 7, we conclude that + is monotonic in its first argument, but not in its second one.

We come now to the type assignment system. A *basis* (also called a context in the literature) is a finite set $\Gamma = \{x_1 : \sigma_n, \ldots, x_n : \sigma_n\}$, 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 [4], also called **BCD** in the literature.

▶ **Definition 8** (Type Assignment). The rules of the assignment system are:

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

$$\frac{\Gamma\vdash M:\sigma\to\tau \qquad \Gamma\vdash N:\sigma}{\Gamma\vdash MN:\tau}\,(\to E) \qquad \frac{\Gamma\vdash M:\sigma \quad \Gamma\vdash M:\tau}{\Gamma\vdash M:\sigma\cap\tau}\,(\cap)$$

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

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

$$\frac{\Gamma\vdash M:\langle l:\sigma\rangle}{\Gamma\vdash M.l:\sigma}\,(sel) \qquad \frac{\Gamma\vdash M:\rho_1 \qquad \Gamma\vdash R:\rho_2 \quad (*)}{\Gamma\vdash M\oplus R:\rho_1+\rho_2}\,(+)$$

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

Using Lemma 5.1, 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 [10], 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 r.e.) 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 $\Lambda_R$ 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 of 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.

▶ **Lemma 9.** *Let $\sigma\neq\omega$:*

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

▶ **Theorem 10** (Subject reduction). $\Gamma \vdash M : \sigma \;\&\; M \longrightarrow N \Rightarrow \Gamma \vdash N : \sigma$.

**Proof Sketch.** The proof is by cases of reduction rules, using Lemma 9. The only relevant case is when $M \equiv R_1 \oplus R_2$, with $R_1 \equiv \langle l_i = M_i \mid i \in I\rangle$ and $R_2 \equiv \langle l_j = N_j \mid \; j \in J\rangle$, and $N \equiv \langle l_i = M_i,\; l_j = N_j \mid i \in I \setminus J,\; j \in J\rangle$. By Lemma 9.6 we may suppose w.l.o.g. that $\sigma = \rho_1 + \rho_2$, and that $\rho_1 = \langle l_i : \sigma_i \mid i \in I'\rangle$ for some $I' \subseteq I$, and $\rho_2 = \langle l_j : \tau_j \mid j \in J'\rangle$ for some $J' \subseteq J$; but also we know that $J' = J$, because of condition $(*)$.

Now by Lemma 5.2, $\rho_1 + \rho_2 = \langle l_i : \sigma_i, l_j : \tau_j \mid i \in I' \setminus J, j \in J\rangle$; on the other hand by Lemma 9.4, we know that $\Gamma \vdash M_i : \sigma_i$ for all $i \in I'$, $\Gamma \vdash N_j : \tau_j$ for all $j \in J$, and therefore we conclude that $\Gamma \vdash N : \langle l_i : \sigma_i, l_j : \tau_j \mid i \in I' \setminus J, j \in J\rangle$ by multiple applications of rules $(rec)$ and $(\cap)$. ◀

## 2.2 Class and Mixin combinators

The following definition of classes and mixins is inspired by [15] and [10] respectively, though with some departures to be discussed below. To make the description more concrete, in the examples we add constants to $\Lambda_R$.

Recall that a *combinator* is a term in $\Lambda_R^0$, 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, since all fixed point combinators have the same types in $\mathbf{BCD}$).

▶ **Definition 11.** Let $\mathsf{myClass}, \mathsf{state}$ and $\mathsf{argClass}$ be (term) variables and $\mathbf{Y}$ be a fixed point combinator; then we define the following sets of combinators:

$$\begin{array}{lrcl} \text{Class:} & C & ::= & \mathbf{Y}(\lambda\,\mathsf{myClass}\;\lambda\,\mathsf{state}.\,\langle l_i = N_i \mid i \in I\rangle) \\ \text{Mixin:} & M & ::= & \lambda\,\mathsf{argClass}.\,\mathbf{Y}(\lambda\,\mathsf{myClass}\,\lambda\,\mathsf{state}.\,(\mathsf{argClass}\;\mathsf{state}) \oplus \langle l_i = N_i \mid i \in I\rangle) \end{array}$$

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

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

$$C \equiv \mathbf{Y}(\lambda\,\mathsf{myClass}\;\lambda\,\mathsf{state}.\;\texttt{let}\;\;\mathsf{self} = (\mathsf{myClass}\;\mathsf{state})\;\;\texttt{in}\;\;\langle l_i = N_i \mid i \in I\rangle). \tag{1}$$

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:

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

where the variable $\mathsf{self}$ is used in the method bodies $N_i$ to call other methods from the same object. Note that the recursive parameter $\mathsf{myClass}$ might occur in the $N_i$ in subterms other than $(\mathsf{myClass}\;\mathsf{state})$, 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 [15] §4, but for the explicit identification of $\mathsf{self}$ with $(\mathsf{myClass}\;\mathsf{state})$.

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 $\sigma$ (a type of its state) and a sequence of types $\rho_1, \ldots, \rho_n \in \mathbb{T}_{\langle\rangle}$ such that for all $i < n$:

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

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 $0 < i \leq n$. In general one has more expressive types, depending on the typings of the $N_i$ in $R$ (see example 12 below). It follows that:

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

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 $\tau_1, \ldots, \tau_n$, we conclude that the typing of classes has the following shape (where $\rho = \rho_n$):

$$\vdash C \equiv \mathbf{Y}(\lambda\,\mathsf{myClass}\ \lambda\,\mathsf{state}.\, \langle l_i = N_i \mid i \in I \rangle) : \sigma \to \rho \tag{2}$$

In conclusion the type of a class $C$ is the arrow from the type of the state $\sigma$ to a type $\rho$ of its instances.

▶ **Example 12.** The class Point has an integer state and contains the method *get* to retrieve the state, *set* to update the current state and *shift* to add a value to the current state.

$$\begin{aligned}\mathsf{Point} = \mathbf{Y}(&\lambda\mathsf{myClass}.\lambda\mathsf{state}.\texttt{let}\ \mathsf{self} = \mathsf{myClass}\ \mathsf{state}\ \texttt{in}\\ &\langle \mathit{get} = \mathsf{state}, \mathit{set} = \lambda\mathsf{state}'.\mathsf{state}', \mathit{shift} = \lambda d.\mathsf{self}.\mathit{set}(\mathsf{self}.\mathit{get} + d)\rangle)\end{aligned}$$

Note that in a setting without references, we rely on purely functional state updates. Therefore, every function returns a new state that can be used to construct the new object. An example for this is set, which just returns the new state.

To type Point we have $\vdash \mathsf{Point} : \mathrm{Int} \to \rho_{\mathrm{Point}}$ where $\rho_{\mathrm{Point}} = \rho_2$ and

$$\begin{aligned}\rho_1 &= \langle \mathit{get} : \mathrm{Int}, \mathit{set} : \mathrm{Int} \to \mathrm{Int}, \mathit{shift} : \omega \rangle\\ \rho_2 &= \langle \mathit{get} : \mathrm{Int}, \mathit{set} : \mathrm{Int} \to \mathrm{Int}, \mathit{shift} : \mathrm{Int} \to \mathrm{Int} \rangle\end{aligned}$$

using $\mathbf{Y} : (\omega \to \mathrm{Int} \to \rho_1) \cap ((\mathrm{Int} \to \rho_1) \to \mathrm{Int} \to \rho_2) \to \mathrm{Int} \to \rho_2$

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

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

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 [10]. Note that our mixins are not the same as class modificators (also called wrappers e.g. in [9]) because the latter do not take the instantiation of a class as the value of $\mathsf{super}$, but the class definition, namely the function defining the class *before* taking its fixed point.

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 $\rho^2_1, \ldots, \rho^2_n \in \mathbb{T}_{\langle\rangle}$ of record types such that for all $1 \leq i < n$ it is true that $\mathit{lbl}(R) = \mathit{lbl}(\rho^2_i)$ and such that, setting $\Gamma_0 = \{\mathsf{argClass} : \sigma^1 \to \rho^1, \mathsf{myClass} : \omega, \mathsf{state} : \sigma^1 \cap \sigma^2\}$ and $\Gamma_i = \{\mathsf{argClass} : \sigma^1 \to \rho^1, \mathsf{myClass} : (\sigma^1 \cap \sigma^2) \to \rho^1 + \rho^2_i, \mathsf{state} : \sigma^1 \cap \sigma^2\}$ for all $1 \leq i < n$, we may deduce for all $0 \leq i < n$:

$$\dfrac{\dfrac{\dfrac{\Gamma_i \vdash \mathsf{state} : \sigma^1 \cap \sigma^2}{\Gamma_i \vdash \mathsf{state} : \sigma^1}\,(\leq)}{\Gamma_i \vdash \mathsf{argClass}\ \mathsf{state} : \rho^1}\,(\to E) \qquad \Gamma_i \vdash R : \rho^2_{i+1} \qquad \mathit{lbl}(R) = \mathit{lbl}(\rho^2_{i+1})}{\Gamma_i \vdash (\mathsf{argClass}\ \mathsf{state}) \oplus R : \rho^1 + \rho^2_{i+1}}\,(+)$$

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

$$\mathsf{argClass} : \sigma^1 \to \rho^1 \vdash \lambda\, \mathsf{myClass}\, \lambda\, \mathsf{state}.\, (\mathsf{argClass}\ \mathsf{state}) \oplus R : \\ ((\sigma^1 \cap \sigma^2) \to (\rho^1 + \rho^2_i)) \to (\sigma^1 \cap \sigma^2) \to (\rho^1 + \rho^2_{i+1})$$

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

$$\vdash M \equiv \lambda\, \mathsf{argClass}.\, \mathbf{Y}(\lambda\, \mathsf{myClass}\, \lambda\, \mathsf{state}.\, (\mathsf{argClass}\ \mathsf{state}) \oplus R) : \\ (\sigma^1 \to \rho^1) \to (\sigma^1 \cap \sigma^2) \to (\rho^1 + \rho^2) \quad (3)$$

Spelling out this type, we can say that $\sigma^1$ 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 $\sigma^1$. $\rho^1$ expresses the requirements of $M$ about the methods of the argument-class, i.e. what is assumed to hold for the usages of $\mathsf{super}$ and $\mathsf{argClass}$ 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.

▶ **Example 13.** The mixin Movable, provided an argument class that contains a *set* and a *shift* method, creates a new class with (potentially overwritten) methods *set* and *move* along with delegated methods. The method *set* is fixed to set the underlying state to 1 and the method *move* shifts the state by 1.

$$\mathsf{Movable} = \lambda \mathsf{argClass}.\mathbf{Y}(\lambda \mathsf{myClass}.\lambda \mathsf{state}.\mathtt{let}\ \mathsf{super} = \mathsf{argClass}\ \mathsf{state}\ \mathtt{in} \\ \mathtt{let}\ \mathsf{self} = \mathsf{myClass}\ \mathsf{state}\ \mathtt{in} \\ \mathsf{super} \oplus \langle \mathit{set} = \mathsf{super}.\mathit{set}(1), \mathit{move} = \mathsf{self}.\mathit{shift}(1)\rangle)$$

Note that to update $\mathsf{self}$ and $\mathsf{super}$ one can use $\mathsf{myClass}$ and $\mathsf{argClass}$. Generalizing for all $\rho \in \mathbb{T}_{\langle\rangle}$ following the argumentation above we can choose:

$$\rho^1 = \rho \cap \langle \mathit{set} : \mathrm{Int} \to \mathrm{Int}, \mathit{shift} : \mathrm{Int} \to \mathrm{Int}\rangle \qquad \rho^2 = \langle \mathit{set} : \mathrm{Int}, \mathit{move} : \mathrm{Int}\rangle \\ \sigma^1 = \mathrm{Int} \qquad \sigma^2 = \omega \\ \rho^2_1 = \langle \mathit{set} : \mathrm{Int}, \mathit{move} : \omega\rangle \qquad \rho^2_2 = \langle \mathit{set} : \mathrm{Int}, \mathit{move} : \mathrm{Int}\rangle$$

Using these choices we can type $\mathbf{Y}$ by

$$\vdash \mathbf{Y} : (\omega \to (\mathrm{Int} \to \rho^1 + \rho^2_1)) \cap ((\mathrm{Int} \to \rho^1 + \rho^2_1) \to (\mathrm{Int} \to \rho^1 + \rho^2_2)) \to (\mathrm{Int} \to \rho^1 + \rho^2_2)$$

and obtain the typings of Movable for all $\rho$:

$$\dfrac{\dfrac{\dots}{\vdash \mathrm{Movable} : (\mathrm{Int} \to \rho \cap \langle \mathit{set} : \mathrm{Int} \to \mathrm{Int}, \mathit{shift} : \mathrm{Int} \to \mathrm{Int}\rangle) \to (\mathrm{Int} \to \rho^1 + \rho^2_2)}}{\vdash \mathrm{Movable} : (\mathrm{Int} \to \rho \cap \langle \mathit{set} : \mathrm{Int} \to \mathrm{Int}, \mathit{shift} : \mathrm{Int} \to \mathrm{Int}\rangle) \to (\mathrm{Int} \to \rho + \langle \mathit{set} : \mathrm{Int}, \mathit{move} : \mathrm{Int}\rangle)}\ (\leq)$$

## 3 Encoding of Record Types in Bounded Combinatory Logic

Our main goal is to combine type information given by intersection types for $\Lambda_R$ and the capabilities of the logical programming language given by BCL inhabitation to synthesize meaningful mixin compositions as terms of a combinatory logic. Such *combinatory terms* are formed by application of combinators from a *repository* (combinatory logic context) $\Delta$.

▶ **Definition 14** (Combinatory Term). $E, E' ::= C \mid (E\ E'),\ C \in dom(\Delta)$

We create repositories of typed combinators that can be considered logic programs for the existing BCL synthesis framework (CL)S [7] to reason about semantics of such compositions. The underlying type system of (CL)S is an extension of the intersection type system **BCD** [4] 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 +. While (CL)S implements covariant constructors of arbitrary arity, we only use unary constructors, which are sufficient for our encoding.

▶ **Definition 15** (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, $\alpha$ over type variables and $c$ over unary constructors $\mathbb{C}$.

$\mathbb{T}_{\mathbb{C}}$ adds the following two subtyping axioms to the **BCD** system

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

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

▶ **Definition 16** (Type Assignment in $\mathbb{T}_{\mathbb{C}}$).

$$\frac{C : \tau \in \Delta \qquad S \text{ Substitution}}{\Delta \vdash_{\mathrm{BCL}} C : S(\tau)}\ (\mathrm{Var}) \qquad \frac{\Delta \vdash_{\mathrm{BCL}} E : \sigma \to \tau \qquad \Delta \vdash_{\mathrm{BCL}} E' : \sigma}{\Delta \vdash_{\mathrm{BCL}} EE' : \tau}\ (\to E)$$

$$\frac{\Delta \vdash_{\mathrm{BCL}} E : \sigma \quad \Delta \vdash_{\mathrm{BCL}} E : \tau}{\Delta \vdash_{\mathrm{BCL}} E : \sigma \cap \tau}\ (\cap) \qquad \frac{\Delta \vdash_{\mathrm{BCL}} E : \sigma \qquad \sigma \leq \tau}{\Delta \vdash_{\mathrm{BCL}} E : \tau}\ (\leq)$$

We extend the necessary property of *beta-soundness* and a notion of *paths* and *organized types* from [20] to constructors in the following way.

▶ **Lemma 17** (Extended Beta-Soundness). *If* $\bigcap_{i \in I}(\sigma_i \to \tau_i) \cap \bigcap_{j \in J} c_j(\tau_j) \cap \bigcap_{k \in K} \alpha_k \cap \bigcap_{k' \in K'} a_{k'} \leq c(\tau)$, *then* $\{j \in J \mid c_j = c\} \neq \varnothing$ *and* $\bigcap\{\tau_j \mid j \in J, c_j = c\} \leq \tau$.

▶ **Definition 18** (Path). A *path* $\pi$ is a type of the form: $\pi ::= a \mid \alpha \mid \tau \to \pi \mid c(\omega) \mid c(\pi)$, where $\alpha$ is a variable, $\tau$ is a type, $c$ is a constructor and $a$ is a constant.

▶ **Definition 19** (Organized Type). A type $\tau$ is called organized, if it is an intersection of paths $\tau \equiv \bigcap_{i \in I} \tau_i$, where $\tau_i$ for $i \in I$ are paths.

Similarly, we obtain the following property of subtyping w.r.t. organized types.

▶ **Lemma 20.** *Given two organized types* $\tau \equiv \bigcap_{i \in I} \tau_i$ *and* $\sigma \equiv \bigcap_{j \in J} \sigma_j$, *we have* $\tau \leq \sigma$ *iff for all* $j \in J$ *there exists an* $i \in I$ *with* $\tau_i \leq \sigma_j$.

Note that for all intersection types there exists an equivalent organized intersection type coinciding with the notion of strict intersection types [2].

For a set of typed combinators $\Delta$ and a type $\tau \in \mathbb{T}_{\mathbb{C}}$ we say $\tau$ is inhabitable in $\Delta$, if there exists a combinatory term $E$ such that $\Delta \vdash_{\mathrm{BCL}} E : \tau$. In the following we fix a finite set of labels $\mathcal{L} \subseteq$ **Label** that are used in the particular domain of interest for mixin composition synthesis.

### Records as Unary Covariant Distributing Constructors

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

$$[\![\cdot]\!]\colon \mathbb{T} \to \mathbb{T}_{\mathbb{C}}, [\![\tau]\!] = \begin{cases} \tau & \text{if } \tau \equiv \omega \text{ or } \tau \equiv a \\ [\![\tau_1]\!] \to [\![\tau_2]\!] & \text{if } \tau \equiv \tau_1 \to \tau_2 \\ [\![\tau_1]\!] \cap [\![\tau_2]\!] & \text{if } \tau \equiv \tau_1 \cap \tau_2 \\ \langle\!\langle l(\tau) \rangle\!\rangle & \text{if } \tau \equiv \langle l : \tau \rangle \\ \langle\!\langle \omega \rangle\!\rangle & \text{if } \tau \equiv \langle\rangle \\ \text{undefined} & \text{else} \end{cases}$$

Since atomic records are covariant and distribute over $\cap$, the presented translation preserves subtyping. 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 $I \neq \varnothing$.

Note that the translation function $[\![\cdot]\!]$ is not defined for types containing + in $\mathbb{T}$. Additionally, + has non-monotonic properties and therefore cannot be immediately represented by a covariant type constructor. Simply applying Lemma 5(3) is impossible, if the left-hand side of + is all-quantified. There are two possibilities to deal with this situation. The first option is extending the type-system used for inhabitation. Here, the main difficulty is that existing versions of the inhabitation algorithm crucially rely on the separation of intersections into paths [20]. As demonstrated in the remark accompanying Lemma 7, 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 the logical programming language given by $\mathsf{BCL}(\mathbb{T}_{\mathbb{C}})$ inhabitation. Specifically, encoding $\mathbb{T}$ types containing + as $\mathbb{T}_{\mathbb{C}}$ types accompanied by following repositories $\Delta_{\mathcal{L}}$ and $\Delta_{\wp(\mathcal{L})}$ suited for $\mathsf{BCL}(\mathbb{T}_{\mathbb{C}})$ inhabitation. We introduce $|\mathcal{L}|$ distinct variables $\alpha_{l'}$ indexed by $l' \in \mathcal{L}$ and $2^{|\mathcal{L}|}$ distinct constructors $w_L(\cdot)$ indexed by $L \subseteq \mathcal{L}$.

$$\Delta_{\mathcal{L}} = \{W_{\{l\}} : w_{\{l\}}(\bigcap_{l'\in\mathcal{L}\setminus\{l\}} \langle\!\langle l'(\alpha_{l'})\rangle\!\rangle) \mid l \in \mathcal{L}\}$$

$$\Delta_{\wp(\mathcal{L})} = \{W_L : w_{\{l_1\}}(\alpha) \to w_{\{l_2\}}(\alpha) \to \ldots \to w_{\{l_k\}}(\alpha) \to w_L(\alpha) \mid k \geq 2, \{l_1, \ldots, l_k\} = L \subseteq \mathcal{L}\}\,.$$

These repositories are purely logical in a sense that they do not represent terms in $\Lambda_R$ but encode necessary side conditions in the logic program. In particular, we formalize the conditions for the absence of a label in a record type by the following Lemma 21. This encoding of negative information is crucial to encode non-monotonic properties of +.

▶ **Lemma 21.** *Let $l \in \mathcal{L}$ be a label and let $\tau \in \mathbb{T}$ be a type such that $[\![\tau]\!]$ is defined. $w_{\{l\}}([\![\tau]\!])$ is inhabitable in $\Delta_{\mathcal{L}} \cup \Delta_{\wp(\mathcal{L})}$ iff $\tau \in \mathbb{T}_{\langle\rangle} \cup \{\omega\}$ with $l \notin lbl(\tau)$.*

**Proof Sketch.** Let $l \in \mathcal{L}$ be a label.

**(⇒)** Let wlog. $[\![\tau]\!] \equiv \bigcap_{i\in I} \tau_i$ be an organized intersection type such that $w_{\{l\}}([\![\tau]\!])$ is inhabitable in $\Delta_{\mathcal{L}} \cup \Delta_{\wp(\mathcal{L})}$. The only combinator with a matching target to inhabit $w_{\{l\}}([\![\tau]\!])$ is $W_{\{l\}}$. Due to distributivity of type constructors there exists a substitution $S$ such that $S(w_{\{l\}}(\bigcap_{l'\in\mathcal{L}\setminus\{l\}} \langle\!\langle l'(\alpha_{l'})\rangle\!\rangle)) \leq w_{\{l\}}([\![\tau]\!])$. By Lemma 20 followed by Lemma 17 we obtain for each $i \in I$ that there exists an $l' \in \mathcal{L} \setminus \{l\}$ such that $\tau_i = \langle\!\langle l'([\![\sigma_i]\!])\rangle\!\rangle$ for some type $\sigma_i \in \mathbb{T}$. By definition of $[\![\cdot]\!]$ and distributivity we obtain $\tau \in \mathbb{T}_{\langle\rangle} \cup \{\omega\}$ with $l \notin lbl(\tau)$.

**(⇐)** Let $\tau = \bigcap_{l'\in L} \langle l' : \tau_{l'}\rangle$ (resp. $\langle\rangle$) for some $L \subseteq \mathcal{L} \setminus \{l\}$ and types $\tau_{l'} \in \mathbb{T}$ for $l' \in L$. We have $W_{\{l\}} : S(w_{\{l\}}(\bigcap_{l'\in\mathcal{L}\setminus\{l\}} \langle\!\langle l'(\alpha_{l'})\rangle\!\rangle)) \leq w_{\{l\}}([\![\tau]\!])$ for a substitution $S(\alpha_{l'}) = \begin{cases} [\![\tau_{l'}]\!] & \text{if } l' \in L \\ \omega & \text{else} \end{cases}$

◀

▶ **Lemma 22.** *Let $L \subseteq \mathcal{L}$ be a non-empty set of labels and let $\tau \in \mathbb{T}$ be a type such that $[\![\tau]\!]$ is defined. $w_L([\![\tau]\!])$ is inhabitable in $\Delta_{\mathcal{L}} \cup \Delta_{\wp(\mathcal{L})}$ iff $\tau \in \mathbb{T}_{\langle\rangle} \cup \{\omega\}$ with $L \cap lbl(\tau) = \varnothing$.*

**Proof Sketch.** Using $W_L$ and Lemma 21 for each argument of $W_L$. ◀

Our intermediate goal is to use inhabitation in $\Delta_{\mathcal{L}} \cup \Delta_{\wp(\mathcal{L})}$ to translate types of the shape $\rho + \bigcap_{l \in L} \langle l : \tau_l \rangle$ into $\rho \cap \bigcap_{l \in L} \langle l : \tau_l \rangle$, which in general is incorrect. The following Lemma 23 describes sufficient circumstances where this translation holds.

▶ **Lemma 23** ($\Delta_{\mathcal{L}} \cup \Delta_{\wp(\mathcal{L})}$ Translation Soundness)**.** *Let $L \subseteq \mathcal{L}$ be a non-empty set of labels, let $\rho \in \mathbb{T}_{\langle\rangle}$ be a type such that $[\![\rho]\!]$ is defined and let $\tau_l \in \mathbb{T}$ for $l \in L$ be types. If $w_L([\![\rho]\!])$ is inhabitable in $\Delta_{\mathcal{L}} \cup \Delta_{\wp(\mathcal{L})}$ then $\rho + \bigcap_{l \in L} \langle l : \tau_l \rangle = \rho \cap \bigcap_{l \in L} \langle l : \tau_l \rangle$.*

**Proof Sketch.** Since $w_L([\![\rho]\!])$ is inhabitable in $\Delta_{\mathcal{L}} \cup \Delta_{\wp(\mathcal{L})}$ we have $lbl(\rho) \cap L = \varnothing$. The result follows from Lemma 6 (2). ◀

Next, we show by the following Lemma 24 that inhabitation in $\Delta_{\mathcal{L}}$ is not too restrictive.

▶ **Lemma 24** ($\Delta_{\mathcal{L}} \cup \Delta_{\wp(\mathcal{L})}$ Translation Completeness)**.** *Let $L \subseteq \mathcal{L}$ be a non-empty set of labels, let $\tau_l \in \mathbb{T}$ for $l \in L$ be types, let $\rho \in \mathbb{T}_{\langle\rangle}$ be a type such that $[\![\rho]\!]$ is defined. There exists a type $\rho' \in \mathbb{T}_{\langle\rangle}$ such that $w_L([\![\rho']\!])$ is inhabitable in $\Delta_{\mathcal{L}} \cup \Delta_{\wp(\mathcal{L})}$ and $\rho' \cap \bigcap_{l \in L} \langle l : \tau_l \rangle \leq \rho + \bigcap_{l \in L} \langle l : \tau_l \rangle$.*

**Proof Sketch.** Given $\rho = \bigcap_{l' \in L'} \langle l' : \tau'_{l'} \rangle$ choose $\rho' = \bigcap_{l' \in L' \setminus L} \langle l' : \tau'_{l'} \rangle$ (resp. $\rho' = \langle\rangle$ if $L' \setminus L = \varnothing$). By Lemma 22 $w_L([\![\rho']\!])$ is inhabitable in $\Delta_{\mathcal{L}} \cup \Delta_{\wp(\mathcal{L})}$ and $\rho' \cap \bigcap_{l \in L} \langle l : \tau_l \rangle \leq \rho + \bigcap_{l \in L} \langle l : \tau_l \rangle$. ◀

Note that in Lemma 24 the type $\rho'$ can be chosen greater than $\rho$ only due to the non-monotonic properties of $+$.

## 4 Mixin Composition Synthesis by Type Inhabitation

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

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

$$\begin{aligned}\Delta^{\mathcal{C},\mathcal{M}}_{\mathcal{L}} =& \{C : [\![\sigma_C \to \rho_C]\!] \mid C \in \mathcal{C}\} \\ & \cup \{M : w_{L_M}(\alpha) \to ([\![\sigma_M]\!] \to \alpha \cap [\![\rho^1_M]\!]) \to ([\![\sigma_M]\!] \to \alpha \cap [\![\rho^2_M]\!]) \mid M \in \mathcal{M}\} \\ & \cup \Delta_{\mathcal{L}} \cup \{W_{L_M} \in \Delta_{\wp(\mathcal{L})} \mid M \in \mathcal{M}, |L_M| > 1\}\end{aligned}$$

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

Although types in $\Delta^{\mathcal{C},\mathcal{M}}_{\mathcal{L}}$ do not contain record-merge, we show by following Theorem 25 that types of mixin compositions in $\mathsf{BCL}(\to, \cap)$ are sound.

▶ **Theorem 25** (Soundness). *Let $M_1, \ldots, M_n \in \mathcal{M}$ be mixins, let $L_1, \ldots, L_n \subseteq \mathcal{L}$ be sets of labels, let $C \in \mathcal{C}$ be a class and let $\sigma \in \mathbb{T}, \rho \in \mathbb{T}_{\langle\rangle}$ be types such that $[\![\sigma \to \rho]\!]$ is defined.*
*If $\Delta_{\mathcal{L}}^{\mathcal{C},\mathcal{M}} \vdash_{BCL} C \gg (M_1\ W_{L_1}) \gg (M_2\ W_{L_2}) \gg \ldots \gg (M_n\ W_{L_n}) : [\![\sigma \to \rho]\!]$,*
*then $\varnothing \vdash_{\langle\rangle} C \gg M_1 \gg M_2 \gg \ldots \gg M_n : \sigma \to \rho$.*

**Proof Sketch.** Induction on $n$ proving $L_i = L_{M_i}$ followed by Lemma 23 and $(\to E)$. ◀

Complementary, we show by the following Theorem 26 that typing of mixin compositions in $\mathsf{BCL}(\to, \cap)$ is complete with respect to previously described typing in $\mathbb{T}$.

▶ **Theorem 26** (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.*
*If $\Gamma \vdash_{\langle\rangle} x_C \gg x_{M_1}^{\rho_1} \gg x_{M_2}^{\rho_2} \gg \ldots \gg x_{M_n}^{\rho_n} : \sigma \to \rho$,*
*then $\Delta_{\mathcal{L}}^{\mathcal{C},\mathcal{M}} \vdash_{BCL} C \gg (M_1\ W_{L_{M_1}}) \gg (M_2\ W_{L_{M_2}}) \gg \ldots \gg (M_n\ W_{L_{M_n}}) : [\![\sigma \to \rho]\!]$.*

**Proof Sketch.** Induction on $n$ choosing for each $x_M^\rho$ where $\rho = \bigcap_{l \in L} \langle l : \tau_l \rangle$ (resp. $\rho = \langle\rangle$) the substitution $S_i(\alpha) = \bigcap_{l \in L \setminus L_M} \langle\!\langle l(\tau_l) \rangle\!\rangle$ to type $M \in \Delta_{\mathcal{L}}^{\mathcal{C},\mathcal{M}}$ and using $(\to E)$ and Lemma 24. ◀

Coming back to our running example, we obtain

$$
\begin{aligned}
\Delta^{\{\text{Point}\},\{\text{Movable}\}}_{\{get,set,shift,move\}} = \{\ \text{Point} :\ & \text{Int} \to \langle\!\langle get(\text{Int}) \cap set(\text{Int} \to \text{Int}) \cap shift(\text{Int} \to \text{Int}) \rangle\!\rangle, \\
\text{Movable} :\ & w_{\{set,move\}}(\alpha) \\
& \to (\text{Int} \to \alpha \cap \langle\!\langle set(\text{Int} \to \text{Int}) \cap shift(\text{Int} \to \text{Int}) \rangle\!\rangle) \\
& \to (\text{Int} \to \alpha \cap \langle\!\langle set(\text{Int}) \cap move(\text{Int}) \rangle\!\rangle), \\
W_{\{get\}} :\ & w_{\{get\}}(\langle\!\langle set(\alpha_1) \cap shift(\alpha_2) \cap move(\alpha_3) \rangle\!\rangle), \\
W_{\{set\}} :\ & w_{\{set\}}(\langle\!\langle get(\alpha_1) \cap shift(\alpha_2) \cap move(\alpha_3) \rangle\!\rangle), \\
W_{\{shift\}} :\ & w_{\{shift\}}(\langle\!\langle get(\alpha_1) \cap set(\alpha_2) \cap move(\alpha_3) \rangle\!\rangle), \\
W_{\{move\}} :\ & w_{\{move\}}(\langle\!\langle get(\alpha_1) \cap set(\alpha_2) \cap shift(\alpha_3) \rangle\!\rangle), \\
W_{\{set,move\}} :\ & w_{\{set\}}(\alpha) \to w_{\{move\}}(\alpha) \to w_{\{set,move\}}(\alpha) \}
\end{aligned}
$$

We may ask inhabitation questions such as

$$\Delta^{\{\text{Point}\},\{\text{Movable}\}}_{\{get,set,shift,move\}} \vdash_{\text{BCL}} ? : [\![\text{Int} \to \langle shift : \text{Int} \to \text{Int}, move : \text{Int} \rangle]\!]$$

and obtain the combinatory term "Movable $W_{\{set,move\}}$ Point" as a synthesized result. From Theorem 25 we know

$$\varnothing \vdash_{\langle\rangle} \text{Movable Point} : \text{Int} \to \langle shift : \text{Int} \to \text{Int}, move : \text{Int} \rangle$$

On the other hand, if we want to inhabit $[\![\text{Int} \to \langle set : \text{Int} \to \text{Int}, move : \text{Int} \rangle]\!]$ we obtain no results. From Lemma 26 we know that, restricted to the previously described typing in $\mathbb{T}$, there is no mixin composition applied to a class with the resulting type $\text{Int} \to \langle set : \text{Int} \to \text{Int}, move : \text{Int} \rangle$.

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 $\mathcal{L}$ requires only to update combinators in $\Delta_{\mathcal{L}}$ leaving existing types of classes and mixins untouched. Third, adding a class/mixin to an existing repository is as simple as adding one typed combinator for the class/mixin and at most one logical combinator. Again, it is important that the existing combinators in the repository remain untouched. As an example, we add the following mixin MovableBy to $\Delta^{\{\text{Point}\},\{\text{Movable}\}}_{\{get,set,shift,move\}}$.

$$\begin{aligned}\text{MovableBy} = \lambda\mathsf{argClass}.\mathbf{Y}(\lambda\mathsf{myClass}.\lambda\mathsf{state}.\\ &\texttt{let}\ \mathsf{super} = \mathsf{argClass}\ \mathsf{state}\ \texttt{in}\\ &\texttt{let}\ \mathsf{self} = \mathsf{myClass}\ \mathsf{state}\ \texttt{in}\ \mathsf{super} \oplus \langle \mathit{move} = \mathsf{super}.\mathit{shift}\rangle)\end{aligned}$$

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

$$\varnothing \vdash_{\langle\rangle} \text{MovableBy}: (\text{Int} \to \rho \cap \langle \mathit{shift} : \text{Int} \to \text{Int}\rangle) \to (\text{Int} \to \rho + \langle \mathit{move} : \text{Int} \to \text{Int}\rangle)$$

We obtain the following extended repository

$$\begin{aligned}\Delta^{\{\text{Point}\},\{\text{Movable},\text{MovableBy}\}}_{\{\mathit{get},\mathit{set},\mathit{shift},\mathit{move}\}} = \Delta^{\{\text{Point}\},\{\text{Movable}\}}_{\{\mathit{get},\mathit{set},\mathit{shift},\mathit{move}\}} \cup \{&\text{MovableBy} : w_{\{\mathit{move}\}}(\alpha)\\ &\to (\text{Int} \to \alpha \cap \langle\!\langle \mathit{shift}(\text{Int} \to \text{Int})\rangle\!\rangle) \to (\text{Int} \to \alpha \cap \langle\!\langle \mathit{move}(\text{Int} \to \text{Int})\rangle\!\rangle)\}\end{aligned}$$

Asking the inhabitation question

$$\Delta^{\{\text{Point}\},\{\text{Movable},\text{MovableBy}\}}_{\{\mathit{get},\mathit{set},\mathit{shift},\mathit{move}\}} \vdash_{\text{BCL}} ? : [\![\text{Int} \to \langle \mathit{set} : \text{Int}, \mathit{move} : \text{Int} \to \text{Int}\rangle]\!]$$

synthesizes "Point $\gg$ (Movable $W_{\{\mathit{set},\mathit{move}\}}$) $\gg$ (MovableBy $W_{\{\mathit{move}\}}$)". 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 may make multiple applications of a single mixin meaningful.

## 5 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 $\lambda$-calculus with records and $\oplus$ typed by intersection types with records and $+$. It is capable of modeling classes as 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 well-studied 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 25) and partially complete (Theorem 26). A notable feature is the encoding of negative information (the absence of labels). It exploits the logic programming capabilities of inhabitation, by adding sets of combinators serving as witnesses for the non-presence of labels. In section 4 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. The partial completeness result indicates a more expressive power of type constructors compared to records. Another direction of future work is to extend types of mixins and classes by semantic as well as modal types [19], a development initiated in [5]. 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 higher semantic accuracy.

**Acknowledgments.** The authors would like to thank the anonymous reviewers for their valuable comments.

## References


1 Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.
2 Steffen Van Bakel. Strict intersection types for the lambda calculus. *ACM Comput. Surv.*, 43(3):20:1–20:49, April 2011.
3 H. Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.
4 H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A Filter Lambda Model and the Completeness of Type Assignment. *Journal of Symbolic Logic*, 48(4):931–940, 1983.
5 Jan Bessai, Boris Düdder, Andrej Dudenhefer, and Moritz Martens. Delegation-based mixin composition synthesis. `http://www-seal.cs.tu-dortmund.de/seal/downloads/papers/paper-ITRS2014.pdf`, 2014.
6 Jan Bessai, Boris Düdder, Andrej Dudenhefner, Tzu-Chun Chen, and Ugo de'Liguoro. Typing classes and mixins with intersection types. *arXiv preprint arXiv:1503.04911*, 2015.
7 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, pages 26–40, 2014.
8 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.
9 Gilad Bracha. *The Programming Language JIGSAW: Mixins, Modularity and Multiple Inheritance*. PhD thesis, Univeristy of Utha, 1992.
10 Gilad Bracha and William R. Cook. Mixin-based inheritance. In *OOPSLA/ECOOP*, pages 303–311, 1990.
11 Kim B. Bruce. *Foundations of Object-Oriented Languages – Types and Semantics*. MIT Press, 2002.
12 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.
13 Luca Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types*, volume 173, pages 51–67, 1984.
14 Adriana B. Compagnoni and Benjamin C. Pierce. Higher-order intersection types and multiple inheritance. *Mathematical Structures in Computer Science*, 6(5):469–501, 1996.
15 William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *POPL'90*, pages 125–135. ACM Press, 1990.
16 Ugo de'Liguoro. Characterizing convergent terms in object calculi via intersection types. In *TLCA*, pages 315–328, 2001.
17 Ugo de'Liguoro and Tzu chun Chen. Semantic Types for Classes and Mixins. `http://www.di.unito.it/~deligu/papers/UdLTC14.pdf`, 2014.
18 Boris Düdder. *Automatic Synthesis of Component & Connector-Software Architectures with Bounded Combinatory Logic*. Dissertation, TU Dortmund, 2014.
19 Boris Düdder, Moritz Martens, and Jakob Rehof. Staged composition synthesis. In *ESOP*, volume 8410 of *Lecture Notes in Computer Science*, pages 67–86, 2014.
20 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.
21 Standard Ecma. ECMA-262 ECMAScript Language Specification, 2011.
22 Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete Completion using Types and Weights. *SIGPLAN Notices*, 48(6):27–38, 2013.
23 Oleg Kiselyov and Ralf Lämmel. Haskell's overlooked object system. *CoRR*, abs/cs/0509027, 2005.
24 Martin Odersky and Matthias Zenger. Scalable component abstractions. In *OOPSLA*, pages 41–57. ACM, 2005.

**25** Benjamin C. Pierce. Intersection types and bounded polymorphism. *Mathematical Structures in Computer Science*, 7(2):129–193, 1997.

**26** Didier Rémy. Typing record concatenation for free. In *POPL'92*, pages 166–176, 1992.

**27** Mitchell Wand. Type inference for record concatenation and multiple inheritance. *Inf. Comput.*, 93(1):1–15, 1991.