Delegation-based Mixin Composition Synthesis

Jan Bessai Boris Düdder Andrej Dudenhefner
Moritz Martens
Technical University of Dortmund, Department of Computer Science

In the object-oriented world existing inheritance concepts have been extended by mixins to assist modular software design. Actually, most languages only have limited mixin support. We use staged computation proposed by Davies and Pfenning together with a specification following an interface based design pattern to introduce mixins to Java 8. We present a synthesis based approach for mixin composition guided by user annotated semantics translated to intersection types. Synthesis results in meaningful Java code utilizing delegation to augment existing classes with mixin features.

Keywords: Mixin, Intersection Types, Composition Synthesis, Staged Computation, Inhabitation

1 Introduction

Complex software cannot be written from scratch and relies on components from existing libraries. Additionally, a high level of modularity is required for software scalability and maintenance. In the object-oriented world, inheritance is a fundamental feature for modular component design. Since class-based inheritance is hierarchically constrained, it only has limited potential for code reuse. *Mixins* [1, 3, 8] and *traits* [11, 6] are two major innovations to support modular component design.

A mixin maps an existing class to an extended class by adding or replacing member variables and methods. It is not bound to a particular parent class but specifies members required to be applicable. Therefore, a mixin can be applied to many parent classes and greatly improves modularity and reusability of code. Additionally, a mixin can be used as an interface for different instantiations of the same mixin.

Figure 1 sketches typical examples of mixins. Given any class C the mixin `ByteOrderSwapper` creates a new class by adding the method `swap` that swaps the byte order of the input to C. `swap` might be useful for communication between big-endian and little-endian architectures. Given any class C that provides methods `process` and `swap`, the mixin `ByteOrderAdapter` redefines the method `process` by applying `swap` to the result of the old implementation of `process`. Therefore, if a class C provides the method `process` then the mixin application `ByteOrderAdapter(ByteOrderSwapper(`C`))` redefines the method `process` by swapping the byte-order of the output of the original method.

```
1 Mixin ByteOrderSwapper(C)
2 Definitions
3  swap: int → int
4  swap(value) = ...
```

```
1 Mixin ByteOrderAdapter(C)
2 Requirements
3  process: int → int
4  swap: int → int
5 Definitions
6  process: int → int
7  process(input) = C.swap(C.process(input))
```

Figure 1: `ByteOrderSwapper` and `ByteOrderAdapter` mixins

Submitted to:
ITRS 2014

Similar to a mixin, a trait provides methods that implement behavior parametrized by required methods and can be used as an interface. In contrast to mixin composition, trait composition affects neither the semantics of the underlying class nor the semantics of other traits. Additionally, the composition order of traits is irrelevant and all conflicts must be resolved explicitly. Preferably, a collection of traits defines pairwise disjoint behavioral building blocks. Traits solve several problems of mixins such as the need for a total order of a mixin composition and the fragility of resulting inheritance chains with respect to change. However, traits are purely behavioral and additive, i.e. they cannot modify the set of member variables or existing behavior of particular methods in a class.

We introduce a structured type system for mixins, modeling them with well understood concepts like intersection and record types. We present a synthesis based approach for mixin composition, which we consider novel. The goal of the synthesis procedure is to find an ordered mixin composition chain, implementing requested behavior. Automatization and the decision to focus on behavior (methods) rather than state (member variables) overcome some existing problems addressed by Schärli et al. with the introduction of traits in [11]. We algorithmically extract typed combinators, which serve as input for the synthesis procedure, from Java code annotated with semantic types. We provide an exemplary translation of a mixin composition chain to Java 8 compensating missing language features by delegation. In contrast to [1], our approach does not rely on adding features to Java, while going beyond the trait-like capabilities achievable by implementing interfaces that provide default methods.

We proceed as follows: Section 2 describes the type system used for class-like structures. Section 3 models mixins using staged computation. Section 4 applies our approach to Java 8 code.

2 Type System

Our main goal is to modify class-like structures. Allowing future extension, we abstract from particular implementations of methods and describe method bodies F by the language L0: $F ::= f$, where $f \in \mathbb{V}_F$ is a variable representing a method body. To type L0 we use intersection types $\mathbb{T}$ [2], which are particularly interesting to explicitly specify semantical properties, e.g. the intersection type int $\cap$ TM may describe an integer that encodes a Turing machine. $\mathbb{T}$ is given by ($a \in \mathbb{A}$ is a constant and $\alpha \in \mathbb{V}$ is a type variable):

$$\mathbb{T} \ni \tau, \tau_1, \tau_2 \ ::= a \mid \alpha \mid \omega \mid \tau_1 \to \tau_2 \mid \tau_1 \cap \tau_2$$

The only assignment rule for method bodies is (Σ is a set of bindings of the form $f : \tau$):

$$\frac{}{\Sigma, f : \tau \vdash_{\mathsf{L0}} f : \tau}(\mathsf{Var})$$

We define class terms T using a subset of the λ-calculus with records Λ_R [5] by the language L1 ($u \in \mathbb{V}_T$ is a term variable, $l \in \mathbb{L}$ is a label and F is a method body):

$$T ::= \langle\rangle \mid u \mid T.l := F$$

We use intersection types extended by record types $\mathbb{T}_{\langle\rangle}$ to type L1. The set $\mathbb{T}_{\langle\rangle}$ of intersection types for Λ_R is given by ($a \in \mathbb{A}$ is a constant and $\alpha \in \mathbb{V}$ is a variable):

$$\mathbb{T}_{\langle\rangle} \ni \xi, \xi_1, \xi_2 \ ::= a \mid \alpha \mid \omega \mid \xi_1 \to \xi_2 \mid \xi_1 \cap \xi_2 \mid \langle\rangle \mid \langle l : \xi \rangle$$

The record type $\langle l : \xi \rangle$ is used to associate the label l with the type ξ; individual record types are combined by intersection. We assume subtyping preorder $\leq$ axioms from [2] adding following axioms:

$$1.\ \langle a : \xi \rangle \leq \langle\rangle \qquad 2.\ \xi_1 \leq \xi_2 \Rightarrow \langle l : \xi_1 \rangle \leq \langle l : \xi_2 \rangle \qquad 3.\ \langle l : \xi_1 \rangle \cap \langle l : \xi_2 \rangle \leq \langle l : \xi_1 \cap \xi_2 \rangle$$

To type class terms T we add the L0-context Σ and the following assignment rules to the BCD [2] system (Δ is a set of bindings of the form $u : \xi$):

$$\frac{}{\Sigma;\Delta \vdash_{\mathsf{L1}} \langle\rangle : \langle\rangle}(\mathsf{EmtpyRec}) \quad \frac{\Sigma \vdash_{\mathsf{L0}} F : \xi}{\Sigma;\Delta \vdash_{\mathsf{L1}} T.l := F : \langle l : \xi\rangle}(\mathsf{Upd1}) \quad \frac{\Sigma;\Delta \vdash_{\mathsf{L1}} T : \langle l_1 : \xi\rangle \qquad l_1 \not\equiv l_2}{\Sigma;\Delta \vdash_{\mathsf{L1}} T.l_2 := F : \langle l_1 : \xi\rangle}(\mathsf{Upd2})$$

Proposition 1 (Class Type)**.** *Given an L0-context Σ, method bodies $F_1,\dots,F_m$ with $\Sigma \vdash_{\mathsf{L0}} F_i : \tau_i$ for $1 \leq i \leq m$ and a term $T = (\dots(\langle\rangle.l_1 := F_1)\dots).l_m := F_m$, there exists a type $\xi = \langle\rangle \cap \bigcap_{i=1}^{m} \langle l_i : \tau_{j_i}\rangle$, where $j_i \in \{1,\dots,m\}$ for $1 \leq i \leq m$, so that $\Sigma;\Delta \vdash_{\mathsf{L1}} T : \xi$. For $m = 0$ we have $\Sigma;\Delta \vdash_{\mathsf{L1}} T : \langle\rangle$.*

3 Staged Mixin Composition Synthesis

Composition synthesis [10] is a type-based approach to code synthesis based on combinators that are implemented in a native language, L1, and contained in a repository. Staged composition synthesis (SCS) [7] enhances this approach by adding a metalanguage, L2, in which composition combinators can be implemented. Such combinators can perform complex manipulations and substitutions of L1-code fragments. Since SCS allows for composition of both L1- and L2-components it results in a flexible synthesis framework, which we use to synthesize mixin compositions. We adapt the necessary formal concepts and refer to [7] for more detailed discussions.

The underlying native language L1 is the language of class terms T. To mirror the language distinction between L1 and L2 at type-level we use modal types $\Box\xi$ describing "L1-code of L1-type ξ"as introduced by Davies and Pfenning in [4] and define the set $\mathbb{T}_2$ of L2-types by:

$$\mathbb{T}_2 \ni \sigma,\sigma_1,\sigma_2 ::= \Box\xi \mid \sigma_1 \to \sigma_2$$

L2-terms N are terms of the $\lambda_e^{\Box\to}$-calculus [4] defined by:

$$N,N_1,N_2 ::= \texttt{box}\ T \mid \texttt{letbox}\ u = N_1\ \texttt{in}\ N_2 \mid x \mid \lambda x.N \mid (N_1\ N_2)$$

and typed by the following rules (Σ and Δ are as above and Γ is a set of bindings of the form $x : \sigma$):

$$\frac{}{\Sigma;\Delta;(\Gamma,x:\sigma) \vdash_{\mathsf{L2}} x : \sigma}(\text{Var}) \quad \frac{\Sigma;\Delta;(\Gamma,x:\sigma) \vdash_{\mathsf{L2}} N : \sigma'}{\Sigma;\Delta;\Gamma \vdash_{\mathsf{L2}} \lambda x.N : \sigma \to \sigma'}(\to\text{I}) \quad \frac{\Sigma;\Delta \vdash_{\mathsf{L1}} T : \xi}{\Sigma;\Delta;\Gamma \vdash_{\mathsf{L2}} \texttt{box}\ T : \Box\xi}(\Box\text{I})$$

$$\frac{\Sigma;\Delta;\Gamma \vdash_{\mathsf{L2}} N_1 : \sigma \to \sigma' \quad \Sigma;\Delta;\Gamma \vdash_{\mathsf{L2}} N_2 : \sigma}{\Sigma;\Delta;\Gamma \vdash_{\mathsf{L2}} (N_1\ N_2) : \sigma'}(\to\text{E}) \quad \frac{\Sigma;\Delta;\Gamma \vdash_{\mathsf{L2}} N_1 : \Box\xi \quad \Sigma;(\Delta,u:\xi);\Gamma \vdash_{\mathsf{L2}} N_2 : \sigma}{\Sigma;\Delta;\Gamma \vdash_{\mathsf{L2}} \texttt{letbox}\ u = N_1\ \texttt{in}\ N_2 : \sigma}(\Box\text{E})$$

Note that $\leq$ can be extended in a natural way such that $\Box$ becomes a covariant type constructor. The operational semantics of L2 is given by the reduction relation $\longmapsto$ of the $\lambda_e^{\Box\to}$-calculus [4] generated by β-reduction and the `letbox`-reduction rule:

$$\texttt{letbox}\ u = \texttt{box}\ T\ \texttt{in}\ N \longmapsto N[u := T]$$

It can be shown [4] that the type system imposes a strict distinction between L1 and L2 such that before L1-computation all L2-terms are reduced to a boxed class term `box` T.

A mixin adds/modifies methods of a class. Given a class term T, we add/modify a method l so that the new implementation of l is F by $T.l := F$. Therefore, a mixin at its core is a sequence of $(\_.l := F)$ applications. The result of such applications to a class term is a new class term.

In most object-oriented languages there is no notion of a mixin at compilation-time. Thus, we use staged computation [4] to introduce an additional composition-time layer complemented by SCS to synthesize mixin compositions. Staged computation applies our well-typed mixins to classes ahead of compilation-time and produces valid code, even without extending the compiled language.

A mixin M that adds/modifies pairwise distinct methods $l_1,\ldots,l_m$ so that the corresponding new implementations are $F_1,\ldots,F_m$ is modeled by the L2 expression:

$$M = \lambda x.\texttt{letbox}\ u = x\ \texttt{in box}\ (\ldots(u.l_1 := F_1)\ldots).l_m := F_m$$

Due to the `letbox`/`box` construct we are able to modify given class code at composition-time.

Proposition 2 (Mixin Type)**.** *Given an L0-context Σ, method bodies $F_1,\ldots,F_m$ with $\Sigma \vdash_{\mathsf{L0}} F_i : \tau_i$ for $1 \leq i \leq m$ and a mixin $M = \lambda x.\,\texttt{letbox}\ u = x\ \texttt{in box}\ (\ldots(u.l_1 := F_1)\ldots).l_m := F_m$ with pairwise distinct labels $l_1,\ldots,l_m$, we have*

(i) $\Sigma;\emptyset;\emptyset \vdash_{\mathsf{L2}} M : \Box\langle\rangle \to \Box\langle l_i : \tau_i\rangle$ *for* $1 \leq i \leq m$.

(ii) $\Sigma;\emptyset;\emptyset \vdash_{\mathsf{L2}} M : \Box\langle l : \tau\rangle \to \Box\langle l : \tau\rangle$ *for all* $\tau \in \mathbb{T}$, $l \in \mathbb{L} \setminus \{l_1,\ldots,l_m\}$.

To abbreviate the infinitely many types of M provided by Proposition 2, we introduce the notation $\eta \setminus \{l_1,\ldots,l_m\}$ which can be seen as variable kinding for $\mathbb{T}_{\langle\rangle}$. Intuitively, $\eta \setminus \{l_1,\ldots,l_m\}$ is a record type containing no labels from $\{l_1,\ldots,l_m\}$, which exactly describes the types in property (ii) in Proposition 2. Using the described abbreviation we specify the type of a mixin M that defines or propagates methods $l_1 : \tau_1,\ldots,l_m : \tau_m$ and requires methods $l_{m+1} : \tau_{m+1},\ldots,l_{m+k} : \tau_{m+k}$ as:

$$M : \Box((\eta \setminus \{l_1,\ldots,l_{m+k}\}) \cap \bigcap_{i=m+1}^{m+k} \langle l_i : \tau_i\rangle) \to \Box((\eta \setminus \{l_1,\ldots,l_{m+k}\}) \cap \bigcap_{i=1}^{m} \langle l_i : \tau_i\rangle)$$

SCS is concerned with the *relativized inhabitation problem* (abbreviated by $\mathscr{C};\mathscr{D} \vdash_{\mathsf{C2}} ? : \sigma$) for combinatory logics C1 and C2 defined in [7], i. e. given typed C1, C2 combinator repositories $\mathscr{C}$, $\mathscr{D}$ and a synthesis goal specification σ, does there exist a combinatory term E with $\mathscr{C};\mathscr{D} \vdash_{\mathsf{C2}} E : \sigma$.

We expose typed C1-combinators for classes implemented in L1 and typed C2-combinators for mixins implemented in L2. Combinator types are restricted according to Propositions 1 and 2. We add semantic types via intersection to further describe the behavior of particular methods inside the classes. By suitably bounding the heights of the terms in substitutions and by adapting the notion of paths (cf. [7]), the inhabitation algorithm from [7] can be employed to synthesize mixin compositions.

4 Application

Integration into existing code bases is an important concern. Java provides an interface concept already compatible with many features of mixins: interfaces can extend multiple other interfaces, they can require methods and (since Java 8) they can also provide default method implementations [9]. Therefore, we can reuse interfaces as a specification language for mixins, if we are able to express their hierarchy independent applicability and its semantic consequences. We do so by introducing a design pattern and by adding annotations. Reflection later allows to retrieve specifications in a systematic manner suitable for automated combinator repository construction. At this point we take the design decision to model behavioral aspects of mixins only. While adding or replacing member variables would require minimal changes to our type system, we only support methods. This avoids programming errors resulting from

the failure to modularly keep up invariants on state deeply hidden in synthesized mixin hierarchies. State modification is still possible via accessor methods explicitly provided in base classes. Listing 1 illustrates a Java encoding of our pattern base interface `Mixin` and the ByteOrder example.

The `Mixin` base interface provides `Mixin.EmptyRequirement`, an interface which all requirements implement. It also requires Mixins to have a method `getSelf` returning a reference to their applicant, which in turn implements their requirement type. `Mixin` is parameterized over this concrete requirement type `R`. We express `ByteOrderAdapter` as an interface extending `Mixin`. It specializes the `EmptyRequirement` interface by adding the method signatures for `process: int` $\to$ `int` and `swap: int` $\to$ `int`. Java 8 allows to state a default method implementation for `process`. Internally, the implementation makes use of the `getSelf()` method available from `Mixin`. Calling `swap` and `process` on the result of `getSelf` will always be possible and well typed, because the parameter `R` passed to `Mixin` is bounded to be at least an extension of `ByteOrderAdapter.Requirement`. We again encode `ByteOrderAdapter` to be parametric over its requirement type, in order to allow further specialization of requirements by other mixins extending `ByteOrderAdapter`. Swapping is provided by the `ByteOrderSwapper` mixin, which has an empty requirement list. Semantic types guide the inhabitation process. Users add them as strings passed to the annotation `@Semantic`, thereby strengthening requirements and providing more information about declared methods. In the latter case they can also include conditions satisfiable by an equally named method existing in the mixin argument. This is especially useful for repeated applications of the same mixin. In our example it allows to toggle the byte order annotation of `process` every time `ByteOrderAdapter` is applied. The additional class `BaseProcess` serves as a starting point to which the mixins can be applied.

```
1  public interface Mixin<R extends Mixin.EmptyRequirement> {
2    public interface EmptyRequirement {}
3    public R getSelf(); }
4  public interface ByteOrderAdapter
5      <R extends ByteOrderAdapter.Requirement> extends Mixin<R> {
6    public interface Requirement extends Mixin.EmptyRequirement {
7      public int process(int input);
8      @Semantic("SWAPS_BYTEORDER")
9      public int swap(int value); }
10   @Semantics({
11     @Semantic(condition="LITTLE_ENDIAN", value="BIG_ENDIAN"),
12     @Semantic(condition="BIG_ENDIAN", value="LITTLE_ENDIAN")})
13   default public int process(int input) {
14     return getSelf().swap(getSelf().process(input));
15   } }
16 public interface ByteOrderSwapper
17     <R extends ByteOrderSwapper.Requirement> extends Mixin<R> {
18   public interface Requirement extends Mixin.EmptyRequirement {}
19   @Semantic("SWAPS_BYTEORDER")
20   default public int swap(int value) {
21     return ((value & 0xFF000000) >>> 24) | ((value & 0x00FF0000) >>> 8)
22          | ((value & 0x0000FF00) <<   8) | ((value & 0x000000FF) << 24);
23   } }
24 public class BaseProcess {
25   @Semantic("BIG_ENDIAN")
26   public int process(int input) { return input ^ 0x6D310AC4; } }
```

Listing 1: Java 8 implementation of the example in Figure 1 with semantic annotations and a base class

Our framework requires typed combinators as input. These should be created automatically for implementations structured like the example above. In order to derive types for a given piece of code we first check if it is a mixin or a class, to determine if the type will be an intersection of arrows or records. In both cases, all methods, including those inherited from base classes, have to be considered. Listing 2 formalizes the algorithm so far. Unless explicitly defined, we suppose `is-` and `get-` methods are implicitly available from reflection libraries. So are `set-` methods, that create copies of their first argument for which future calls to `get-` return the second argument. While collecting parent methods we assume that child methods with matching names (`correspondingTo`, line 26) shadow parent methods (`overrideUnion`, line 22).

```
1  buildType c
2    | isMixin c =
3      ⋂ {buildMixinArrow c' |
4         c'∈ enumerateDifferentSemantics (inlineParentMethods c)}
5    | otherwise =
6      □ (⋂ {buildRecordEntry RHS (inlineParentMethods c) m |
7            m ∈ getAllMethods getDeclaredMethods c})
8
9  inlineParentMethods c
10   | isMixin c =
11     setRequiredMethods
12       (setDeclaredMethods c (getAllMethods getDeclaredMethods c))
13       (getAllMethods getRequiredMethods c)
14   | otherwise =
15     setDeclaredMethods c (getAllMethods getDeclaredMethods c)
16
17 getAllMethods getter c =
18   | isDerived c =
19     (getter c) overrideUnion (getAllMethods getter (getSuperType c))
20   | otherwise = getter c
21
22 overrideUnion newMethods oldMethods =
23   newMethods ∪
24   (oldMethods \ ⋃ {oldMethods correspondingTo m | m ∈ newMethods})
25
26 correspondingTo ms m =
27   {m' | m' ∈ ms ∧ m ≠ m' ∧ getMethodName m = getMethodName m'}
```

Listing 2: Combinator type derivation for a given code fragment `c`

Semantic types of declared mixin methods can be conditional and during application each constellation of semantic implications of each declared method can be satisfied. We take this fact into account by intersecting the arrows built for every version of a mixin resulting from a combination drawn out of its methods with each method restricted to a single semantic annotation. These versions are computed by `enumerateDifferentSemantics`[1] from Listing 3.

[1] Sets of methods are always discrete and finite. Thus avoiding notational overhead, we can mix set and tuple notation for the Cartesian product Π choosing an arbitrary fixed element order.

```
enumerateDifferentSemantics c =
  { setMethods c ms |
    ms ∈ Π { toSingleSemantic m | m ∈ getDeclaredMethods c } }

toSingleSemantic m
  | getSemantics m ≠ ∅ = { setSemantics m { s } | s ∈ getSemantics m }
  | otherwise = { m }
```

Listing 3: Algorithm generating versions of the same mixin for each constellation of semantic types

Finally, records and arrows between records can be derived as types. For mixins `buildMixinArrow` from Listing 4 creates a new arrow. Methods which are neither declared nor required are propagated using the $\eta \setminus L$ notation from Section 3.

```
buildMixinArrow c =
  let methods = getAllDeclaredMethods c ∪ getAllRequiredMethods c
      forbiddenLabels = getMethodNames methods
  in □((η \ forbiddenLabels) ∩
        (⋂ { buildRecordEntry LHS c m | m ∈ methods }))
     → □((η \ forbiddenLabels) ∩
          (⋂ { buildRecordEntry RHS c m | m ∈ methods }))

buildRecordEntry side c m
  | ((isLHS side) ∧ (isMixin c) ∧ (m ∈ getDeclaredMethods c) ∧
     (conditional (getSemantics m) = ∅)) ∨
    ((isRHS side) ∧ (isMixin c) ∧ (m ∈ getRequiredMethods c) ∧
     ((getDeclaredMethods c) correspondingTo m ≠ ∅)) = ω
  | (isMixin c) ∧ (m ∈ getRequiredMethods c) ∧
    ((getDeclaredMethods c) correspondingTo m = ∅) =
    ⟨ getMethodName m : nativeType m ∩ β_m ∩ semanticTypes side c m ⟩
  | otherwise =
    ⟨ getMethodName m : nativeType m ∩ semanticTypes side c m ⟩

semanticTypes side c m
  | (isLHS side) ∧ (isMixin c) ∧ (m ∈ getRequiredMethods c) =
    ⋂ { getValue s | s ∈ getSemantics m }
  | (isLHS side) ∧ (isMixin c) =
    ⋂ { getCondition s | s ∈ conditional (getSemantics m) }
  | (isRHS side) ∧ (m ∈ (getDeclaredMethods c)) =
    (⋂ { getValue s | s ∈ getSemantics m }) ∩ (Type (getClassName c))
  | (isRHS side) ∧ (isMixin c) ∧ (m ∈ getRequiredMethods c) ∧
    ((getDeclaredMethods c) correspondingTo m = ∅) =
    semanticTypes LHS c m
  | otherwise = ω

conditional sems = { s | s ∈ sems ∧ isConditionalAnnotation s }

nativeType m =
  Type (intercalate "to" (getParameterTypes m + getResultType m))
```

Listing 4: Derivation of single record entries and arrows between records

According to Listing 4, occurring methods are explicitly treated by record entries created with `buildRecordEntry` which is also used to create entries for classes. There are three different cases how to build record entries: the default behavior (line 17) is to use the name of a method as label and to type it by a constant created from its parameters and result type (`nativeType`, line 34) together with a semantic intersection type (`semanticType`, line 20). Required methods which are not overwritten by a declaration additionally carry a propagation variable β in order to preserve their semantic type. Variable names are chosen for each method to prevent unwanted sharing. The third case arises for declarations without a semantic condition and for overwritten requirements. Conditionless declarations do not impose new requirements and therefore only add ω to the intersection on the left hand side of arrows. Overwritten requirements are gone after mixin application and only add ω to the right hand intersection. A similar case distinction is necessary for semantic types. On left hand sides they can originate from semantic annotations of required methods or conditions of declared methods. On right hand sides (also used for classes) declared methods receive semantic types from their annotations and are additionally intersected with a type indicating the class or interface they belong to. Required methods which are not overwritten just propagate their semantic type from the left hand side of the arrow. Overwritten requirements do not add semantic types, because their overall type on right hand sides is ω. To avoid partiality of `semanticTypes` their semantic type is also mapped to ω.

Application of `buildType` to `ByteOrderAdapter` and simplification of intersections by the rules $\sigma \cap \omega = \sigma$ and $\langle l : \sigma \rangle \cap \langle l : \tau \rangle = \langle l : \sigma \cap \tau \rangle$ results in the following combinator type (method type constants marked in blue, semantic types marked in orange):

$$
\begin{aligned}
&M_{\texttt{ByteOrderAdapter}} : \\
&\quad (\Box((\eta \setminus \{\texttt{process, swap}\}) \cap \langle \texttt{process} : \text{inttoint} \cap \text{LITTLE\_ENDIAN}, \\
&\qquad\qquad \texttt{swap} : \text{inttoint} \cap \beta_{\texttt{swap}} \cap \text{SWAPS\_BYTEORDER} \rangle) \\
&\quad \to \Box((\eta \setminus \{\texttt{process, swap}\}) \cap \langle \texttt{process} : \text{inttoint} \cap \text{BIG\_ENDIAN} \cap \text{ByteOrderAdapter}, \\
&\qquad\qquad \texttt{swap} : \text{inttoint} \cap \beta_{\texttt{swap}} \cap \text{SWAPS\_BYTEORDER} \rangle)) \cap \\
&\quad (\Box((\eta \setminus \{\texttt{process, swap}\}) \cap \langle \texttt{process} : \text{inttoint} \cap \text{BIG\_ENDIAN}, \\
&\qquad\qquad \texttt{swap} : \text{inttoint} \cap \beta_{\texttt{swap}} \cap \text{SWAPS\_BYTEORDER} \rangle) \\
&\quad \to \Box((\eta \setminus \{\texttt{process, swap}\}) \cap \langle \texttt{process} : \text{inttoint} \cap \text{LITTLE\_ENDIAN} \cap \text{ByteOrderAdapter}, \\
&\qquad\qquad \texttt{swap} : \text{inttoint} \cap \beta_{\texttt{swap}} \cap \text{SWAPS\_BYTEORDER} \rangle))
\end{aligned}
$$

The methods `swap` and `process` are explicitly required to apply the combinator. While $\beta_{\texttt{swap}}$ allows `swap` to continue to exist unchanged, the byte order semantic of `process` will be toggled and its originating class will be `ByteOrderAdapter`. All other methods can continue to be available by propagation via $\eta \setminus L$. Assuming we have a method body implementation `ByteOrderAdapter_process`, in principle we can state the L2 implementation of $M_{\texttt{ByteOrderAdapter}}$ as follows:

$$M_{\texttt{ByteOrderAdapter}} = \lambda x.\ \texttt{letbox}\ u = x\ \texttt{in}\quad \texttt{box}\ (u).\texttt{process} := \texttt{ByteOrderAdapter\_process}$$

In a practical implementation, we want to postprocess the synthesis outcome to Java code, which is why we choose to encode[2] record updates into XML augmented with bookkeeping information like package names. Using a repository $\mathscr{D}$ constructed from the example code in Listing 1 including a lifted combinator for `BaseProcess` we can ask an inhabitation question like

$$\emptyset; \mathscr{D} \vdash_{\text{C2}} ? : \Box\langle \texttt{process} : \text{intotint} \cap \text{LITTLE\_ENDIAN} \rangle$$

[2]Not presented here due to the limited amount of space available

One of the inhabitants of the requested type is

$$M_{\texttt{ByteOrderAdapter}}(M_{\texttt{ByteOrderSwapper}}(T_{\texttt{BaseProcess}})).$$

It reduces to L1 code including the necessary information to obtain the delegation based Java implementation shown in Listing 5 from postprocesing.

```
public class ByteOrderSwapperBaseProcess
    implements ByteOrderSwapper<ByteOrderSwapper.Requirement> {
  private class SelfBaseProcess
      extends BaseProcess
      implements ByteOrderSwapper.Requirement {}
  private final SelfBaseProcess self;
  public ByteOrderSwapperBaseProcess() { self = new SelfBaseProcess(); }
  public ByteOrderSwapper.Requirement getSelf() { return self; }
  public int process(int input) { return self.process(input); } }
public class ByteOrderAdapterByteOrderSwapperBaseProcess
    implements ByteOrderAdapter<ByteOrderAdapter.Requirement> {
  private class SelfByteOrderSwapperBaseProcess
      extends ByteOrderSwapperBaseProcess
      implements ByteOrderAdapter.Requirement {
    public int process(int input) { return super.process(input); }
    public int swap(int value) { return super.swap(value); } }
  private final SelfByteOrderSwapperBaseProcess self;
  public ByteOrderAdapterByteOrderSwapperBaseProcess() {
    self = new SelfByteOrderSwapperBaseProcess();
  }
  public ByteOrderAdapter.Requirement getSelf() { return self; }
  public int swap(int value) { return self.swap(value); } }
```

Listing 5: Synthesized implementation of an inhabitant for $\square\langle\texttt{process} : \text{intotint} \cap \text{LITTLE\_ENDIAN}\rangle$

In Listing 5 we created a new class for each mixin application. Each of those classes implements the interface of the applied mixin. They fill in `getSelf` by providing an instance of the applicant that also implements the necessary requirement interface. The latter is always possible due to the construction of our mixin typing rules. Additionally, newly created classes delegate all propagated methods to the instance returned by `getSelf`. Class names are computed by concatenating mixin names. Executing

```
(new ByteOrderAdapterByteOrderSwapperBaseProcess()).process(0xC0CAC01A)
```

first executes `process` of `BaseProcess` resulting in `0xADFBCADE` and then applies `swap` from `ByteOrderSwapper`, which results in the expected `0xDECAFBAD`.

5 Conclusion and Future Work

We presented an approach to synthesize mixin compositions from specifications encoded in Java 8 interfaces adhering to a design pattern. Our notion of mixins can add and replace methods in existing classes. We sketched out an algorithm to translate specifications to meaningful and well typed combinators for staged composition synthesis. Our combinator language is backed by an intersection typed version of the lambda calculus with records. The synthesis process is guided by user provided semantic type information incorporated via intersections. It results in a series of mixin combinator applications

inhabiting a requested type. We gave an example in which these applications were transformed to Java code, compensating missing language features by delegation.

Since the underlying type system $\mathbb{T}_{\langle\rangle}$ relies on intersection, we cannot apply type erasure as defined in [7]. Therefore, we need to extend the staged composition synthesis approach in our future work to prove the conservative extension theorem (implementation type correctness). Moreover, the staged composition synthesis framework needs to handle the introduced $\eta \setminus L$ abbreviation.

Currently, we abstract from particular implementations of method bodies and rely heavily on the meaningfulness of (user defined) semantic types. While this is general and applicable to different languages, it requires careful consideration regarding correctness.

It may be useful to further analyze and incorporate types of given method bodies in the synthesis process. This particular aspect of mixin composition synthesis is being actively developed by Ugo de'Liguoro and provides great potential for future collaboration. It may also help to identify and extend the supported subset of Java 8, e. g. allowing method overloading.

Complementary to formal aspects an empirical evaluation will be an important future task. It remains an interesting question if and how exactly our system can be used to improve code quality, especially with respect to our main goal of enhanced modularity.

References

[1] Davide Ancona, Giovanni Lagorio & Elena Zucca (2000): *Jam - A Smooth Extension of Java with Mixins*. In Elisa Bertino, editor: *ECOOP*, *Lecture Notes in Computer Science* 1850, Springer, pp. 154–178.

[2] 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.

[3] Gilad Bracha & William R. Cook (1990): *Mixin-based Inheritance*. In Akinori Yonezawa, editor: *OOPSLA/ECOOP*, ACM, pp. 303–311.

[4] Rowan Davies & Frank Pfenning (2001): *A Modal Analysis of Staged Computation*. *Journal of the ACM* 48(3), pp. 555–604.

[5] Ugo de'Liguoro (2001): *Characterizing Convergent Terms in Object Calculi via Intersection Types*. In: *TLCA*, pp. 315–328.

[6] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts & Andrew P. Black (2006): *Traits: A mechanism for fine-grained reuse*. *ACM Trans. Program. Lang. Syst.* 28(2), pp. 331–388.

[7] Boris Düdder, Moritz Martens & Jakob Rehof (2014): *Staged Composition Synthesis*. In: *Proceedings of ESOP'14*, *LNCS* 8410, Springer, pp. 67–86.

[8] Matthew Flatt, Shriram Krishnamurthi & Matthias Felleisen (1998): *Classes and Mixins*. In David B. MacQueen & Luca Cardelli, editors: *POPL*, ACM, pp. 171–183.

[9] James Gosling, Bill Joy, Guy Steele, Gilad Bracha & Alex Buckley (2014): *The Java® Language Specification - Java SE 8 Edition*. Available at `http://docs.oracle.com/javase/specs/`.

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

[11] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz & Andrew P. Black (2003): *Traits: Composable Units of Behaviour*. In Luca Cardelli, editor: *ECOOP*, *Lecture Notes in Computer Science* 2743, Springer, pp. 248–274.