Typing Classes and Mixins with Intersection Types

Jan Bessai   Boris Düdder   Andrej Dudenhefner

Technical University of Dortmund, Germany

`{jan.bessai, boris.duedder, Andrej.dudenhefner}@cs.tu-dortmund.de`

Tzu-Chun Chen   Ugo de'Liguoro

Technical University of Darmstadt, Germany   University of Torino, Italy

`tcchen@rbg.informatik.tu-darmstadt.de`   `ugo.deliguoro@unito.it`

We study an assignment system of intersection types for a lambda-calculus with records and a record-merge operator, where types are preserved both under subject reduction and expansion. The calculus is expressive enough to naturally represent mixins as functions over recursively defined classes, whose fixed points, the objects, are recursive records. In spite of the double recursion that is involved in their definition, classes and mixins can be meaningfully typed without resorting to neither recursive nor F-bounded polymorphic types.

We then adapt mixin construct and composition to Java and C#, relying solely on existing features in such a way that the resulting code remains typable in the respective type systems. We exhibit some example code, and study its typings in the intersection type system via interpretation into the lambda-calculus with records we have proposed.

1 Introduction

Starting with Cardelli's pioneering work [11], 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, 9, 17]).

Here we focus on object composition based on mixins, a study initiated in [7, 8] and based on the recursive record model and F-bounded polymorphic types [10, 12]. In the object-oriented paradigm, mixins have been introduced as an alternative construct w.r.t. class inheritance to avoid semantic ambiguities caused by multiple inheritance. Together with abstract classes and traits, mixins can be considered as an advanced construct to obtain flexible implementation of module libraries and to enhance code reusability; many popular programming languages miss native support for mixins, but they are object of intensive study and research (e.g. [6, 18]). Scala supports mixins, which recently have been used to model feature oriented programming [19]. Dynamic languages, such as JavaScript, offer multiple ways to compensate for the lack of mixins [20].

In [22] a new technique for synthesizing programs from components has been based on combinatory logic and intersection types. Aiming at a future extension of the approach to Object-Oriented (OO) libraries, we study an assignment system of intersection types for a λ-calculus with records and a record-merge operator, building over and extending the systems proposed in [14, 3, 23].

In section 2, we propose a type-free λ-calculus with record and record-merge operator, and define a Curry-style type assignment system. Such a system is polymorphic in the sense that any term has infinitely many types, including the trivial type ω. In doing that, we look for an extension of the system in [4], and in particular we expect that typing is invariant under subject reduction and expansion. In this respect the main difficulty consists in the typing of Bracha-Cook's merge operator, which embodies

at the same time record extension and field overriding. To solve such a problem we distinguish among ordinary λ-terms and record terms, allowing just record terms to be merged to arbitrary terms. In section 3, we then study the typings in our system of the encoding of classes and mixins into the λ-calculus proposed in [12, 8], identifying a subset of intersection types that can be seen as meaningful types for such entities. We then consider (Section 3) the problem of using the proposed system in the case of the actual programming languages Java and C#, that do not embody mixins in their syntax; nevertheless programs that typecheck w.r.t. the type system of the respecitive programming language can be seen as the product of the fully applied mixin compositions. We face these challenges by using delegation instead of inheritance. This gives rise to a design pattern that can be interpreted in the λ-calculus with records, such that the typings we have devised for classes and mixins in the theoretical model apply to actual programming languages.

This work evolves from the contributions [5, 15] to the workshop ITRS'14. Joining the contributions from both previous papers, this work contributes a way to model classes and mixins typable in the intersection type system. The flexibility of this type system allows to augment types by semantic features, required for a future extension to automatically synthesize OO code [5]. Furthermore we present a correspondence between the model and actual programming languages.

2   Intersection types for a λ-calculus with records and merge

We consider a type-free λ-calculus of records equipped with a merge operator. The syntax of Λ_R is defined by the following grammar:

$$\text{Term:}\quad M, N ::= x \mid (\lambda x.M) \mid (MN) \mid R \mid (M.a) \mid (M \oplus R) \qquad \text{Record:}\quad R ::= \langle a_i = M_i \mid i \in I \rangle$$

where $x \in \mathbf{Var}$ and $a \in \mathbf{Label}$ range over denumerably many variables and labels respectively. In $R \equiv \langle a_i = M_i \mid i \in I \rangle$ (writing $\equiv$ for syntactic identity) the set I is finite; we define $lbl(R) = \{a_i \mid i \in I\}$ to denote the set of labels of R and we use the abbreviation $a = M \in R$ for $a \equiv a_i$ & $M \equiv M_i$ for some $i \in I$.

Definition 2.1 (Λ_R Reduction)

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

The last reduction makes it apparent that $\oplus$ is record merge from [7], which is written $\leftarrow_r$, and the analogous operator in [8], although in our notation $M \oplus R$ the record arguments are listed in the opposite order than in [8], namely whenever M reduces to a record R' the fields in R prevail over those in R' having the same labels.

Definition 2.2 (Intersection types for Λ_R)

$$\sigma, \tau ::= \alpha \mid t \mid \omega \mid \sigma \to \tau \mid \sigma \cap \tau \mid \langle a : \sigma \rangle.$$

Here α ranges over a countable set of type variables, t are type constants for ground types. e.g. $\mathsf{int}, \mathsf{bool}$, or for (atomic) abstract properties, like $\mathsf{EvenInt}$ or $\mathsf{CelsiusDeg}$; ω is the universal type; $\sigma \to \tau$ and $\sigma \cap \tau$ are the arrow and the intersection types respectively. Finally $\langle a : \sigma \rangle$ is the type of records having a label a with value of type σ.

Definition 2.3 (Type Inclusion) Over types we consider the preorder $\leq$ extending EATS axioms (see e.g. [2], Def. 3.3.1) by the last two axioms:

$$\sigma \leq \omega, \qquad \omega \leq \omega \to \omega, \qquad \sigma \cap \tau \leq \sigma, \qquad \sigma \cap \tau \leq \tau, \qquad (\sigma \to \tau_1) \cap (\sigma \to \tau_2) \leq \sigma \to \tau_1 \cap \tau_2,$$

$$\sigma \leq \tau_1 \ \& \ \sigma \leq \tau_2 \Rightarrow \sigma \leq \tau_1 \cap \tau_2, \qquad \sigma_2 \leq \sigma_1 \ \& \ \tau_1 \leq \tau_2 \Rightarrow \sigma_1 \to \tau_1 \leq \sigma_2 \to \tau_2,$$

$$\sigma \leq \tau \Rightarrow \langle a : \sigma \rangle \leq \langle a : \tau \rangle, \qquad \langle a : \sigma \rangle \cap \langle a : \tau \rangle \leq \langle a : \sigma \cap \tau \rangle.$$

Note that we have $\langle a : \sigma \rangle \cap \langle a : \tau \rangle = \langle a : \sigma \cap \tau \rangle$ where $=$ is the symmetric closure of $\leq$. Also we have $\langle a : \omega \rangle \neq \omega$ for any a.

Definition 2.4 (Type Assignment) The assignment system adds the rules $(sel), (rec), (\oplus_l), (\oplus_r)$ to the intersection type assignment [4] (but with the $\leq$ relation from Def. 2.3):

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

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

$$\frac{\Gamma \vdash M : \langle a : \sigma \rangle}{\Gamma \vdash M.a : \sigma} (sel) \qquad \frac{\Gamma \vdash M : \sigma \quad a = M \in R}{\Gamma \vdash R : \langle a : \sigma \rangle} (rec)$$

$$\frac{\Gamma \vdash M : \langle a : \sigma \rangle \quad a \notin lbl(R)}{\Gamma \vdash M \oplus R : \langle a : \sigma \rangle} (\oplus_l) \qquad \frac{\Gamma \vdash R : \langle a : \sigma \rangle}{\Gamma \vdash M \oplus R : \langle a : \sigma \rangle} (\oplus_r)$$

Record types express just partial information about the fields in a record, as it is apparent from rule (rec); typings of more than one field in a record are obtained by means of intersection (see below). Rule (sel) is the expected one. Rules $(\oplus_l)$ and $(\oplus_r)$ are not symmetric because of the restriction $a \notin lbl(R)$ in the last rule. Such a restriction is essential for the soundnes of the system, in particular for subject reduction to hold. E.g. we have $\langle a = M \rangle \oplus \langle a = N \rangle \longrightarrow \langle a = N \rangle$; now supposing that $M : \sigma, N : \tau$ where $\sigma \neq \tau$ and that σ cannot be assigned to N, without the restriction we could type $\langle a = M \rangle \oplus \langle a = N \rangle$ by $\langle a : \sigma \rangle$ and even by $\langle a : \sigma \rangle \cap \langle a : \tau \rangle = \langle a : \sigma \cap \tau \rangle$, but we couldn't type $\langle a = N \rangle$ by neither of these types.

Let us abbreviate $\langle a_i : \sigma_i \mid i \in I \rangle \equiv \bigcap_{i \in I} \langle a_i : \sigma_i \rangle$ where we assume the a_i to be pairwise distinct. By abusing terminology we call it a *record type*. Then the following rule is admissible:

$$\frac{\sigma_j \leq \tau_j \qquad \forall j \in J \subseteq I}{\langle a_i : \sigma_i \mid i \in I \rangle \leq \langle a_j : \tau_j \mid j \in J \rangle}$$

which is record subtyping in width and depth. Further we have the following admissible typing rules:

$$\frac{\Gamma \vdash N_i : \sigma_i \qquad \forall i \in I \subseteq J}{\Gamma \vdash \langle a_i = N_i \mid i \in J \rangle : \langle a_i : \sigma_i \mid i \in I \rangle} \qquad \frac{\Gamma \vdash M : \langle a_i : \sigma_i \mid i \in I \rangle \qquad \Gamma \vdash R : \langle a_j : \tau_j \mid j \in J \rangle}{\Gamma \vdash M \oplus R : \langle a_i : \sigma_i, a_j : \tau_j \mid i \in I \setminus J, j \in J \rangle}$$

We exemplify the typings of functions from records to records, that are at the earth of the encoding of mixins. Let $M_{R_i} \equiv \lambda x.(x \oplus R_i)$, where $R_1 \equiv \langle a = N_1 \rangle$, $R_2 \equiv \langle b = N_2 \rangle$ and $R_3 \equiv \langle a = N_3 \rangle$. For simplicity let us assume $x \notin fv(R_i)$ for $i = 1, 2, 3$; then

$$
\cfrac{\Gamma, x : \omega \vdash x : \omega \qquad \cfrac{\Gamma, x : \omega \vdash N_1 : \sigma_1}{\Gamma, x : \omega \vdash \langle a = N_1 \rangle : \langle a : \sigma_1 \rangle}}{\cfrac{\Gamma, x : \omega \vdash x \oplus \langle a = N_1 \rangle : \langle a : \sigma_1 \rangle}{\Gamma \vdash M_{R_1} \equiv \lambda x.(x \oplus \langle a = N_1 \rangle) : \omega \to \langle a : \sigma_1 \rangle}}
$$

and also

$$
\cfrac{\cfrac{\Gamma, x : \langle b : \sigma_2 \rangle \vdash x : \langle b : \sigma_2 \rangle \qquad b \notin lbl(\langle a = N_1 \rangle)}{\Gamma, x : \langle b : \sigma_2 \rangle \vdash x \oplus \langle a = N_1 \rangle : \langle b : \sigma_2 \rangle}}{\Gamma \vdash M_{R_1} \equiv \lambda x.(x \oplus \langle a = N_1 \rangle) : \langle b : \sigma_2 \rangle \to \langle b : \sigma_2 \rangle}
$$

therefore by $\omega \to \langle a : \sigma_1 \rangle \leq \langle b : \sigma_2 \rangle \to \langle a : \sigma_1 \rangle$ we have

$$
\Gamma \vdash M_{R_1} : (\langle b : \sigma_2 \rangle \to \langle a : \sigma_1 \rangle) \cap (\langle b : \sigma_2 \rangle \to \langle b : \sigma_2 \rangle)
$$

that is $\Gamma \vdash M_{R_1} : \langle b : \sigma_2 \rangle \to \langle a : \sigma_1, b : \sigma_2 \rangle$. Similarly for M_{R_2}, assuming that $\Gamma \vdash N_2 : \sigma_2$ we have $\Gamma \vdash M_{R_2} : \omega \to \langle b : \sigma_2 \rangle$ so that

$$
\cfrac{\Gamma \vdash M_{R_2} : \omega \to \langle b : \sigma_2 \rangle \qquad \Gamma \vdash M_{R_1} : \langle b : \sigma_2 \rangle \to \langle a : \sigma_1, b : \sigma_2 \rangle}{\Gamma \vdash M_{R_1} \circ M_{R_2} : \omega \to \langle a : \sigma_1, b : \sigma_2 \rangle}
$$

where $M \circ N \equiv \mathbf{B} M N =_\beta \lambda x.M(N x)$. If $\Gamma \vdash N_1 : \sigma_1$, $\Gamma \vdash N_3 : \sigma_3$ but $\Gamma \nvdash N_1 : \sigma_3$ we have

$$
\Gamma \vdash M_{R_3} : \omega \to \langle a : \sigma_3 \rangle \text{ and } \Gamma \vdash M_{R_1} : \omega \to \langle a : \sigma_1 \rangle \leq \langle a : \sigma_3 \rangle \to \langle a : \sigma_1 \rangle
$$

so that $\Gamma \vdash M_{R_1} \circ M_{R_3} : \omega \to \langle a : \sigma_1 \rangle$ but $\Gamma \nvdash M_{R_1} \circ M_{R_3} : \omega \to \langle a : \sigma_3 \rangle$.

Let $M = N$ be the convertibility relation generated by $\longrightarrow$; then we can prove[1]:

Theorem 2.5 (Type Invariance for Λ_R) For any $M, N \in \Lambda_R$,

$$
\Gamma \vdash M : \sigma \, \& \, M = N \ \Rightarrow \ \Gamma \vdash N : \sigma.
$$

Remark 2.6 By definition $R \oplus x \notin \Lambda_R$ and hence $\lambda x.(R \oplus x) \notin \Lambda_R$. If we admit record terms of the shape $R \oplus x$ then their typing would be highly problematic. The problem is that to apply rule $(\oplus_l)$ when typing $R \oplus x$ we have to extend lbl to variables. If we set $lbl(x) = \emptyset$ the side condition $a \notin \emptyset$ of rule $(\oplus_l)$ is always satisfied, and we could derive for $R \oplus x$ all the types of R; hence for any σ:

$$
\cfrac{\cfrac{\Gamma, x : \sigma \vdash R : \langle a : \tau \rangle \qquad a \notin lbl(x) = \emptyset}{\Gamma, x : \sigma \vdash R \oplus x : \langle a : \tau \rangle}}{\Gamma \vdash \lambda x.(R \oplus x) : \sigma \to \langle a : \tau \rangle}
$$

Let $\sigma \equiv \langle a : \rho \rangle$ and N such that $\Gamma \vdash N : \rho$ and $\Gamma \nvdash N : \tau$. Then $\Gamma \vdash (\lambda x.(R \oplus x))\langle a = N \rangle : \langle a : \tau \rangle$, while $\Gamma \nvdash R \oplus \langle a = N \rangle : \langle a : \tau \rangle$, contradicting subject reduction.

One could think to change the definition of $lbl(x)$ to $lbl'(x) = \mathbf{Label}$, since the variable x can be replaced by β-reduction with any record R', whose set of labels we cannot predict. This time the side condition $a \notin \mathbf{Label}$ never holds true, so that $(\lambda x.(R \oplus x))\langle a = N \rangle$ would have less types than $R \oplus \langle a = N \rangle$, breaking subject expansion.

---

[1]Proof available in the appendix of `http://www-seal.cs.tu-dortmund.de/seal/downloads/papers/paper-ITRS2014-postproceedings.pdf`

3 Class and mixin interpretation and typings

Here we study the typings of the interpretations of classes and mixins based on the recursive-record model, following [8]. We proceed by steps, considering non-recursive classes and mixins first; then we move to the respective recursive versions.

Non-recursive classes. Under recursive record interpretations a *non-recursive class* is a function:

$$C \overset{\Delta}{=} \lambda \vec{x} \, \lambda \, \text{self.} \langle a_1 = M_1, \dots, a_k = M_k \rangle$$

where $\vec{x}$ are the parameters for the initial values (the *state*). Class C is pre-instantiated when applied to a tuple of values $\vec{v}$ of the same length as $\vec{x}$: $C \vec{v} = \lambda \, \text{self.} \langle a_1 = M_1[\vec{v}/\vec{x}], \dots, a_k = M_k[\vec{v}/\vec{x}] \rangle$. *Objects*, or *class instances*, are fixed points of pre-instantiated classes:

$O \overset{\Delta}{=} \mathbf{Y}(C \vec{v}) = \langle a_1 = M_1[\vec{v}/\vec{x}, O/\text{self}], \dots, a_k = M_k[\vec{v}/\vec{x}, O/\text{self}] \rangle$ where $\mathbf{Y} \overset{\Delta}{=} \lambda f.(\lambda x f(xx))(\lambda x f(xx))$ is Curry's fixed point combinator and equality is conversion. Then sending a message a to the object O is defined as record selection: $O \Leftarrow a \overset{\Delta}{=} O.a$. Indeed, for $i \in \{1, \dots, k\}$:

$$O \Leftarrow a_i = (\langle a_1 = M_1[\vec{v}/\vec{x}, O/\text{self}], \dots, a_k = M_k[\vec{v}/\vec{x}, O/\text{self}] \rangle).a_i = M_i[\vec{v}/\vec{x}, O/\text{self}].$$

For example let us consider the class: $\text{Point} \overset{\Delta}{=} \lambda x \, \lambda \, \text{self.} \langle \mathsf{X} = x, \ \text{get} = \text{self.X} \rangle$, such that X is the field holding the position of a one dimensional point (the object state), get returns the position of its point. Looking at the typings of $\text{Point}(3) = \lambda \, \text{self.} \langle \mathsf{X} = 3, \ \text{get} = \text{self.X} \rangle$, we have:

$$\frac{\dfrac{\text{self} : \omega \vdash 3 : \text{int}}{\text{self} : \omega \vdash \langle \mathsf{X} = 3, \ \text{get} = \text{self.X} \rangle : \langle \mathsf{X} : \text{int} \rangle}}{\vdash \lambda \, \text{self.} \langle \mathsf{X} = 3, \ \text{get} = \text{self.X} \rangle : \omega \to \langle \mathsf{X} : \text{int} \rangle}$$

but also, setting $\sigma_1 \overset{\Delta}{=} \langle \mathsf{X} : \text{int} \rangle$:

$$\frac{\dfrac{\text{self} : \sigma_1 \vdash 3 : \text{int} \qquad \text{self} : \sigma_1 \vdash \text{self.X} : \text{int}}{\text{self} : \sigma_1 \vdash \langle \mathsf{X} = 3, \ \text{get} = \text{self.X} \rangle : \langle \mathsf{X} : \text{int}, \ \text{get} : \text{int} \rangle}}{\vdash \lambda \, \text{self.} \langle \mathsf{X} = 3, \ \text{get} = \text{self.X} \rangle : \sigma_1 \to \langle \mathsf{X} : \text{int}, \ \text{get} : \text{int} \rangle}$$

so that, putting $\sigma_2 \overset{\Delta}{=} \langle \mathsf{X} : \text{int}, \ \text{get} : \text{int} \rangle$, we have

$$\text{Point} : \text{int} \to (\omega \to \sigma_1) \cap (\sigma_1 \to \sigma_2)$$

and hence $\text{Point}(3) : (\omega \to \sigma_1) \cap (\sigma_1 \to \sigma_2)$. It is folklore (see e.g. [16], Sec. 2) that with intersection types we can type $\mathbf{Y} \equiv \lambda f.(\lambda x. f(xx))(\lambda x. f(xx))$ by $(\omega \to \tau_1) \cap (\tau_1 \to \tau_2) \cap \cdots \cap (\tau_{n-1} \to \tau_n) \to \tau_n$ for any n and $\tau_1, \dots, \tau_n$; hence we conclude that

$$\text{point} \overset{\Delta}{=} \mathbf{Y}(\text{Point}(3)) = \langle \mathsf{X} = 3, \ \text{get} = \text{point.X} \rangle : \langle \mathsf{X} : \text{int}, \ \text{get} : \text{int} \rangle$$

and therefore both $(\text{point} \Leftarrow \mathsf{X}) : \text{int}$ and $(\text{point} \Leftarrow \text{get}) : \text{int}$. Observe that neither typings of the class Point nor of the object point are recursive types.

Let $\vec{t}$ be a vector of ground types and $\sigma_1, \ldots, \sigma_n$ be record types, i.e. the types of *class instances*; then by generalizing from the above example we define:

Non-recursive class type: $\kappa ::= \vec{t} \to (\omega \to \sigma_1) \cap \bigcap_{i=1}^{n-1} (\sigma_i \to \sigma_{i+1})$.

Note that $\kappa \leq \vec{t} \to (\sigma_i \to \sigma_{i+1})$ for any $i = 1, \ldots, n-1$, but this typing would be not enough for typing objects obtained as fixed points of pre-instantiated classes.

Non-recursive mixins are functions from non-recursive classes to non-recursive classes:

$$M \stackrel{\Delta}{=} \lambda\,\mathsf{super}\,\lambda\vec{x}\,\lambda\mathsf{self}.\,\mathbf{Y}(\mathsf{super}\,\vec{y}) \oplus \langle a_1 = M_1, \ldots, a_k = M_k \rangle, \quad \vec{y} \subseteq \vec{x}.$$

Note that both self and super may occur free in the M_i. Let's consider the example of a mixin adding a second dimension to one dimension points of class Point:

$$\mathsf{Point2D} \stackrel{\Delta}{=} \lambda\,\mathsf{super}\,\lambda x\,\lambda y\,\lambda\mathsf{self}.\,\mathbf{Y}(\mathsf{super}\,x) \oplus \langle \mathsf{Y} = y, \mathsf{get} = (c.\mathsf{X}, \mathsf{self.Y}) \rangle$$

The apparently equivalent term $\lambda\,\mathsf{super}\,\lambda x\,\lambda y\,\lambda\mathsf{self}.\,\mathbf{Y}(\mathsf{super}\,x) \oplus \langle \mathsf{Y} = y, \mathsf{get} = (x, \mathsf{self.Y}) \rangle$ is coarser than Point2D, because it ignores the usage of the state variable x by the superclass super, which is a parameter. By simple computations we obtain:

$$
\begin{aligned}
\mathsf{Point2D}(\mathsf{Point}) \quad &= \quad \lambda x\,\lambda y\,\lambda\mathsf{self}.\,\mathbf{Y}(\mathsf{Point}\,x) \oplus \langle \mathsf{Y} = y, \mathsf{get} = ((\mathbf{Y}(\mathsf{Point}\,x)).\mathsf{X}, \mathsf{self.Y}) \rangle \\
&= \quad \lambda x\,\lambda y\,\lambda\mathsf{self}.\,\langle \mathsf{X} = x, \mathsf{get} = \ldots \rangle \oplus \langle \mathsf{Y} = y, \mathsf{get} = (x, \mathsf{self.Y}) \rangle \\
&= \quad \lambda x\,\lambda y\,\lambda\mathsf{self}.\,\langle \mathsf{X} = x, \mathsf{Y} = y, \mathsf{get} = (x, \mathsf{self.Y}) \rangle
\end{aligned}
$$

By adding product types (for exemplification purposes) we easily derive that $\mathsf{Point2D}(\mathsf{Point})$ has type:

$$\mathsf{int} \to \mathsf{int} \to (\omega \to \langle \mathsf{X} : \mathsf{int}, \mathsf{Y} : \mathsf{int} \rangle) \cap (\langle \mathsf{X} : \mathsf{int}, \mathsf{Y} : \mathsf{int} \rangle \to \langle \mathsf{X} : \mathsf{int}, \mathsf{Y} : \mathsf{int}, \mathsf{get} : \mathsf{int} \times \mathsf{int} \rangle)$$

which is a non-recursive class type. In general we conclude that a non-recursive mixin has type $\kappa_1 \to \kappa_2$, where κ_1, κ_2 are non-recursive class types.

Recursive classes and mixins. Recursive classes and mixins are necessary whenever the resulting class is used to generate a new object of the same class (see the example with movable points below). Recursive classes have an extra parameter myClass:

$$C' \stackrel{\Delta}{=} \lambda\,\mathsf{myClass}\,\lambda\vec{x}\,\lambda\mathsf{self}.\langle a_1 = M_1, \ldots, a_k = M_k \rangle.$$

The fixed point of a recursive class is a class; so to instantiate a recursive class to an object, a double fixed point is needed: $O' \stackrel{\Delta}{=} \mathbf{Y}((\mathbf{Y}C')\vec{v})$. Because of this, types of recursive classes have the shape:

Recursive class type: $\kappa' ::= (\omega \to \kappa_1) \cap \bigcap_{i=1}^{n-1} (\kappa_i \to \kappa_{i+1})$

where $\kappa_1, \ldots, \kappa_n$ are non-recursive class types. We observe that all κ_i begin by the same the same $\vec{t}$ and include record types $\sigma_{i,1}, \ldots, \sigma_{i,n_i}$ that can be choosed so that $\sigma_{i,j} \geq \sigma_{i,j+1}$, to form a descending chain. The object O' is a record, that can be merged in a mixin with a record $\Delta(\mathsf{super})(\mathsf{myClass})(\vec{x})(\mathsf{self}) \equiv \langle a_1 = M_1, \ldots, a_k = M_k \rangle$ of added/overridden members:

$$M' \stackrel{\Delta}{=} \lambda\,\mathsf{super}\,\lambda\,\mathsf{myClass}\,\lambda\vec{x}\,\lambda\mathsf{self}.\,\mathbf{Y}((\mathbf{Y}\,\mathsf{super})\,\vec{y}) \oplus \Delta(\mathsf{super})(\mathsf{myClass})(\vec{x})(\mathsf{self}), \quad \vec{y} \subseteq \vec{x}.$$

As an example, consider the following mixin:

$$\text{Movable} \triangleq \lambda\,\text{super}\,\lambda\,\text{myClass}\,\lambda x\,\lambda\,\text{self}.\ \mathbf{Y}((\mathbf{Y}\,\text{super})\,x) \oplus \langle \text{move} = \lambda dx.\ \mathbf{Y}((\mathbf{Y}\,\text{myClass})\,(\text{self}.X + dx))\rangle$$

Now let $\text{RecPoint} \triangleq \lambda\,\text{myClass}.\ \text{Point}$ be a (vacuously) recursive version of Point, and consider the object $\text{movable} \triangleq \mathbf{Y}(\mathbf{Y}\,(\text{Movable}\,\text{RecPoint})\,3)$. Then for example we have:

$$
\begin{aligned}
&(\text{movable} \Leftarrow \text{move})(4)\\
=\ &((\mathbf{Y}(\mathbf{Y}\,\text{RecPoint}\,3) \oplus\\
&\qquad \langle \text{move} = \lambda dx.\ \mathbf{Y}(\mathbf{Y}\,(\text{Movable}\,\text{RecPoint})\,(\text{movable}.X + dx))\rangle)) \Leftarrow \text{move})(4)\\
=\ &(\langle X = 3, \text{get} = \dots, \text{move} = \lambda dx.\ \mathbf{Y}(\mathbf{Y}\,(\text{Movable}\,\text{RecPoint})\,(\text{movable}.X + dx))\rangle.\text{move})(4)\\
=\ &\mathbf{Y}(\mathbf{Y}\,(\text{Movable}\,\text{RecPoint})\,(\text{movable}.X + 4))\\
=\ &\mathbf{Y}((\mathbf{Y}\,(\text{Movable}\,\text{RecPoint}))\,(3 + 4)) \qquad\qquad\qquad \text{since}\quad \text{movable}.X = 3\\
=\ &\langle X = 7, \text{get} = \dots, \text{move} = \dots\rangle.
\end{aligned}
$$

To see a possible typing of Movable, let $\kappa_1 \triangleq \text{int} \to (\omega \to \sigma_1) \cap (\sigma_1 \to \sigma_2)$, which we know to be a type of the class Point; then $\omega \to \kappa_1$ is a type of RecPoint. Now

$$
\frac{
\dfrac{x : \text{int} \vdash \mathbf{Y}\,\text{RecPoint} : \kappa_1 \qquad x : \text{int} \vdash x : \text{int}}{x : \text{int} \vdash \mathbf{Y}\,\text{RecPoint}\,x : (\omega \to \sigma_1) \cap (\sigma_1 \to \sigma_2)}
}{
x : \text{int} \vdash \mathbf{Y}(\mathbf{Y}\,\text{RecPoint}\,x) : \sigma_2 = \langle X : \text{int}, \text{get} : \text{int}\rangle
}
$$

and therefore, with X and get being different from move, and super being bound to RecPoint in movable we have:

$$\text{super} : \omega \to \kappa_1, x : \text{int} \vdash \mathbf{Y}(\mathbf{Y}\,\text{super}\,x) \oplus \langle \text{move} = \lambda dx.\ \mathbf{Y}((\mathbf{Y}\,\text{myClass})\,(\text{self}.X + dx)\rangle : \sigma_2.$$

Let $\Gamma_2 = \{\text{super} : \omega \to \kappa_1, \text{myClass} : \omega \to \kappa_1, x : \text{int}, \text{self} : \sigma_1\}$; then, since we have:

$$
\frac{\Gamma_2, dx : \text{int} \vdash \text{self}.X : \text{int} \qquad \Gamma_2, dx : \text{int} \vdash dx : \text{int}}{\Gamma_2, dx : \text{int} \vdash \text{self}.X + dx : \text{int}}
$$

it follows that:

$$
\frac{
\dfrac{
\dfrac{\Gamma_2, dx : \text{int} \vdash \mathbf{Y}\,\text{myClass} : \kappa_1 = \text{int} \to (\omega \to \sigma_1) \cap (\sigma_1 \to \sigma_2)}{\Gamma_2, dx : \text{int} \vdash (\mathbf{Y}\,\text{myClass})\,(\text{self}.X + dx) : (\omega \to \sigma_1) \cap (\sigma_1 \to \sigma_2)}
}{\Gamma_2, dx : \text{int} \vdash \mathbf{Y}((\mathbf{Y}\,\text{myClass})\,(\text{self}.X + dx)) : \sigma_2}
}{
\Gamma_2, \vdash \lambda dx.\,\mathbf{Y}((\mathbf{Y}\,\text{myClass})\,(\text{self}.X + dx)) : \text{int} \to \sigma_2
}
$$

and we conclude that:

$$\Gamma_2 \vdash \mathbf{Y}(\mathbf{Y}\,\text{super}\,x) \oplus \langle \text{move} = \lambda dx.\ \mathbf{Y}((\mathbf{Y}\,\text{myClass})\,(\text{self}.X + dx)\rangle : \sigma_2 \cap \langle \text{move} : \text{int} \to \sigma_2\rangle,$$

so that by taking $\kappa_2 \triangleq \text{int} \to (\sigma_2 \cap \langle \text{move} : \text{int} \to \sigma_2\rangle)$ we can derive the typing:

$$\text{Movable} : (\omega \to \kappa_1) \to (\omega \to \kappa_1) \cap (\kappa_1 \to \kappa_2),$$

that is of the form $\kappa_1' \to \kappa_2'$ where κ_1', κ_2' are recursive class types.

Mixin composition. Since mixins are just functions from classes to classes, the natural way to compose them is by functional composition. To make it easier to compare with definitions in [8], we compute their composition up to conversion. For $i = 1, 2$ let

$$M_i \stackrel{\Delta}{=} \lambda\,\mathsf{super}_i\,\lambda\,\mathsf{self}_i.\ \mathbf{Y}(\mathsf{super}_i) \oplus \Delta_i(\mathsf{super}_i)(\mathsf{self}_i)$$

be a pair of (non-recursive) mixins, where $\Delta_i(\mathsf{super}_i)(\mathsf{self}_i)$ is the record of added/overridden methods, and we forget about the state for simplicity. Then, for a suitably non-recursive class C we have:

$$
\begin{aligned}
(M_2 \circ M_1)(C) &= M_2(M_1(C)) \\
&= \lambda\,\mathsf{self}_2.\ \mathbf{Y}(M_1(C)) \oplus \Delta_2(M_1(C)))(\mathsf{self}_2) \\
&= \lambda\,\mathsf{self}_2.\ M_1(C)(\mathbf{Y}(M_1(C))) \oplus \Delta_2(M_1(C)))(\mathsf{self}_2) \\
&= \lambda\,\mathsf{self}_2.\ \mathbf{Y}(C) \oplus \Delta_1(C)(\mathbf{Y}(M_1(C))) \oplus \Delta_2(M_1(C)))(\mathsf{self}_2)
\end{aligned}
$$

so that we obtain:

$$M_2 \circ M_1 = \lambda\,\mathsf{super}\,\lambda\,\mathsf{self}.\ \mathbf{Y}(\mathsf{super}) \oplus \Delta_1(\mathsf{super})(M_1(\mathsf{super})) \oplus \Delta_2(M_1(\mathsf{super}))(\mathsf{self}),$$

which is essentially Bracha's wrapper composition in §5.2 of [7]. In the slightly more complex case of recursive mixins (without the state):

$$M_i' \stackrel{\Delta}{=} \lambda\,\mathsf{super}_i\,\lambda\,\mathsf{myClass}_i\,\lambda\,\mathsf{self}_i.\ \mathbf{Y}(\mathbf{Y}\,\mathsf{super}_i) \oplus \Delta_i'(\mathsf{super}_i)(\mathsf{myClass}_i)(\mathsf{self}_i)$$

composition $M_2' \circ M_1'$ is given by:

$$
\begin{aligned}
\lambda\,\mathsf{super}\,\lambda\,\mathsf{myClass}\,\lambda\,\mathsf{self}.\ &\mathbf{Y}(\mathbf{Y}\,\mathsf{super}) \oplus \\
&\Delta_1'(\mathbf{Y}(M_1'\,\mathsf{super}))\,(\mathbf{Y}(\mathbf{Y}(M_1'\,\mathsf{super}))) \oplus \Delta_2'(M_1'\,\mathsf{super})(\mathsf{myClass})(\mathsf{self}).
\end{aligned}
$$

In spite of their complex shape, recursive mixins have arrow types with recursive-record types as both left and right-hand sides. It follows that their composition into a *linearized* form [8] is just functional composition, that combined with the subtyping relation can be typed by:

$$\frac{\vdash M_1' : \kappa_1' \to \kappa_2' \qquad \vdash M_2' : \kappa_3' \to \kappa_4' \qquad \kappa_2' \le \kappa_3'}{\vdash M_2' \circ M_1' : \kappa_1' \to \kappa_4'}$$

Subtype properties. In general a mixin abstraction

$$\lambda\,\mathsf{super}.\lambda\,\mathsf{myClass}.\lambda\,\vec{x}.\lambda\,\mathsf{self}.(\mathbf{Y}((\mathbf{Y}\,\mathsf{super})\,\vec{y})) \oplus \Delta(\mathsf{super})(\mathsf{myClass})(\vec{x})(\mathsf{self}), \qquad \vec{y} \subseteq \vec{x} \qquad (1)$$

has a type of the form $\kappa_1' \to \kappa_2'$, and the instances of classes κ_i' have record-type σ_i for $i \in \{1, 2\}$, say, so that σ_2 is the type of body of the abstraction (1). Inequation $\sigma_2 \le \sigma_\Delta$, where σ_Δ is the type of $\Delta(\mathsf{super})(\mathsf{myClass})(\vec{x})(\mathsf{self})$, holds as a direct consequence of the rule ($\oplus_r$). However, $\sigma_2 \le \sigma_1$ is only satisfied, if no label of the class super was overwritten. Therefore, our mixin encoding is more powerful [21] than the existing approach of abstract base class inheritance. Analogously to Cook et al. [12], we pinpoint the distinction between mixin application and subtyping.

4 Mixin Features In Existing Programming Languages

In this section we implement mixins, presented in the lambda-calculus with records and a record-merge operator, using standard features of modern Object Oriented (OO) languages. The correspondence between calculus and programs is exemplified, in particular the correspondence between typability in lambda calculus and typability in Java like type systems.

Applying the theoretical model to a practically useful programming language, we face several challenges. First, mixins are not native concepts in mainstream OO languages. Code-generation and the regularity of a design pattern exploiting existing features, namely delegation, abstract classes, inheritance and generics, can compensate for this deficit. In Java-like languages inheritance alone is incapable to implement all mixin features due to restrictions of overriding (cf. Section 3) and overloading methods. Second, real languages use references and side effects which are not present in the presented calculus. To approximate this, state is modeled functionally in the calculus, while access to state is restricted only to *get* and *set* methods in the real language, e.g. Java. We consider the following subset of the presented calculus:

1. We consider recursive classes C with an aggregated state x:
   $C \stackrel{\Delta}{=} \lambda \mathsf{myClass}.\lambda x.\lambda \mathsf{self}.\langle a_1 = M_1, \ldots, a_k = M_k \rangle$

2. We consider recursive mixins M that map recursive classes to recursive classes and do no add state:
   $M \stackrel{\Delta}{=} \lambda \mathsf{super}.\lambda \mathsf{myClass}.\lambda x.\lambda \mathsf{self}.\ \mathbf{Y}((\mathbf{Y}\,\mathsf{super})\,x) \oplus \langle a_1 = M_1, \ldots, a_k = M_k \rangle$

3. State, passed as variable x, is directly accessed only by *get* and *set* methods.

4. Each method that modifies the state of the underlying object returns a tuple (*newState*, *result*), where *newState* is the new state of the object (updated value to pass as x, not the updated object) after the method call and *result* is the result of the method call.

5. Each class and each mixin contains the special method $new = \lambda x'.\mathbf{Y}(\mathsf{myClass}\,x')$

In addition the above restrictions, we capture common subterms and decompose tuples using the "let ... in" construction, which only serves as syntactic sugar and disallows any recursive definition.

Restriction 2 is used to separate the concerns of storing data in classes and add/replace methods in mixins. Additionally, restrictions 2 and 4 are used for seamless delegation of methods. Restrictions 4 and 5 are used to implement functional state update. As an example, we model the invocation of method a in an object o that updates the state of o and returns a result r by

$$\mathsf{let}\ (x,r) = o.a\ \mathsf{in}\ (\mathsf{let}\ o' = o.new(x)\ \mathsf{in}\ \ldots)$$

After the invocation of a, the variable o' is bound to a new instance of the underlying class with an updated state. Note that the underlying class of o is not expressed explicitly.

There are various more advanced approaches for modeling state in typed λ-calculi, e.g. reference cells [13] or Monads [17]. The presented version using *get* and *set* methods, is chosen for simplicity and compatibility to the type system.

The remaining section describes the relationship between mixins in the presented calculus and well typed code implementing mixin features. This relationship is a necessary step for a future extension of manual mixin application to provably reliable automatic synthesis. In particular, the following correspondences are demonstrated:

- Recursive classes in the presented calculus and classes in OO languages.

- Δ-terms (records of updated methods) of recursive mixins and design pattern based Δ-definitions.

- Mixin application and delegation based composition of classes and mixin implementations.

```
Class  Point
State
 x:  int
Definitions
 get:  Unit  →  int
 get() = { self.x }
 set:  int  →  Unit
 set(x') = { self.x := x' }
 shift:  Unit  →  Unit
 shift() = { self.set(self.get() + 1) }
```

(a) Pseudo-code of class Point

Point
- x : int
+ Point(x : int) + get() : int + set(x : int) : void + shift() : void
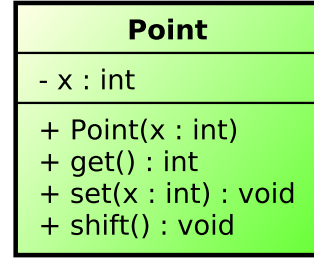
(b) UML diagram of class Point

Figure 1: Pseudo-code and UML diagram of class Point

To illustrate those correspondences we use pseudo-code as an intermediate layer between calculus and real code. For brevity, implementation code is illustrated via UML diagrams. Example implementations in Java and C# are available in online [2].

We extend the model-theoretic class Point from Section 2 to the new class Point for interpreting the stateful Java/C# class Point, by adding the methods *set* and *shift*. The pseudo-code implementation as well as an UML representation of the Java/C# implementation of Point are shown in Figure 1. Respecting above restrictions to the presented calculus, we represent Point by the following term:

$$\text{Point} = \lambda\,\text{myClass}.\lambda x.\lambda\,\text{self}.$$
$$\langle \text{get} = x, \text{set} = \lambda x'.(x', ()), \text{shift} = \text{self.set}(\text{self.get} + 1), \text{new} = \lambda x'.\mathbf{Y}(\text{myClass}\,x') \rangle$$

Note that interaction with the state x of Point happens via the get and set. We derive a type κ'_{Point} out of the many types of Point using the following pattern:

$$\sigma_1^{\text{Point}} = \langle \text{get} : int, \text{set} : int \rightarrow (int \times Unit), \text{shift} : \omega, \text{new} : int \rightarrow \omega \rangle$$
$$\sigma_2^{\text{Point}} = \langle \text{get} : int, \text{set} : int \rightarrow (int \times Unit), \text{shift} : (int \times Unit), \text{new} : int \rightarrow \omega \rangle$$
$$\sigma_3^{\text{Point}} = \langle \text{get} : int, \text{set} : int \rightarrow (int \times Unit), \text{shift} : (int \times Unit), \text{new} : int \rightarrow \sigma_2^{\text{Point}} \rangle$$
$$\kappa_1^{\text{Point}} = int \rightarrow (\omega \rightarrow \sigma_1^{\text{Point}}) \cap (\sigma_1^{\text{Point}} \rightarrow \sigma_2^{\text{Point}})$$
$$\kappa_2^{\text{Point}} = int \rightarrow (\omega \rightarrow \sigma_1^{\text{Point}}) \cap (\sigma_1^{\text{Point}} \rightarrow \sigma_2^{\text{Point}}) \cap (\sigma_2^{\text{Point}} \rightarrow \sigma_3^{\text{Point}})$$
$$\kappa'_{\text{Point}} = (\omega \rightarrow \kappa_1^{\text{Point}}) \cap (\kappa_1^{\text{Point}} \rightarrow \kappa_2^{\text{Point}})$$

The type σ_3^{Point} describes an instance of Point after the double fixed point construction fixing myClass and self. The type σ_2^{Point} describes the intermediate term resulting from the first fixed point construction that fixes myClass.

Next, we define two mixins Movable and SetAdapter presented in Figure 2. Movable requires the get and set methods to define the move method, moving the current point by *dx*. *SetAdapter* redefines the set method to handle an instance of Point as input. Note that super is used to access methods of the corresponding mixin argument.

---

[2]`http://www-seal.cs.tu-dortmund.de/seal/downloads/research/ITRS2014PostProc.zip`

```
Mixin  Movable (C)
Requirements
 get:  Unit  →  int
 set:  int  →  Unit
Definitions
 move:  int  →  Unit
 move(dx)  =
   {  super.set(super.get()+dx)  }
```
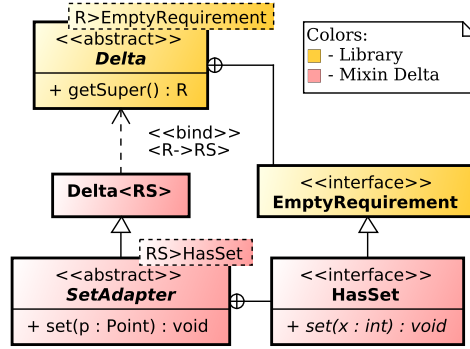
(a) Pseudo-code of mixin Movable
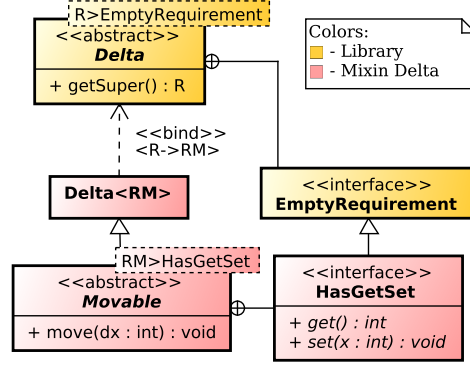
(b) UML Diagram of mixin Movable

```
Mixin  SetAdapter (C)
Requirements
 set:  int  →  Unit
Definitions
 set:  Point  →  Unit
 set(p)  =
   {  super.set(p.get())  }
```

(c) Pseudo-code of mixin SetAdapter

(d) UML Diagram of mixin SetAdapter

Figure 2: Pseudo-code and UML diagrams of mixins Movable and SetAdapter

Respecting above restrictions to the presented calculus, we represent Movable by the following term:

$$Movable = \lambda\, super.\lambda\, myClass.\lambda x.\lambda\, self.$$
$$\text{let } c = \mathbf{Y}((\mathbf{Y}\, super)x) \text{ in}$$
$$\text{let } \Delta = \langle move = \lambda dx.\ c.set(c.get + dx), new = \lambda x'.\mathbf{Y}(myClass\, x') \rangle \text{ in}$$
$$c \oplus \Delta$$

We derive a type $\kappa'_{\mathsf{Point}} \to \kappa'_{\mathsf{Movable}}$ out of the many types of Movable using the following pattern:

$$\sigma_1^{\mathsf{Movable}} = \langle \mathrm{get} : int, \mathrm{set} : int \to (int \times Unit), \mathrm{shift} : (int \times Unit) \rangle$$
$$\cap \langle \mathrm{move} : int \to (int \times Unit), \mathrm{new} : int \to \omega \rangle$$
$$\sigma_2^{\mathsf{Movable}} = \langle \mathrm{get} : int, \mathrm{set} : int \to (int \times Unit), \mathrm{shift} : (int \times Unit) \rangle$$
$$\cap \underbrace{\langle \mathrm{move} : int \to (int \times Unit), \mathrm{new} : int \to \sigma_1^{\mathsf{Movable}} \rangle}_{\text{type } \sigma_{\Delta}^{\mathsf{Movable}} \text{ induced by the mixin description}}$$
$$\kappa_1^{\mathsf{Movable}} = int \to (\omega \to \sigma_1^{\mathsf{Movable}})$$
$$\kappa_2^{\mathsf{Movable}} = int \to (\omega \to \sigma_1^{\mathsf{Movable}}) \cap (\sigma_1^{\mathsf{Movable}} \to \sigma_2^{\mathsf{Movable}})$$
$$\kappa'_{\mathsf{Movable}} = (\omega \to \kappa_1^{\mathsf{Movable}}) \cap (\kappa_1^{\mathsf{Movable}} \to \kappa_2^{\mathsf{Movable}})$$

Note that the type of move : $int \rightarrow (int \times Unit)$ follows a uniform type structure not bound to the mixin Movable and, therefore, allows for seamless delegation.

Similarly, SetAdapter is represented by the following term and is typed by $\kappa'_{\text{Movable}} \rightarrow \kappa'_{\text{SetAdapter}}$:

$$\text{SetAdapter} = \lambda \text{super.} \lambda \text{myClass.} \lambda x. \lambda \text{self.}$$
$$\text{let } c = \mathbf{Y}((\mathbf{Y}\text{super})x) \text{ in}$$
$$\text{let } \Delta = \langle set = \lambda p.c.set(p.get), new = \lambda x'.\mathbf{Y}(\text{myClass}x')\rangle \text{ in}$$
$$c \oplus \Delta$$

$$\sigma_1^{\text{SetAdapter}} = \langle get : int, shift : (int \times Unit), move : int \rightarrow (int \times Unit)\rangle$$
$$\cap \langle set : \langle get : int\rangle \rightarrow (int \times Unit), new : int \rightarrow \omega\rangle$$
$$\sigma_2^{\text{SetAdapter}} = \langle get : int, shift : (int \times Unit), move : int \rightarrow (int \times Unit)\rangle$$
$$\cap \langle set : \langle get : int\rangle \rightarrow (int \times Unit), new : int \rightarrow \sigma_1^{\text{SetAdapter}}\rangle$$
$$\sigma_3^{\text{SetAdapter}} = \langle get : int, shift : (int \times Unit), move : int \rightarrow (int \times Unit)\rangle$$
$$\cap \underbrace{\langle set : \langle get : int\rangle \rightarrow (int \times Unit), new : int \rightarrow \sigma_2^{\text{SetAdapter}}\rangle}_{\text{type } \sigma_\Delta^{\text{SetAdapter}} \text{ induced by the mixin description}}$$
$$\kappa_1^{\text{SetAdapter}} = int \rightarrow (\omega \rightarrow \sigma_1^{\text{SetAdapter}})$$
$$\kappa_2^{\text{SetAdapter}} = int \rightarrow (\omega \rightarrow \sigma_1^{\text{SetAdapter}}) \cap (\sigma_1^{\text{SetAdapter}} \rightarrow \sigma_2^{\text{SetAdapter}})$$
$$\kappa_3^{\text{SetAdapter}} = int \rightarrow (\omega \rightarrow \sigma_1^{\text{SetAdapter}}) \cap (\sigma_1^{\text{SetAdapter}} \rightarrow \sigma_2^{\text{SetAdapter}}) \cap (\sigma_2^{\text{SetAdapter}} \rightarrow \sigma_3^{\text{SetAdapter}})$$
$$\kappa'_{\text{SetAdapter}} = (\omega \rightarrow \kappa_1^{\text{SetAdapter}}) \cap (\kappa_1^{\text{SetAdapter}} \rightarrow \kappa_2^{\text{SetAdapter}}) \cap (\kappa_2^{\text{SetAdapter}} \rightarrow \kappa_3^{\text{SetAdapter}})$$

We encode Δ-terms in the implementation. Each Δ-term is represented by an abstract class extending the class `Delta`, which is available in an utility library. Δ-term representations are parameterized over a requirement interface type. This type is used for the library method `getSuper`, which provides the instance with which the Δ implementation will be merged. This instance can be guaranteed to be at least of the required interface type. Required interface types may be declared for each Δ (e.g. `HasGetSet`). They are interfaces which, for reasons of structural regularity, extend the `EmptyRequirement` interface, also available in the utility library. Δ-term representations remain parametric over the type of the term they are merged with in order to allow `getSuper` to return an instance of that type, rather than just an instance of the requirement interface type. This is necessary to reuse `getSuper` for delegation of methods propagated after mixin application, without mentioning them in the requirement interface.

Figure 4 shows the implemented result of applying `Movable` to `Point` and (below the dashed line) the application of `SetAdapter` to the previous result. First, `MovablePoint` is obtained by creating a new class that inherits from `Movable`. It thereby satisfies $\sigma_n^{\text{Mixin}} \leq \sigma_\Delta^{\text{Mixin}}$. Internally, `MovablePoint` delegates to the methods of a `PointHasGetSet` instance provided by the `getSuper` method inherited from `Delta`. Class `PointHasGetSet` extends `Point` and implements the requirement interface. The introduction of a new class is necessary in languages not supporting structural subtyping. Structurally `Point` $\leq$ `HasGetSet` does hold, but in a nominal type system [12] subtype relationships have to be stated explicitly. The class `MovablePoint` propagates the non overwritten methods of `Point`, e.g. `shift`. Note that `shift` is not part of the requirement interface, but available since the type parameter of `Movable` is

```
1
2  p1 := new SetAdapter(Movable(Point))(1)
3  p2 := new Point(2)
4  p1.set(p2)
5
6  p1.move(1)
7
8  p1.get()
```

$$\text{let } C = (SetAdapter \circ Movable)(Point) \text{ in}$$
$$\text{let } p1 = \mathbf{Y}((\mathbf{Y}C)1) \text{ in}$$
$$\text{let } p2 = \mathbf{Y}((\mathbf{Y}Point)2) \text{ in}$$
$$\text{let } (x,r) = p1.set(p2) \text{ in}$$
$$\text{let } p1' = p1.new(x) \text{ in}$$
$$\text{let } (x',r') = p1'.move(1) \text{ in}$$
$$\text{let } p1'' = p1'.new(x') \text{ in}$$
$$p1''.get()$$

(a) Implementation pseudo-code   (b) Λ_R-calculus

Figure 3: Instantiation and usage of Mixins

bound to `PointHasGetSet`. Similarly, `SetAdapterMovablePoint` extends `SetAdapter` and delegates to `MovablePointHasSet`, which connects `HasSet` and `MovablePoint`.

The UML diagram reveals the very regular nature of mixin application, which in a practical setting can be automated, s.t. blue classes in the diagram are synthesized by a code generator and do not have to be implemented by hand. Code generation corresponds to Λ_R reduction, performing the merge operation $\oplus$ on the super argument. In the case of Java and C# it is to be performed outside of the implementation language, because the creation of new named classes is not a native language feature.

We compare the usage of the pseudo-code implementation and calculus description in the example shown in Figures 3a and 3b. Implementations in Java and C# are also available online and almost identical to the pseudo-code implementation. The main differences between the pseudo-code and the Λ_R version are the initial class instantiation and the different styles of passing state. In OO languages class instantiation is performed by the built-in operator `new`. In Λ_R we model `new` using the fixpoint operator $\mathbf{Y}$ respectively (cf. Section 3). Passing state functionally requires explicit instantiation of copies of $p1$ with an updated state (lines 5 and 7). Observe that in $p1.set$ (line 4) is the new set method, however, $p1.move$ (line 6) uses the old set method, which is the required behavior. Therefore, the final result produced by both implementations (line 8) is 3 : *int*. Observing that $\sigma_2^{\mathsf{SetAdapter}}$ are $\sigma_3^{\mathsf{SetAdapter}}$ necessary for the twofold state update of $p1$, we derive $\vdash p1 : \sigma_3^{\mathsf{SetAdapter}}$ using the following derivation for $(\mathsf{SetAdapter} \circ \mathsf{Movable})(\mathsf{Point})$:

$$\frac{\dfrac{\vdash \mathsf{Movable} : \kappa'_{\mathsf{Point}} \to \kappa'_{\mathsf{Movable}} \qquad \vdash \mathsf{SetAdapter} : \kappa'_{\mathsf{Movable}} \to \kappa'_{\mathsf{SetAdapter}}}{\vdash \mathsf{SetAdapter} \circ \mathsf{Movable} : \kappa'_{\mathsf{Point}} \to \kappa'_{\mathsf{SetAdapter}}} \qquad \vdash \mathsf{Point} : \kappa'_{\mathsf{Point}}}{\vdash (\mathsf{SetAdapter} \circ \mathsf{Movable})(\mathsf{Point}) : \kappa'_{\mathsf{SetAdapter}}}$$
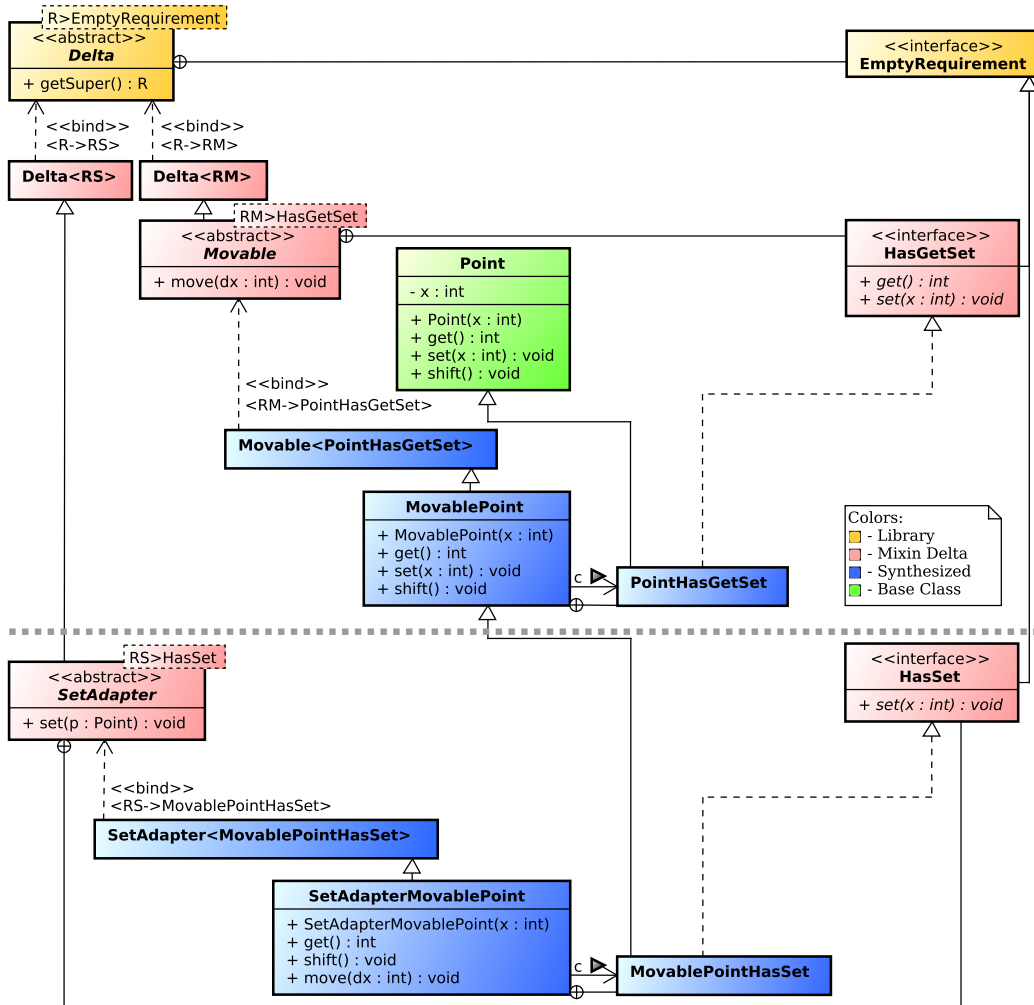
Figure 4: UML Diagram for synthesized code. The dotted line separates Movable(Point) from SetAdapter(Movable(Point))

5   Conclusions and further work

In this paper we provide a lambda-calculus with records and a record-merge operator to describe classes, mixins and mixin-applications in object-oriented languages featuring an early binding of method calls. We demonstrate its potential by composing well typed programs in Java and C# that itself do not inherently support mixins. The delegation based approach in combination with subtyping turns out to be more flexible as well as powerful than inheritance and overloading. In future, we aim to develop a theory for automatically synthesizing code for mixin-applications using an approach introduced in [5].

References

[1] Martín Abadi & Luca Cardelli (1996): *A Theory of Objects*. Springer, doi:10.1007/978-1-4419-8598-9.

[2] Roberto M. Amadio & Pierre-Louis Curien (1998): *Domains and Lambda-Calculi*. Cambridge University Press, doi:10.1017/CBO9780511983504.

[3] Steffen van Bakel & Ugo de'Liguoro (2008): *Logical Equivalence for Subtyping Object and Recursive Types*. Theory Comput. Syst. 42(3), pp. 306–348, doi:10.1007/s00224-007-9079-5.

[4] H. Barendregt, M. Coppo & M. Dezani-Ciancaglini (1983): *A Filter Lambda Model and the Completeness of Type Assignment*. Journal of Symbolic Logic 48(4), pp. 931–940, doi:10.2307/2273659.

[5] Jan Bessai, Boris Düdder, Andrej Dudenhefer & Moritz Martens (2014): *Delegation-based Mixin Composition Synthesis*. Available at `http://www-seal.cs.tu-dortmund.de/seal/downloads/papers/paper-ITRS2014.pdf`.

[6] Viviana Bono, Amit Patel & Vitaly Shmatikov (1999): *A Core Calculus of Classes and Mixins*. In: *ECOOP*, Lecture Notes in Computer Science 1628, pp. 43–66, doi:10.1007/3-540-48743-3_3.

[7] Gilad Bracha (1992): *The Programming Language JIGSAW: Mixins, Modularity and Multiple Inheritance*. Ph.D. thesis, Univeristy of Utha.

[8] Gilad Bracha & William R. Cook (1990): *Mixin-based Inheritance*. In: *OOPSLA/ECOOP*, pp. 303–311, doi:10.1145/97945.97982.

[9] Kim B. Bruce (2002): *Foundations of Object-Oriented Languages - Types and Semantics*. MIT Press.

[10] Peter S. Canning, William R. Cook, Walter L. Hill, Walter G. Olthoff & John C. Mitchell (1989): *F-Bounded Polymorphism for Object-Oriented Programming*. In: *FPCA*, pp. 273–280, doi:10.1145/99370.99392.

[11] Luca Cardelli (1984): *A Semantics of Multiple Inheritance*. In: Semantics of Data Types, Sophia-Antipolis, France, June 27-29, 1984, Proceedings, LNCS 173, pp. 51–67, doi:10.1016/0890-5401(88)90007-7.

[12] William R. Cook, Walter L. Hill & Peter S. Canning (1990): *Inheritance Is Not Subtyping*. In: *POPL'90*, ACM Press, pp. 125–135, doi:10.1145/96709.96721.

[13] Rowan Davies & Frank Pfenning (2000): *Intersection Types and Computational Effects*. In: *ICFP*, pp. 198–208, doi:10.1145/351240.351259.

[14] Ugo de'Liguoro (2001): *Characterizing Convergent Terms in Object Calculi via Intersection Types*. In: *TLCA*, pp. 315–328, doi:10.1007/3-540-45413-6_25.

[15] Ugo de'Liguoro & Tzu chun Chen (2014): *Semantic Types for Classes and Mixins*. Available at `http://www.di.unito.it/~deligu/papers/UdLTC14.pdf`.

[16] Mariangiola Dezani-Ciancaglini, Elio Giovannetti & Ugo de' Liguoro (1998): *Intersection Types, Lambda-models and Böhm Trees*. In: *MSJ-Memoir Vol. 2 "Theories of Types and Proofs"*, 2, Mathematical Society of Japan, pp. 45–97. Available at `http://www.di.unito.it/~dezani/papers/11.ps`.

[17] Oleg Kiselyov & Ralf Lämmel (2005): *Haskell's overlooked object system*. CoRR abs/cs/0509027.

[18] Martin Odersky & Matthias Zenger (2005): *Scalable component abstractions*. In: *OOPSLA*, ACM, pp. 41–57, doi:10.1145/1094811.1094815.

[19] Bruno C d S Oliveira, Tijs Van Der Storm, Alex Loh & William R Cook (2013): *Feature-Oriented programming with object algebras*. In: *ECOOP*, Springer, pp. 27–51, doi:10.1007/978-3-642-39038-8_2.

[20] Addy Osmani (2012): *Learning JavaScript Design Patterns*. O'Reilly Media, Inc.

[21] Klaus Ostermann (2002): *Dynamically composable collaborations with delegation layers*. In: *ECOOP*, Springer, pp. 89–110, doi:10.1007/3-540-47993-7_4.

[22] Jakob Rehof (2013): *Towards Combinatory Logic Synthesis*. In: *BEAT'13, 1st International Workshop on Behavioural Types*, ACM.

[23] Reuben N. S. Rowe & Steffen van Bakel (2014): *Semantic Types and Approximation for Featherweight Java*. Theor. Comput. Sci. 517, pp. 34–74, doi:10.1016/j.tcs.2013.08.017.