Science of Computer Programming 78 (2013) 521–541

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

TraitRecordJ: A programming language with traits and records☆

Lorenzo Bettini [a,*], Ferruccio Damiani [a], Ina Schaefer [b], Fabio Strocco [a]

[a] Dipartimento di Informatica, Università di Torino, Italy

[b] Technische Universität Braunschweig, Germany

ARTICLE INFO

Article history:
Available online 22 July 2011

Keywords:
Java
Trait
Type system
Implementation
Eclipse

ABSTRACT

Traits have been designed as units for fine-grained reuse of behavior in the object-oriented paradigm. Records have been devised to complement traits for fine-grained reuse of state. In this paper, we present the language TraitRecordJ, a Java dialect with *records* and *traits*. Records and traits can be composed by explicit linguistic operations, allowing code manipulations to achieve fine-grained code reuse. Classes are assembled from (composite) records and traits and instantiated to generate objects. We introduce the language through examples and illustrate the prototypical implementation of TraitRecordJ using Xtext, an Eclipse framework for the development of programming languages as well as other domain-specific languages. Our implementation comprises an Eclipse-based editor for TraitRecordJ with typical IDE functionalities, and a stand-alone compiler, which translates TraitRecordJ programs into standard Java programs. As a case study, we present the TraitRecordJ implementation of a part of the software used in a web-based information system previously implemented in Java.

1. Introduction

The term *trait* was used by Ungar et al. [66] in the dynamically-typed prototype-based language Self to denote a parent object to which an object may delegate some of its behavior. Subsequently, traits have been introduced by Schärli et al. [60,33] in the dynamically-typed class-based language Squeak/Smalltalk [17,18] to play the role of *units for fine-grained reuse of behavior*, in order to counter the problems of class-based inheritance with respect to code reuse (see, e.g., [33,48,26] for discussions and examples). A trait is a set of methods, completely independent from any class hierarchy. The common behavior (i.e., the common methods) of a set of classes can be factored into a trait, and traits can be composed to form other traits or classes. Two distinctive features of traits are that traits can be composed in arbitrary order and that composed traits or classes must resolve possible name conflicts explicitly. These features make traits more flexible and simpler than *mixins* [39,23,45,35,8]. Various formulations of traits in a Java-like setting can be found in the literature (see, e.g., [61,49,20,58,21,46]). The recent programming language Fortress [7] incorporates a form of trait construct, while the "trait" construct incorporated in Scala [50] is indeed a form of mixin.

Records have been proposed by Bono et al. [20] as the counterpart of traits, with respect to state, to play the role of *units for fine-grained reuse of state*. A record is a set of fields, completely independent from any class hierarchy. The common state (i.e., the common fields) of a set of classes can be factored into a record.

☆ This work has been partially supported by the Ateneo Italo-Tedesco/Deutsch-Italienisches Hochschulzentrum (Vigoni project "Language constructs and type systems for object oriented program components"), the Deutsche Forschungsgemeinschaft (DFG), the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (http://www.hats-project.eu), and the MIUR PRIN 2009 project DISCO: Distribution, Interaction, Specification, Composition for Object Systems.

* Corresponding author.
E-mail address: bettini@di.unito.it (L. Bettini).

0167-6423/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2011.06.007

In this paper, we present the programming language TraitRecordJ,[1] a Java dialect using traits and records as composable units of behavior and state reuse, respectively, and aiming at interface-based polymorphism. TraitRecordJ is based on the calculus presented in [13] whose complete formalization and proof of type safety is available as a technical report in [14]. In TraitRecordJ, the declarations of object type, state, behavior and instance generation are completely separated. TraitRecordJ considers:

- *Interfaces*, as pure types, defining only method signatures;
- *Records*, as pure units of state reuse, defining only fields;
- *Traits*, as pure units of behavior reuse, defining only methods; and
- *Classes*, as pure generators of instances, implementing interfaces by using traits and records, and defining constructors.

In TraitRecordJ, a trait consists of *provided methods* (i.e., methods defined in the trait), of *required methods* which parametrize the behavior, and of *required fields* that can be directly accessed in the body of the methods. Traits are building blocks to compose classes and other, more complex, traits. A suite of trait composition operations allows the programmer to build classes and composite traits. The complete syntax and features of TraitRecordJ will be illustrated in Section 2. Here we briefly illustrate some features of TraitRecordJ with an example.

Example 1. Consider the task of developing a class `Stack` that implements the interface:

```
interface IStack {
  boolean isEmpty();
  void push(Object o);
  Object pop();
}
```

In Java, the corresponding implementation would be a class as follows:

```
class Stack implements IStack {
   List l;
   Stack() { l = new ArrayList(); }
   boolean isEmpty() { return (l.size() == 0); }
   void push(Object o) { l.addFirst(); }
   Object pop() { Object o = l.getFirst(); l.removeFirst(); return o; }
}
```

In a language with traits and records, the fields and the methods of the class can be defined independently from the class itself. If the class `Stack` was originally developed in TraitRecordJ, it would have been written as follows (the interface `IList` and the class `CArrayList`, not shown here, have all the standard methods of lists and are part of the library of the language; they correspond to `List` and `ArrayList` in Java):

```
record RElements is { IList l; }

trait TStack is {
   IList l;   /* required field */
   boolean isEmpty() { return (l.size() == 0); }
   void push(Object o) { l.addFirst(o); }
   Object pop() { Object o = l.getFirst(); l.removeFirst(); return o; }
}

class Stack implements IStack by RElements and TStack {
   Stack() { l = new CArrayList(); }
}
```

In TraitRecordJ, like in Fortress, there is no class hierarchy and consequently no class-based inheritance. Multiple inheritance with respect to methods is obtained via the trait construct, and multiple inheritance with respect to fields is obtained via the record construct. Thus, multiple inheritance is subsumed by ensuring that, in the spirit of the original trait proposal in Squeak/Smalltalk, the composite unit has complete control over the composition and must resolve conflicts explicitly.

Although TraitRecordJ rules out class-based inheritance and relies on traits and records as the reuse mechanisms, class-based inheritance could be encoded with the existing concepts of TraitRecordJ as a syntactic sugar construct, as we will explain in Section 4. However, this "shortcut" might undermine the code reuse potential provided by traits and records leading back to the pitfalls of standard class-based inheritance. Furthermore, the introduction of class-based inheritance in

[1] TraitRecordJ was formerly called Sugared Welterweight Record-Trait Java (SWRTJ) [15]. We adopted a new name since the old one turns out to sound rather odd. The genesis of the old name of the language, which dates back to an early prototype developed as part of the fourth author's bachelor thesis [65], is quite involved. Welterweight refers to the fact that, in a preliminary design, the "weight" of the language was meant to lay between the weight of Lightweight Java [64] and Middleweight Java [16], while sugared expressed the fact the implemented language contained some syntactic sugar not present in the underlying calculus [13,14].

Table 1
TraitRecordJ syntax.

ID	::=	interface I extends $\overline{\mathtt{I}}$ { $\overline{\mathtt{S}}$; }	interface definition
S	::=	I m($\overline{\mathtt{I}}$ $\overline{\mathtt{x}}$)	method signature
RD	::=	record R is RE	record definition
RE	::=	{ $\overline{\mathtt{F}}$; } | R | RE + RE | RE[$\overline{\mathtt{RO}}$]	record expression
F	::=	I f	field definition
RO	::=	exclude f | f renameTo f	record field operation
TD	::=	trait T is TE	trait definition
TE	::=	{ $\overline{\mathtt{F}}$; $\overline{\mathtt{S}}$; $\overline{\mathtt{M}}$ } | T | TE + TE | TE[$\overline{\mathtt{TO}}$]	trait expression
M	::=	S {$\overline{\mathtt{I}}$ $\overline{\mathtt{x}}$ = $\overline{\mathtt{e}}$; $\overline{\mathtt{se}}$; return e; }	method
e	::=	null | x | x = e | this | this.f | this.f = e | e.m($\overline{\mathtt{e}}$) | new C($\overline{\mathtt{e}}$) | (I)e	expression
se	::=	x = e | this.f = e | e.m($\overline{\mathtt{e}}$) | if (e) be else be | while (e) be	statement expression
be	::=	{$\overline{\mathtt{I}}$ $\overline{\mathtt{x}}$ = $\overline{\mathtt{e}}$; $\overline{\mathtt{se}}$; }	block expression
TO	::=	exclude m | f renameFieldTo f | m renameTo m | m aliasAs m	trait elements operation
CD	::=	class C implements $\overline{\mathtt{I}}$ by RE and TE { $\overline{\mathtt{K}}$ }	class definition
K	::=	C($\overline{\mathtt{I}}$ $\overline{\mathtt{x}}$) be	constructor

TraitRecordJ is not necessary, since a TraitRecordJ program can be used in conjunction with any standard Java program. TraitRecordJ is a Java dialect, and the TraitRecordJ compiler generates standard Java classes that do not depend on a specific library. Thus, a Java program can reuse TraitRecordJ interfaces and create standard Java objects using the code generated by the TraitRecordJ compiler. This technique can be used to combine code written in TraitRecordJ with Java libraries whose sources cannot be changed. For instance, TraitRecordJ objects can be passed to standard Java code provided the respective interfaces are compatible. The possibility of combining TraitRecordJ generated Java code with standard Java code can also be used to incrementally refactor Java class hierarchies into trait and record-based code. The refactoring can be done along the lines of the work by Bettini et al. [12], where an approach for identifying the methods in a Java class hierarchy that can be good candidates to be refactored in traits is presented.

TraitRecordJ is equipped with a Java-like nominal type system where the only user-defined types are interface names. This supports type checking traits in isolation from other traits and classes that use them. In Scala [50] and Fortress [7], each trait, as well as each class, also defines a type. But the role of unit of reuse and the role of type are competing, as already pointed out by Snyder [62] and Cook et al. [31]. In order be able to define the subtyping relation on traits such that a trait is always a subtype of the component traits, both Scala and Fortress rule out the method exclusion operation. (Method exclusion forms a new trait by removing a method from an existing trait.) However, without method exclusion, the reuse potential of traits is restricted. In TraitRecordJ, traits (as well as records and classes) do not define types. Thus, in TraitRecordJ method exclusion is supported and enhances the reuse potential of traits. This will be illustrated in Example 2 of Section 2.1.

We present the implementation of TraitRecordJ using Xtext [5], a framework for the development of programming languages as well as other domain-specific languages (DSLs). The syntax of TraitRecordJ is defined in Xtext using an EBNF grammar. The Xtext generator creates a parser, an AST-meta model (implemented in EMF [63]) as well as a full-featured Eclipse-based editor for TraitRecordJ. Using the TraitRecordJ compiler, a TraitRecordJ program is translated into a standard Java program. The TraitRecordJ compiler can also be used as a command line program, outside of the Eclipse IDE. The implementation of TraitRecordJ described in the paper is available as an open source project at http://traitrecordj.sourceforge.net.

A preliminary version of this paper appeared as [15]. The current version of the paper includes new examples, presents a case study based on a commercially used application software, and provides a more detailed presentation of the TraitRecordJ implementation.

Organization of the Paper. Section 2 presents the syntax of TraitRecordJ and briefly introduces its operational semantics and type system. Section 3 illustrates the language by means of an example. In Section 4 we discuss the features of TraitRecordJ and their impact on programming. Section 5 presents the case study. Section 6 describes the implementation of TraitRecordJ using the Xtext framework. Related work is discussed in Section 7. We conclude by outlining possible directions for future work.

2. The TraitRecordJ programming language

In this section we describe the syntax of the TraitRecordJ programming language and briefly illustrate its operational semantics and type system (we refer to [14] for the formalization of the FRTJ calculus on which TraitRecordJ is based).

2.1. Syntax

The syntax of TraitRecordJ is illustrated in Table 1, without `void`, primitive types (`boolean`, `byte`, `char`, `short`, `int`, `long`, `float` and `double`) and file imports, which are not relevant for this description. An interface must extend zero or more interfaces, whereas a class must implement one or more interfaces. In the syntax, the overline bar indicates a (possibly empty) list, as in Featherweight Java [40]. For instance, $\overline{\mathtt{I}} = \mathtt{I}_1, \ldots, \mathtt{I}_n,\ n \geq 0$. The parameter declarations are denoted by $\overline{\mathtt{I}}\ \overline{\mathtt{x}}$ to indicate $\mathtt{I}_1\ \mathtt{x}_1, \ldots, \mathtt{I}_n\ \mathtt{x}_n$. The same notation can be applied to the other lists. In the syntax, `m` denotes a method name, `f` a field name and `x` a local variable or parameter. Note that the grammar contains only the field access `this.f` (even for field assignment) because a field can be used only within a method (trait method or constructor) due to its private visibility. Variables and methods can only have an interface type. Trait expressions in trait composition operations do not have to be names of already defined traits: they can also be "anonymous" traits $\{\ \overline{\mathtt{F}};\ \overline{\mathtt{S}};\ \overline{\mathtt{M}}\ \}$. The same holds for record expressions. This is in particular useful for traits, when a trait expression has to define some "glue" code for other traits used in a trait composition operation. In this way, there is no need to define a trait with a name only for that. Examples of such usage have been given in Section 1.

In TraitRecordJ, a trait consists of *provided methods* (i.e., methods defined in the trait), of *required methods* which parametrize the behavior, and of *required fields* that can be directly accessed in the body of the methods, along the lines of the proposal by Bono et al. [20] (see also [13,21]). Traits are building blocks to compose classes and other, more complex, traits. A suite of trait composition operations allows the programmer to build classes and composite traits. A distinguished characteristic of traits is that the composite unit (class or trait) has complete control over conflicts that may arise during composition and must solve them explicitly. Traits do not specify any state. Therefore a class composed by using traits has to provide the required fields through records, as explained in the following. The trait composition operations considered in TraitRecordJ are as follows:

- A basic trait defines a set of methods and declares the required fields and the required methods.
- The *symmetric sum* operation, $+$, merges two traits to form a new trait. It requires that the summed traits must be disjoint (that is, they must not provide identically named methods) and have compatible requirements (two requirements on the same method/field name are compatible if they are identical).
- The operation `exclude` forms a new trait by removing a method from an existing trait.
- The operation `aliasAs` forms a new trait by giving a new name to an existing method. When a recursive method is aliased, its recursive invocation refers to the original method (as proposed by Schärli et al. [60,33]). The variant of method aliasing where when a recursive method is aliased its recursive invocation refers to the new method (proposed by Liquori and Spiwack [46]), can be straightforwardly encoded by exclusion, renaming and symmetric sum.
- The operation `renameTo` creates a new trait by renaming all the occurrences of a required/provided method name from an existing trait (the return type and the parameters of the method are unchanged).
- The operation `renameFieldTo` creates a new trait by renaming all the occurrences of a required field name.

Note that, in TraitRecordJ, the actual names of the methods defined in a trait and the names of the required methods and fields are irrelevant, since they can be changed by the `renameTo` and `renameFieldTo` operations. While developing the case study illustrated in Section 5, we found it natural to exploit these operations (which are not supported, e.g., in the languages Smalltalk/Squeak, Fortress and Scala).

Records are building blocks to compose classes and other, more complex, records by means of operations analogous to the ones described above for traits. The record composition operations considered in TraitRecordJ are as follows:

- A basic record defines a set of fields.
- The *symmetric sum* operation, +, merges two disjoint records to form a new record.
- The operation `exclude` forms a new record by removing a field from a record.
- The operation `renameTo` creates a new record by renaming a field in a record.

TraitRecordJ fosters the programming methodology based on design by interface, starting from declaring the services (methods) that our implementation components provide using a separate mechanism (the interface) which has only that aim; this naturally turns the interface in the only means to declare a type. The state and the implementations of such services are once again split into two different entities which can be reused separately, i.e., records and traits, respectively. Note that these blocks, records and traits, are pure units of reuse, and cannot be instantiated directly (this would undermine their usability). However, they can be used by the classes, which are the only means to assemble records and traits in order to implement interfaces and to instantiate objects.

The entry point of a TraitRecordJ program is specified by

```
program <name> {
  <expression>
}
```

In the scope of the program, the implicit object `args` is the list of the program's command line arguments.

In TraitRecordJ, traits, records and classes are not types in order to subdivide the roles of the different constructs. Moreover, as illustrated in the following Example 2, traits and records support exclusion operations that violate subtyping. Therefore, a type can be only an interface or a primitive type (e.g., int, boolean etc.).

Example 2. Consider the JAVA version of the class `Stack` introduced in Example 1. Suppose that a class implementing the following interface should be developed:

```
interface ILifo {
  boolean isNotEmpty();
  void push(Object o);
  void pop();
  Object top();
}
```

In JAVA there is no straightforward way to reuse the code in class `Stack`, as it would not be possible to override the `pop` method changing the return type from `Object` to `void`.

Consider now the TRAITRECORDJ version of the class `Stack` (in Example 1). The record `RElements` and the trait `TStack` are completely independent from the code of class `Stack`. TRAITRECORDJ extends method reusability of traits to state reusability of records and fosters a programming style relying on small components that are easy to reuse. Because of the trait and record operations to exclude and rename methods and fields, they can be reused to develop completely unrelated classes. Based on this, a programmer would be able to write a class `Lifo` implementing the interface `ILifo` by reusing the record `RElements` and by defining a trait `TLifo` that reuses the trait `TStack` by exploiting the method exclusion operation:

```
trait TLifo is (TStack[exclude pop]) + {
  IList l; /* required field */
  boolean isEmpty(); /* required method */
  boolean isNotEmpty() { return !isEmpty(); }
  void pop() { l.removeFirst(); }
  Object top() { return l.getFirst(); }
}

class Lifo implements ILifo by RElements and TLifo {
  Lifo() { l = new CArrayList(); }
}
```

The body of trait `TLifo` satisfies the requirements of the trait sum operation described at the beginning of the section: the method `pop` is excluded thus it does not generate a conflict and the field requirement is identical to the one of `TStack`.

The example above illustrates a paradigmatic case of trait composition that does not preserve structural subtyping. If traits were types and composed traits were subtypes of the component traits (as in SCALA and FORTRESS), then the declaration of the trait `TLifo` would not type check, since:

- the trait `TLifo` should provide all the methods provided by `TStack`, and
- a method with signature `Object pop()` and a method with signature `void pop()` could not belong to the same trait/class.

Alternatively, the trait `TLifo` could be implemented by exploiting method renaming instead of method exclusion:

```
trait TLifo is (TStack[pop renameTo old_pop]) + {
  IList l; /* required field */
  Object old_pop(); /* required method */
  boolean isEmpty(); /* required method */
  boolean isNotEmpty() { return !isEmpty(); }
  void pop() { old_pop(); }
  Object top() { return l.getFirst(); }
}
```

Again, if traits were types and composed traits were subtypes of the component traits, then the declaration of the trait `TLifo` would not type check.

For simplicity, method overloading is not supported. Constructor overloading allows only the definition of constructors with different numbers of parameters within a class.

Visibility modifiers are ruled out since they are not necessary. The traditional visibility modifiers are implied by the constructs used in TRAITRECORDJ as follows.

private — Every instance variable is private. Since a class is not a type, fields can be accessed only from the `this` parameter. Every provided method is private if it does not appear in the interfaces implemented by a class. Interfaces are the only way to make a method accessible from the outside.

protected — This modifier is not necessary since inheritance is ruled out.

public — Every method declared in the interface implemented by a class is public. Fields cannot be public in order to support information hiding.

Table 2
Flattening TraitRecordJ to Java.

$$
\begin{array}{rcl}
[\![\texttt{class C implements } \bar{\texttt{I}} \texttt{ by RE and TE } \{ \bar{\texttt{K}} \}]\!] & = & \texttt{class C implements } \bar{\texttt{I}} \ \{[\![\texttt{RE}]\!]\ \bar{\texttt{K}}\ [\![\texttt{TE}]\!]\ \} \\
 & & \\
[\![\{ \bar{\texttt{F}};\ \}]\!] & = & \bar{\texttt{F}} \\
[\![\texttt{R}]\!] & = & [\![\texttt{RE}]\!] \quad \textbf{if } \texttt{RT(R)} = \texttt{record R is RE} \\
[\![\texttt{RE}_1 + \texttt{RE}_2]\!] & = & [\![\texttt{RE}_1]\!] \cdot [\![\texttt{RE}_2]\!] \\
[\![\texttt{RE[exclude f]}]\!] & = & \mathit{exclude}([\![\texttt{RE}]\!], \texttt{f}) \\
[\![\texttt{RE[f renameTo f}']\!] & = & [\![\texttt{RE}]\!][\texttt{f}'/\texttt{f}] \\
 & & \\
[\![\{ \bar{\texttt{F}};\ \bar{\texttt{S}};\ \bar{\texttt{M}} \}]\!] & = & \bar{\texttt{M}} \\
[\![\texttt{T}]\!] & = & [\![\texttt{TE}]\!] \quad \textbf{if } \texttt{TT(T)} = \texttt{trait T is TE} \\
[\![\texttt{TE}_1 + \texttt{TE}_2]\!] & = & [\![\texttt{TE}_1]\!] \cdot [\![\texttt{TE}_2]\!] \\
[\![\texttt{TE[exclude m]}]\!] & = & \mathit{exclude}([\![\texttt{TE}]\!], \texttt{m}) \\
[\![\texttt{TE[m aliasAs m}']\!] & = & \bar{\texttt{M}} \cdot (\texttt{I m}'(\bar{\texttt{I}}\ \bar{\texttt{x}})\ \texttt{MB}) \quad \textbf{if } [\![\texttt{TE}]\!] = \bar{\texttt{M}} \textbf{ and } \texttt{I m}(\bar{\texttt{I}}\ \bar{\texttt{x}})\ \texttt{MB} \in \bar{\texttt{M}} \\
[\![\texttt{TE[f renameFieldTo f}']\!] & = & [\![\texttt{TE}]\!][\texttt{f}'/\texttt{f}] \\
[\![\texttt{TE[m renameTo m}']\!] & = & \mathit{mR}([\![\texttt{TE}]\!], \texttt{m}, \texttt{m}') \\
 & & \\
\mathit{mR}(\texttt{I n}(\bar{\texttt{I}}\ \bar{\texttt{x}})\ \texttt{MB}, \texttt{m}, \texttt{m}') & = & \texttt{I n}[\texttt{m}'/\texttt{m}](\bar{\texttt{I}}\ \bar{\texttt{x}})\ \texttt{MB}[\texttt{this.m}'/\texttt{this.m}] \\
\mathit{mR}(\texttt{M}_1 \cdot \ldots \cdot \texttt{M}_n, \texttt{m}, \texttt{m}') & = & (\mathit{mR}(\texttt{M}_1, \texttt{m}, \texttt{m}')) \cdot \ldots \cdot (\mathit{mR}(\texttt{M}_n, \texttt{m}, \texttt{m}'))
\end{array}
$$

The system library of TraitRecordJ provides interfaces such as `IObject` (implicitly extended by every interface), `IInteger`, `IString`, etc. Every interface has a corresponding class implementing it, e.g., `CObject`, `CInteger`, etc. For synchronization mechanisms, we provide the interface `ILock` (and the corresponding class `CLock`) with methods `lock` and `unlock` avoiding to introduce specific concurrency features in the language. In order to deal with collections, `IList` is an interface with typical list operations. `CArrayList` is the corresponding implementation class. The interfaces `IPrintStream` and `IScanner`, and their implicit "global" instances `out` and `in` can be used to perform basic operations such as writing to standard output and reading from standard input, respectively. Standard basic types, such as `int`, `boolean`, etc., are also provided. Methods can be declared as `void` with the usual meaning.

The TraitRecordJ compiler, as illustrated in Section 6, produces standard Java code, which does not need any additional libraries to be compiled and executed. Our code generation phase basically implements the flattening procedure sketched in Section 2.2. The above interfaces and classes of the system library of TraitRecordJ are basically "stubs" for the actual Java corresponding library classes. Thus, the programmer can easily mix TraitRecordJ code with Java code in a program, since the former will be translated into plain Java.

2.2. Operational semantics

The semantics of TraitRecordJ conforms to the *flattening principle*, that has been introduced in the original formulation of traits in Squeak/Smalltalk [33] (see also [49,43,42]) in order to provide a canonical semantics for traits. According to the flattening principle, the semantics of a method/field introduced in a class by a trait/record is identical to the semantics of the same method/field defined directly within the class.

The semantics of TraitRecordJ is specified through a flattening function $[\![\cdot]\!]$, given in Table 2, that translates a TraitRecordJ class declaration into a Java class declaration, a record expression into a sequence of fields declarations, and a trait expression into a sequence of method declarations. The clauses in Table 2 are mostly self-explanatory. The auxiliary function *exclude* is such that $\mathit{exclude}(\bar{\texttt{F}}, \texttt{f})$ denotes the sequence of field declarations obtained from $\bar{\texttt{F}}$ by removing the declaration of the field `f`, and $\mathit{exclude}(\bar{\texttt{M}}, \texttt{m})$ denotes the sequence of method declarations obtained from $\bar{\texttt{M}}$ by removing the declaration of the method `m`. Note that, in the translation of trait expressions, the clause for field renaming is simpler than the clause for method renaming (which uses the auxiliary function *mR*); this is due to the fact that fields can be accessed only on `this`.

Example 3. The flattening of the versions of the class `Lifo` given in Example 2 produces the class

```
class Lifo implements ILifo {
  IList l;
  Lifo() { l = new CArrayList(); }
  boolean isEmpty() { return (l.size() == 0); }
  boolean isNotEmpty() { return !isEmpty(); }
  void push(Object o) { l.addFirst(); }
  void pop() { l.removeFirst(); }
  Object top() { return l.getFirst(); }
}
```

and the flattening of the versions of the class `Lifo` obtained by using the version of the trait `TLifo` given immediately after Example 2 produces the class

```
class Lifo implements ILifo {
  IList l;
```

```
  Lifo() { l = new CArrayList(); }
  Object old_pop() { Object o = l.getFirst(); l.removeFirst(); return o; }
  boolean isEmpty() { return (l.size() == 0); }
  boolean isNotEmpty() { return !isEmpty(); }
  void push(Object o) { l.addFirst(); }
  void pop() { old_pop(); }
  Object top() { return l.getFirst(); }
}
```

2.3. Type system

The TraitRecordJ type system supports type checking traits in isolation from classes and other traits that use them, and supports type checking records in isolation from classes and other records that use them. Therefore, each trait and record definition has to be type checked only once, i.e., every class or trait can use any trait or record without type checking it again, and every class or record can use any record without type checking it again. This is more efficient and convenient in practice, e.g., if the trait/record source is not available. The basic idea of the type system is to collect constraints when checking traits and records, and to establish that these constraints hold when a class is declared, in order to ensure that the pseudo-variable `this` in all the methods of used traits can be used safely.

In the TraitRecordJ type system, a nominal type is either a class name or an interface name. The *subinterfacing relation* is the reflexive and transitive closure of the immediate subinterfacing relation declared by the `extends` clauses in the interface definitions. The *subtyping relation* for nominal types is the reflexive and transitive closure of the relation obtained by extending subinterfacing with the interface implementation relation declared by the `implements` clauses in the class definitions.

Note that in TraitRecordJ classes are not source types. Class names cannot be used as types in the code written by the programmer: they are used only internally by the compiler to type object creation expressions (`new C(···)`). The type of `this` is an inferred structural type, used only internally by the compiler. The syntax for expression types is as follows $\theta ::= \mathtt{I} \mid \mathtt{C} \mid \langle \overline{\mathtt{F}} \mid \overline{\sigma} \rangle$ where $\mathtt{I}$ is an interface name, $\mathtt{C}$ is a class name and the $\langle \overline{\mathtt{F}} \mid \overline{\sigma} \rangle$ is the structural type for `this`, which contains all the fields ($\overline{\mathtt{F}}$) and the signatures ($\overline{\sigma}$) of the methods that can be selected on `this` in the context where the expression occurs. If the expression is a constructor call, such as `new C()`, its type is the class `C`; if the expression is `this`, its type is $\langle \overline{\mathtt{F}} \mid \overline{\sigma} \rangle$; otherwise, the type of the method is an interface, for instance, if the expression is a method call, such as `x.m()`, the type is the interface that `I` declares as return type of the method `m`.

The TraitRecordJ type system analyzes each trait (in isolation from classes and other traits that use it) and infers a constraint (used only internally by the compiler) for each method provided by the trait. A constraint is a triple $\langle \overline{\mathtt{F}} \mid \overline{\mathtt{S}} \mid \overline{\mathtt{I}} \rangle$ consisting of required fields, method signatures and interfaces collected while analyzing an expression. The constraints contain the types of every field and method selected on `this` and the name of every interface used, either as type in the methods parameters to which `this` is passed as argument or as return type in the methods in which `this` is returned. When analyzing a class, the TraitRecordJ type system checks that all the constraints of the methods in used traits are satisfied in the class.

Example 4. The constraints inferred by the TraitRecordJ type system for the methods provided by the version of the trait `TLifo` given immediately after Example 2 are as follows:

```
Object old_pop() : < IList l | | >
boolean isEmpty() : < IList l | | >
boolean isNotEmpty() : < | boolean isEmpty() | >
void push(Object o) : < IList l | | >
void pop() : < | Object old_pop() | >
Object top() : < IList l | | >
```

3. TraitRecordJ by example

In this section, we explain the use of TraitRecordJ using a comprehensive example. We implement a bank account with additional authentication and billing functionalities. This example is an adapted version from the example used in [32]. In that paper, the implementation of these bank account functionalities illustrates the subtle difficulties introduced by multiple inheritance. Here, we show that the trait and record composition operations of TraitRecordJ provide flexible means for code reuse, while alleviating the problems of multiple inheritance.

The class hierarchy of an implementation of the bank account example in a language supporting class-based multiple inheritance is depicted in Fig. 1. The base class is the class `CAccount` which implements basic functionalities for depositing and withdrawing money from an account. The balance of the account is stored in the field `bal` and the owner of the account in the field `owner` (both are represented by integers for simplicity). The balance of the account is changed by the `update` method. The `validate` method checks that the account is accessed by its owner. Listing 1 shows the implementation of the account class `CAccount` in TraitRecordJ. The `bal` and the `owner` fields are provided by the record `RAccount`. The methods of the class are introduced by the trait `TAccount`. The class `CAccount` is assembled from the record `RAccount` and the trait `TAccount`.

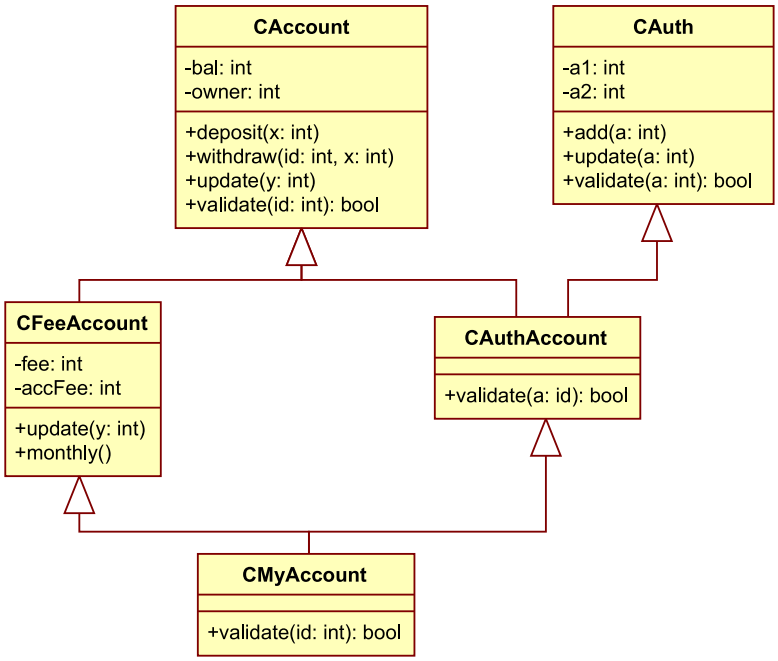

Fig. 1. Class diagram of the account example.

```
interface IAccount {
  void deposit(int x);
  void withdraw(int id, int x);
}

record RAccount is {
  int bal; //Provided field
  int owner;//Provided field
}

trait TAccount is {
  int owner; //Required field
  int bal; //Required field

  void deposit(int x) {this.update(x);}

  void update(int y) {this.bal = this.bal + y;}

  void withdraw(int id, int x){
    boolean v = this.validate(id);
    if (v) {this.update(−x);}
  }

  boolean validate(int id) {
    boolean b = false;
    if (this.owner == id){b = true;}
    return b;
  }
}

class CAccount implements IAccount by RAccount and TAccount {
  CAccount() {
     this.bal = 0;
     this.owner = 0;
  }
}
```

Listing 1. Implementation of class `CAccount`.

```
interface IAuth { }

record RAuth is {
   int a1;
   int a2;
}

trait TAuth is {
   int a1;
   int a2;

   void update(int a) {this.a2 = this.a1; this.a1 = a;}

   void add(int a) {
      if (this.a1 != −1) {this.update(a);}
      else {this.a1 = a;}
   }

   boolean validate(int a){
      boolean b = (a == this.a1) || (a == this.a2));
      return  b;
   }
}

class CAuth implements IAuth by RAuth and TAuth {
   CAuth() {
      this.a1 = −1;
      this.a2 = −1;
   }
}
```

Listing 2. Implementation of class `CAuth`.

The class `CAuth` is independent of class `CAccount`. It provides the functionality for storing two client identities in the fields `a1` and `a2`. Listing 2 shows the implementation of the class `CAuth` in TraitRecordJ. The fields are introduced by the record `RAuth`. The methods `update` for updating the stored identities, `add` for adding an identity and `validate` for validating a stored identity are provided by the trait `TAuth`. The record `RAuth` and the trait `TAuth` are composed in the class `CAuth`.

In the class diagram in Fig. 1, the class `CAuthAccount` inherits its functionalities from the classes `CAccount` and `CAuth` such that it can provide more sophisticated authentication facilities by reusing the fields of the class `CAuth`. In class `CAuthAccount`, the method `validate` first performs the same validation as the `validate` method of the class `CAccount` and, if this validation fails, it performs the same validation as the method `validate` of class `CAuth`. In TraitRecordJ, in order to realize this functionality, a new implementation of the method `validate` is provided in trait `TAuthAccount` (cf. Listing 3) that has two required methods: the method `validateAccount`, which comes from trait `TAccount`, and the method `validateAuth`, which comes from trait `TAuth`. When the class `CAuthAccount` is composed from the traits `TAccount`, `TAuth` and `TAuthAccount`, the `validate` methods of the traits `TAccount` and `TAuth` are renamed to match the names of the methods required by trait `TAuthAccount`. Instead of renaming, also the operation `aliasAs`, followed by an exclusion of the original method, can be used, as it is shown in Listing 3. Additionally, in the class `CAuthAccount`, the `update` methods of traits `TAccount` and `TAuth` are renamed for disambiguation. In this way, the functionalities provided by the traits `TAccount` and `TAuth` (originally introduced for implementing the classes `CAccount` and `CAuth`, respectively) can be reused and combined in a new manner. To summarize: the class `CAuthAccount` is composed from the records `RAccount` and `RAuth` and the traits `TAccount`, `TAuth` and `TAuthAccount`, reusing fields and methods developed for other classes with the necessary adjustments described above.

According to the class diagram in Fig. 1, besides authentication, the `CAccount` class can also be extended by billing facilities which is done in class `CFeeAccount`. For each transaction, a particular fee has to be paid. The fees are accumulated and deducted from the account once a month. In TraitRecordJ, in order to realize this functionality, a new trait `TFeeAccount` is introduced in Listing 4 which modifies the `update` method of the trait `TAccount` by first updating the account's balance and then adding the transaction fee to the accumulated fee. The fee for each transaction and the accumulated fees are stored in the fields provided by the record `RFeeAccount`. Additionally, the trait `TFeeAccount` provides the method `monthly` which deduces the accumulated fees once a month. The class `CFeeAccount` is composed from the records `RAccount` and `RFeeAccount` and the traits `TAccount` and `TFeeAccount` with renaming the `update` method of trait `TAccount` to `updateAccount` such that this method can be called form the modified `update` method of trait `TFeeAccount`. In this way, the original `update` method can be reused and wrapped with additional functionality.

Finally, the class `CMyAccount` combines the billing and the authentication functionalities. Hence, the class `CMyAccount` (cf. Listing 5) is composed from the records `RAccount`, `RAuth` and `RFeeAccount` providing together the fields for the

```
interface IAuthAccount extends IAuth, IAccount {}

trait TAuthAccount is {
  boolean validateAccount(int id); // Required Method
  boolean validateAuth(int id); // Required Method

  boolean validate(int id){
    boolean r = this.validateAccount(id);
    if (!r) {r = this.validateAuth(id);}
    return r;
  }
}

class CAuthAccount implements IAuthAccount by RAuth + RAccount and
  TAccount[validate aliasAs validateAccount,exclude validate,update renameTo updateAccount]
  + TAuth[validate renameTo validateAuth, update renameTo updateAuth] + TAuthAccount {
  CAuthAccount(){
    [...] // set default values
  }
}
```

Listing 3. Implementation of class `CAuthAccount`.

```
interface IFeeAccount extends IAccount {
  void monthly();
}

record RFeeAccount is {
  int fee;
  int accFee;
}

trait TFeeAccount is {
  void updateAccount(int y); // Required method
  int fee; // Required field
  int accFee; // Required field

  void update(int y) {
    this.updateAccount(y);
    if (y < 0){
      this.accFee = this.accFee + this.fee;
    }
  }

  void monthly() {
    this.updateAccount(−this.accFee);
    this.accFee = 0;
  }
}

class CFeeAccount implements IFeeAccount by RFeeAccount + RAccount and
  TFeeAccount + TAccount[update renameTo updateAccount]{
     CFeeAccount() {
    [...] // set default values
  }
}
```

Listing 4. Implementation of class `CFeeAccount`.

basic account, the authentication and the billing functionalities and from the traits `TAccount`, `TAuth`, `TFeeAccount` and `TMyAccount` where the trait `TMyAccount` only provides a wrapper for the `validate` method of the trait `TAuthAccount`.

In this example, it can be observed that flexible code reuse can be easily be achieved in TraitRecordJ using the trait and record composition operations when assembling classes from pre-existing records and traits. Furthermore, this example demonstrates that TraitRecordJ is powerful enough to achieve the same reuse capabilities as (multiple) inheritance while allowing explicit disambiguation and wrapping of methods by renaming, aliasing and method exclusion.

```
trait TMyAccount is {
   boolean validateAuthAccount(int id); // Required method
   boolean validate(int id) {this.validateAuthAccount(id);}
}

class CMyAccount implements IFeeAccount, IAuthAccount
by RFeeAccount + RAccount + RAuth
and TFeeAccount +
TAccount[validate aliasAs validateAccount,exclude validate,update renameTo updateAccount]
   + TAuth[validate renameTo validateAuth, update renameTo updateAuth]
   + TAuthAccount [validate renameTo validateAuthAccount]{
    CMyAccount(int owner, int f) {
        [...] // set default values
    }
}
```

Listing 5. Implementation of class `CMyAccount`.

4. Discussion

TraitRecordJ programs may look more verbose than standard class-based programs. However, the degree of reuse provided by records and traits is higher than the reuse potential of standard static class-based hierarchies. The distinction of each programming concept in a separate entity pays off in the long run, since each component is reusable in different contexts, in an unanticipated way. This does not happen so easily with standard class-based OO linguistic constructs. Class hierarchies need to be designed from the start with a specific reuse scenario in mind. In particular, for single inheritance, precise design decisions should be made from the very beginning. This design might be hard to change, if not impossible, forcing either to refactoring or to code duplication [48].

Our linguistic constructs are lower-level than standard OO mechanisms. However, some syntactic sugar can be added to reduce the amount of code to write in some situations. For instance: fields might also be declared directly in classes (thus encompassing the Java-like calculus by Bono et al. [21]), class names might be used as types, etc. Along the same lines, we can simulate class inheritance with our linguistic constructs; thus encoding Java dialects, like $Chai_2$ by Smith and Drossopoulou [61], and programming environments, like the one by Quitslund et al. [54], where traits coexist with class inheritance. This can be achieved easily by inverting the "flattening" concept. For instance, if we start from the Java-style class `Stack` given in Section 1, we can easily separate fields and methods into automatically generated records and traits, and generate the corresponding TraitRecordJ class declaration. Similarly, inheritance and method overriding can be simulated with inherited interfaces and trait method renaming, respectively. The `super` call can be simulated with a call to the renamed version of the method. However, this would decrease the level of reuse for such components.

This issue of having these additional constructs in the language is related to the debate of whether it is better to have a pure or hybrid programming language. In this respect, the purity of a language usually imposes more verbose solutions than a hybrid language. For instance, consider the amount of code for writing the `main` static public method of a public class in Java, and compare it to the simple form of the corresponding function in C++ which also provides functions besides classes and methods. The goal of reducing verbosity often led to additional language constructs which may break the pure linguistic features, e.g., the addition of imperative features to a purely functional language (see, e.g., Objective Caml [55]). The general debate between pure and hybrid languages is out of the scope of the present paper. Nonetheless, we argue that having linguistic constructs for reusable code development eases adding other high-level linguistic constructs which otherwise may often only be possible at the cost of code duplication and of the resulting complexity of code maintenance.

We might also introduce other syntactic sugar constructs to ease programming with our language; for instance, instead of writing the required methods in a trait, we could group the required methods in a specific interface and let the trait "implement" that interface meaning that the trait requires all the methods specified in the interface. However, this would not make traits types, since as we stressed throughout the paper, not having traits as types enhances their reuse potential.

Concerning the performance of the generated Java code we did not experience any difference with respect to a possible manually implemented version. Indeed traits and records (and the TraitRecordJ versions of classes and interfaces) are static constructs which do not have any overhead in the generated Java code.

5. Case study

In order to evaluate the usability of TraitRecordJ and its prototypical implementation based on Xtext, we reimplemented a part of a commercially used software system (that was previously implemented in Java) with TraitRecordJ. The goal of this case study is to analyze how far code duplications caused by the limitations of single inheritance can be avoided by trait and record composition. Furthermore, we aim at evaluating the potential for code reuse offered by records and traits in TraitRecordJ for existing application software.

The software analyzed in this case study is a part of the rendering engine for a web-based geographical information system [59]. The rendering engine allows printing geometrical objects to a screen or plotting them on paper. Fig. 2 shows

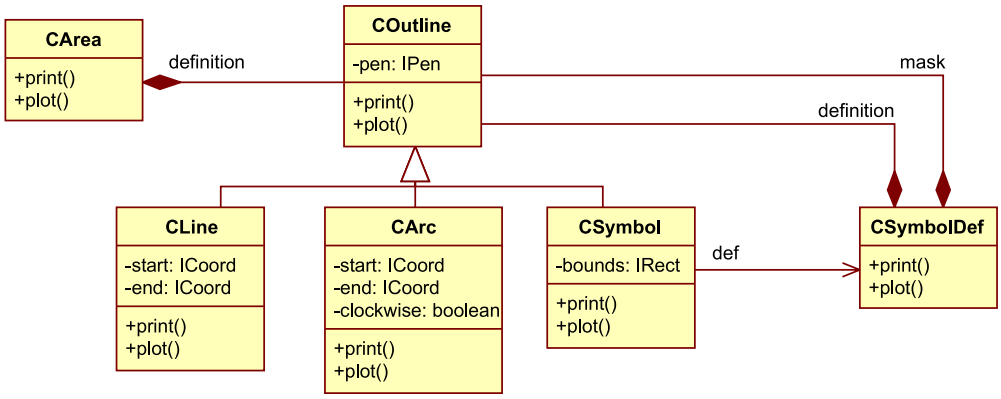

Fig. 2. Class diagram of the case example.

the class diagram of the analyzed code. The geometrical objects are organized in a class hierarchy in order to provide a taxonomy by inheritance. The `COutline` class is the superclass of all geometrical objects. `COutline` objects have a `pen` attribute describing how the objects should be painted and a `plot` and a `print` method. The `plot` method displays an object on a screen (using the API of the operating system) and the `print` method prints the object to paper (using a transformation to PostScript). Subclasses of the `COutline` class are the classes `CLine` and `CArc`, which represent more specialized geometrical objects. Both lines and arcs have start and end coordinates determining their position on a map. Additionally, a `CArc` object has an attribute `clockwise` denoting the direction in which the arc is bent. The `CArc` and the `CLine` class override the `plot` and `print` method of the `COutline` class.

A `CSymbolDef` object is a composite of `COutline` objects and represents a particular configuration of `Outline` objects on a map as a symbol. It contains two lists of `COutline` objects, `definition` containing the actual `COutline` objects and `mask` storing information about the order of the combined `COutline` objects for clipping purposes. Besides two methods to add objects to these lists, the `CSymbolDef` class contains a `plot` and a `print` method that call the `plot` and the `print` method, respectively, on each `COutline` object in the list. The class `CSymbol` is a subclass of the `COutline` class encapsulating a reference to a symbol definition described by a `CSymbolDef` object. The `plot` and `print` method implemented by the `CSymbol` class call the `plot` and `print` method, respectively, on the referenced `CSymbolDef` object with additionally performing some scaling operations.

The `CArea` class represents a region on a map. This region is defined by the set of `Outline` objects stored in the list-valued attribute `definition`. Thus, a `CArea` is not a `COutline` object itself. It implements a `plot` and `print` method that iterate over the `COutline` objects contained in the list referenced by the `definition` attribute.

In this case example, code duplication occurs in several places: Both classes `CSymbolDef` and `CArea` contain lists of `Outline` objects. However, `CSymbolDef` and `CArea` objects do not have a common superclass, since they are technically different entities in the rendering engine application. Hence, in the JAVA version of the code, the code for handling these lists is copied in the `CArea` and `CSymbolDef` classes. In TRAITRECORDJ, this code duplication can be avoided by factoring common attributes and methods into records and traits. On the top part, Listing 6 shows how a list of `COutline` objects and the method `add` for inserting objects into the list can be implemented as a TRAITRECORDJ record `RList` and trait `TAddList`. On the bottom part of the listing, it is shown how this record and this trait can be instantiated to implement the `definition` and `mask` lists and the respective `add` methods in the `CArea` and `CSymbolDef` objects.

The `plot` and `print` methods for the `CArea` and `CSymbolDef` classes are provided by the traits `TAreaPlot`, `TSymbolDefPlot`, `TAreaPrint` and `TSymbolDefPrint` depicted in Listing 6 in the class definitions. The `plot` and `print` methods of both classes iterate over the `definition` list and call the `plot` and `print` method, respectively, on each contained `COutline` object. Since the structure of these method implementations is very similar, it is possible to compose the above traits from a set of shared basic traits. These basic traits are depicted in Listing 7. The traits `TSymbolDefOutput` and `TAreaOutput` provide the generic output functionality for `CSymbolDef` and `CArea` objects, respectively. For `CSymbolDef` objects, some operations for the mask list are required before generating the output. For `CArea` objects, the target of the output has to be initialized and closed appropriately. Both provided output methods require a method `invokeIterator` that encapsulates the iteration over a required list of `COutline` objects. This iterator method is provided by the trait `TIterateOutput`. In order to adapt the `invokeIterator` method to calling the `print` or `plot` method on the `Outline` objects, the trait `TIterateOutput` requires a method `invokeOutputMethod` that encapsulates the respective calls. The `invokeOutputMethod` can be provided by the trait `TInvokePlot` for plotting or by the trait `TInvokePrint` for printing. The level of indirection is necessary to rename a method call on an object different from `this`.

In order to build the `TAreaPlot`, `TSymbolDefPlot`, `TAreaPrint` and `TSymbolDefPrint` traits, the basic traits defined in Listing 7 have to be adapted by renaming and composed appropriately. This composition is depicted in Listing 8. The `TSymbolDefPlot` and `TSymbolDefPrint` traits use the trait `TSymbolDefOutput` for outputting `CSymbolDef` objects. The `TAreaPlot` and `TAreaPrint` traits reuse the `TAreaOutput` trait. The traits providing plotting functionality

```
record RList is{
   IList list;
}

trait TAddList is {
   IList list;
   void add(IObject o){
      this.list.add(this.list.size(), o);
   }
}

class CArea implements IArea by RList[list renameTo definition]
      and TAddList[list renameFieldTo definition]
      + TAreaPlot + TAreaPrint {
   CArea() {
      this.definition = null;
   }
}


class CSymbolDef implements ISymbolDef by RList[list renameTo definition] + RList[list renameTo mask]
      and TAddList[list renameFieldTo definition] + TAddList[list renameFieldTo mask, add renameTo addToMask]
      + TSymbolDefPlot + TSymbolDefPrint {
   CSymbolDef(){
      this.definition = null;
      this.mask = null;
   }
}
```

Listing 6. Reuse of record `RList` and trait `TAddList` for storing `Outline` lists.

contain the `TInvokePlot` trait and rename the respective output method to `plot`. The traits involved with printing comprise the `TInvokePrint` trait and rename the output method to `print`. Since all outputs involve iteration over a list of `COutline` objects, the `TIterateOutput` trait is contained in every composed trait. In all traits requiring a reference to the list of `COutline` objects, the `list` field is renamed to the `definition` field.

In the original implementation, the code of the two `print` and the two `plot` methods was written separately and contained code duplications to a large extent since the `CArea` and `CSymbolDef` are unrelated in the class hierarchy. In contrast, Listing 8 shows how similar functionality can be factored into shared traits and how it can be adapted and composed to provide the functionality of a particular class without suffering the limitation of a class hierarchy with respect to code reuse.

Fig. 3 provides a visualization of how classes `CSymbolDef` and `CArea` can be composed from four shared composite traits and six shared basic traits (for simplicity interfaces are records are not shown in the diagram). The basic traits are listed in the left column of the figure, the composite parts are listed in the middle column, and the classes are in the right column.

In our case study, we observed that there is a high potential to reuse common code parts in various places of the considered implementation. The main reason for code duplication we found in this case example is that common functionality in two classes that are non-related in the class hierarchy cannot be shared by the means of class-based inheritance. Using TraitRecordJ, this common code can be factored out into records and traits that can be customized and composed to define more complex traits and classes. Notably, we found natural to exploit the field renaming and the method renaming operations, which are not supported by Fortress and Scala.

6. Implementing TraitRecordJ in Xtext

In this section, we describe the implementation of TraitRecordJ using Xtext [5]. Although Eclipse itself provides a framework for implementing an IDE for programming languages, this procedure is still quite laborious and requires a lot of manual programming. Xtext eases this task by providing a high-level framework that generates most of the typical and recurrent artifacts necessary for a fully-fledged IDE on top of Eclipse.

The first task in Xtext is to write the grammar of the language using an EBNF-like syntax. Starting from this grammar, Xtext generates an ANTLR parser [51]. The generation of the abstract syntax tree is handled by Xtext as well. In particular, during parsing, the AST is generated in the shape of an EMF model (Eclipse Modeling Framework [63]). Thus, the manipulation of the AST can use all mechanisms provided by EMF itself. There is a direct correspondence between the names used in the rules of the grammar and the generated EMF model Java classes. For instance, consider the Xtext grammar snippet on top of Listing 9. The Xtext framework generates the EMF model Java interfaces (and the corresponding implementation class) as shown in the rest of Listing 9. Note also the inferred inheritance relations, since both `MethodInvocation` and `FieldAccess` are `Messages`.

```
trait TInvokePlot is {
   void invokeOutputMethod(IPlotable elem1, IOutput target){
      elem1.plot(target);
   }
}

trait TInvokePrint is {
   void invokeOutputMethod(IPlotable elem1, IOutput target){
      elem1.print(target);
   }
}

trait TIterateOutput is {
   IList list;
   void invokeOutputMethod(IOutline elem1, IOutput target);
   void invokeIterator(IOutput target) {
      int i =0;
      while(i < this.list.size()) {
         IObject elem = this.list.get(i);
         IOutline elem1 = (IOutline) elem;
         this.invokeOutputMethod(elem1,target);
         inc(i);
      }
   }
}

trait TSymbolDefOutput is {
   IList list;
   IList mask;
   void invokeIterator(IList list, IOutput target);
   void symbolDefOutput(IOutput target) {
      if (this.mask != null){
         // perform operations for mask
      }
      this.invokeIterator(this.list,target);
   }
}


trait TAreaOutput is {
   IList list;
   void invokeIterator(IList list, IOutput target);
   void areaOutput(IOutput target){
      if (this.list.size() > 0){
         target.newPath();
         this.invokeIterator(this.list,target);
         target.closePath();
      }
   }
 }
```

Listing 7. Basic Traits for plotting and printing of `CArea` and `CSymbolDef` class.

```
trait TSymbolDefPlot is TSymbolDefOutput[list renameFieldTo definition, symbolDefOutput renameTo plot]
      + TInvokePlot + TIterateOutput[list renameFieldTo definition]

trait TSymbolDefPrint is TSymbolDefOutput[list renameFieldTo definition, symbolDefOutput renameTo print]
      + TInvokePrint + TIterateOutput[list renameFieldTo definition]

trait TAreaPlot is TAreaOutput[list renameFieldTo definition, areaOutput renameTo plot]
            + TIterateOutput[list renameFieldTo definition]
            + TInvokePlot

trait TAreaPrint is TAreaOutput[list renameFieldTo definition, areaOutput renameTo print]
            + TIterateOutput[list renameFieldTo definition]
            + TInvokePrint
```

Listing 8. Composed Traits for plotting and printing in `CArea` and `CSymbolDef` classes.

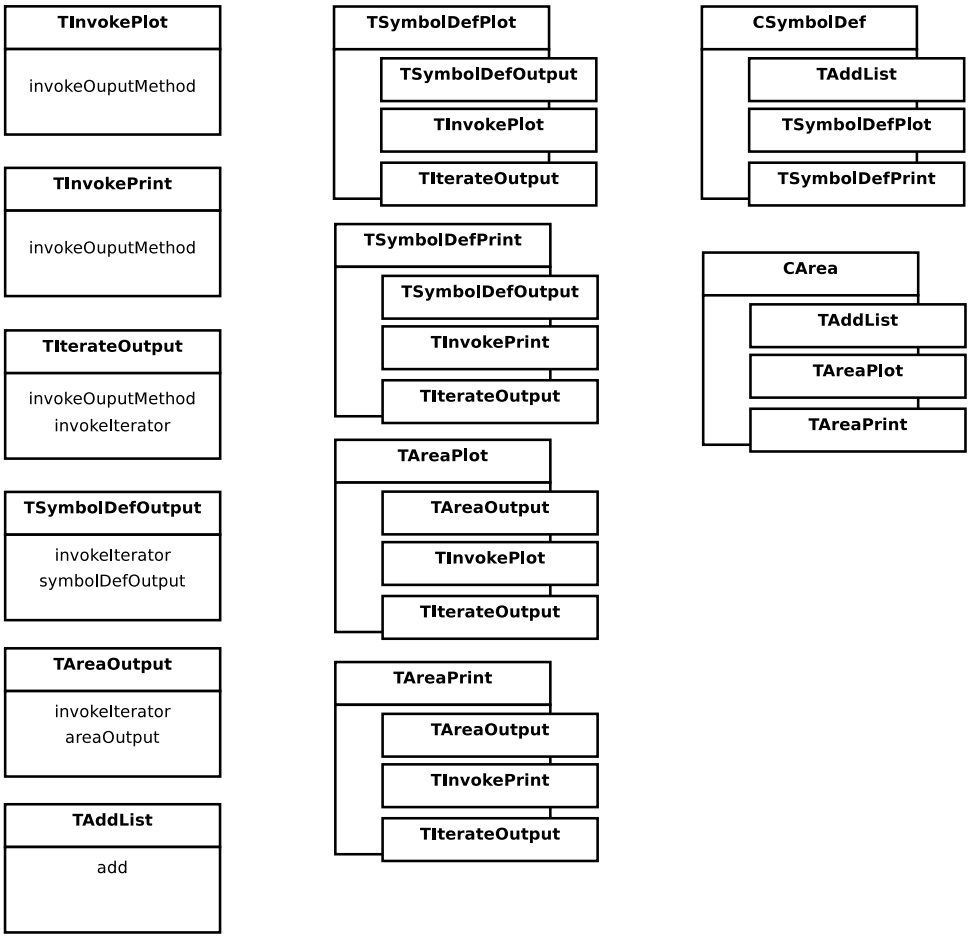

Fig. 3. Classes `CSymbolDef`, `CArea` and used traits.

```
Message : MethodInvocation | FieldAccess;

MethodInvocation :
   method=[Method]
       '(' (argumentList+=Expression (',' argumentList+=Expression)*)? ')';

FieldAccess : field=[Field];

   public interface MethodInvocation extends Message {
      Method getMethod();
      void setMethod(Method value);
      EList<Expression> getArgumentList();
   }

   public interface FieldAccess extends Message {
      Field getField();
      void setField(Field value);
   }
```

Listing 9. From the Xtext grammar to the EMF model generated code.

Moreover, Xtext generates many other classes with IDE functionalities; in particular, it generates an editor with syntax highlighting, background parsing with error markers, outline view and code completion. Furthermore, Xtext provides the infrastructure for code generation. Most of the code generated by Xtext can already be used off the shelf, but other parts have to be adapted by customizing some classes used in the framework. The usage of the customized classes is dealt with by relying on Google-Guice [2], a *dependency injection* framework, so that the programmer does not have to maintain customized abstract factories [37].

After the parsing stage, in a compiler, once we have the AST of the program, we need to visit the AST in order to perform, for instance, type checking, possible optimizations, and code generation. This usually requires specific visitors to be written for the abstract syntax tree [37]. The main problem of the visitor patter, besides requiring lot of manual programming and cluttering the classes of the model with specific methods, is that it works with methods with only one parameter; this might not be enough in some cases, e.g., to check the subtyping relation we need two parameters, and have a dynamic overloading like mechanism to select the right method at run-time according to the run-time type of the arguments. Xtext provides a more powerful mechanism for inspecting a model (i.e., the AST) at runtime, through the class `PolymorphicDispatcher`, which performs method dispatching according to the run-time type of arguments (*dynamic overloading*).

6.1. Scoping

Binding the symbols (e.g., the binding of a field reference to its declaration) is part of the validation of a program, since if a symbol cannot be bound then it means that there is an error in the program (e.g., a reference to a mispelled variable name). Actually, it can be considered one of the checking parts which should come earlier in the validation of a program; in fact, once all the symbols are bound to the corresponding declarations, implementing type checking is easier, since given a symbol we can immediately "jump" to its declaration. If a symbol cannot be bound, the type checking makes no sense at all.

When defining the grammar of the language in Xtext, we can already specify that a token is actually a reference to a specific declared element. Going back to the grammar snippet of Listing 9, we note that the name of the method in a method invocation `MethodInvocation` is enclosed in square brackets (`[Method]`). In the generated EMF Java code, the name of the method in a method invocation expression will not be simply a string, but it will be a cross reference to an instance of `Method` (see `getMethod` in Listing 9).

EMF uses "proxies" to represent references. It can delay the resolution (binding) of references when they are accessed. Note that, since the EMF model representing the program contains cross-references, the AST is not actually a tree; however, we will continue to use the term AST to refer to such a model. Xtext already provides an implementation for binding references, which basically binds a reference to a symbol *n* to the first element definition with name *n* occurring in the model. This usually has to be adapted in order to take the visibility of names in a program into account. For instance, a field is visible only in the methods of a class, such that different hierarchies can safely have fields with the same name. Xtext supports the customization of binding in an elegant way with the abstract concept of "scope". The actual binding is still performed by Xtext, but it can be driven by providing the scope of a reference, i.e., all declarations that are available in the current context of a reference.

The programmer can provide a customized `AbstractDeclarativeScopeProvider`. Xtext will search for methods to invoke, using reflection and polymorphic dispatch, according to a convention on method name signature:

```
IScope scope_<ContextRuleName>_<ReferenceAttributeName>(<ContextType> ctx, EReference ref)
```

This is used when evaluating the scope for a specific cross-reference stored in the attribute *ReferenceAttributeName* of an element of type *ContextRuleName* (separated by an underscore) in the context of a rule *ContextType*. if such a method does not exist, then Xtext will use the default binding semantics (see above).

For instance, if we consider the grammar rules for message selection on an expression (called `receiver` of a message)[2]:

```
DottedExpression: receiver=Expression '.' message=Message
Message : MethodInvocation | FieldAccess;
MethodInvocation : method=[Method]
       '(' (argumentList+=Expression (',' argumentList+=Expression)*)? ')';
FieldAccess : field=[Field];
```

we can drive the resolution of the method name in a `MethodInvocation` statement in any `DottedExpression` where such a statement can occur by defining the method in Listing 10. The code should be understandable without the knowledge of Xtext: we get the type of the `receiver` of the message, using additional classes we implemented to perform type checking (not shown here); if the type can be inferred then we return as the scope all the methods which are invokable on such a type. Thus, in this case, *ReferenceAttributeName* is `method`, *ContextRuleName* is `MethodInvocation` and *ContextType* is `DottedExpression`. The scope implementation for `FieldAccess` is similar, but it returns the list of fields which are available on the type of the receiver. It is up to the programmer to choose the right context for computing the scope of a reference; in this example, to compute the list of all invokable methods, we need the `receiver`, thus the context with this information is `DottedExpression`.

Using the returned scope, Xtext will then take care of resolving a cross reference. If Xtext succeeds in resolving a cross reference, it also takes care of implementing the functionalities Eclipse users are used to, e.g., by Ctrl+clicking on a reference (or using F3) we can jump to the declaration for that symbol. Furthermore, the scope provider will be used by Xtext to also implement code completion. Thus, a programmer achieves two goals by implementing the abstract concept of scope. Note that the code above can also return an empty scope, e.g., if the receiver expression in a method call cannot be typed. In that

[2] This grammar rule is slightly more involved in the actual implementation of TraitRecordJ; this is due to the fact that in order to deal with left recursion, which LL-parsing tools cannot handle directly, we need to "left-factor" the grammar [6].

```
public IScope scope_MethodInvocation_method(DottedExpression context, EReference ref) {
    ExpressionType expressionType = ExpressionType.createInstance(context.getReceiver());
    Collection<Method> methodList = new LinkedList<Method>();
    if (expressionType != null)
        methodList = expressionType.getInvokableMethods();
    return Scopes.scopeFor(methodList);
}

public IScope scope_FieldAccess_field(DottedExpression context, EReference ref) {
    ExpressionType expressionType = ExpressionType.createInstance(context.getReceiver());
    Collection<Field> fieldList = new LinkedList<Field>();
    if(expressionType != null)
        fieldList = expressionType.getInvokableFields();
    return Scopes.scopeFor(fieldList);
}
```

Listing 10. Scoping for method invocation and field access.

```
@Check
public void thisCheck(This thisRule) {
  if((lookup.getOwner(thisRule) instanceof Program)) {
    error("'this' is not allowed in program context", thisRule));
  }
}
```

Listing 11. Validation of an occurrence of `this`.

case, Xtext generates an error due to an unresolvable method name during validation, and an empty code completion list in case the programmer requests content assistance when writing the method name of a method invocation expression. This mechanism is handled by the framework itself, so that the programmer is completely relieved from these issues, once the correct scope provider is implemented.

6.2. Validation

The concept of *validation* comes from the EMF framework: it corresponds to checking that a model is consistent. When implementing a language with Xtext the EMF model is the AST of the current program. In the case of a statically typed programming language the validation corresponds to checking that the current program is well typed according to the type system of the language.

As we said at the beginning of Section 6, Xtext provides some useful and powerful mechanisms that do not require the programmer to implement the typical visitor structure [37] in order to validate the AST. In Section 6.1 we already saw that Xtext uses reflection and polymorphic dispatch to deal with the different kinds of nodes in the AST. In the context of validation, Xtext leverages this mechanism by requiring methods with a `@Check` annotation, in a customized validator (extending the base class `AbstractDeclarativeValidator`), that will be called automatically for validating the model according to the type of the AST node that has to be checked. The validation takes place in the background, together with parsing, while the user is writing a TraitRecordJ program, so that an immediate feedback is available, as it usually happens in IDEs.

For instance, the method in our validator shown in Listing 11 checks that the `this` variable is not used in the `program` context (i.e., that `this` can be used only in method bodies). Note that the important things in this method definition are the `@Check` annotation and the parameter: the internal validator of Xtext will invoke this method when it needs to validate an AST node representing an occurrence of `this`, which is declared in the grammar as the `This` non-terminal symbol (remember that given a grammar symbol, Xtext will generate a corresponding class for the AST with the same name). We do not have to handle the actual visiting of the EMF model corresponding to the program: we just need to tell Xtext what to do in order to check that a specific kind of node in the AST is correct.

If an error is found during the validation, then calling the method `error` of the validator base class will make Xtext generate the appropriate error marker (and since we specify the element in the AST which did not pass validation, Xtext is able to put the marker in the right place without any additional information by the programmer).

In Listing 12 we show another example of validator method: this checks that the constructor name is the same as the name of the class where it is defined. Note that in this case, since the `Constructor` object in the model corresponds to a long string in the text of the program, we also specify, when we invoke the method `error`, the attribute (in the example, it is the constructor `name`, using the constant generated by EMF) to be highlighted in the error marker and in the problem view of the Eclipse editor.

```
@Check
public void checkConstructorName(Constructor constructor) {
  Class classRef = (Class)lookup.getOwner(constructor);
  if(!classRef.getName().equals(constructor.getName()))
      error("Constructor must have the same name of the class",
        constructor,
        CONSTRUCTOR_NAME));
}
```

Listing 12. Validation of a constructor.

6.3. Other functionalities

Xtext provides a (mostly) automatic support for file import/inclusion in the developed language. We use this functionality so that TraitRecordJ programs can be split into separate files. This way, in the EMF model (AST) corresponding to the current file, there are cross references to the EMF models corresponding to the imported files. These cross references are handled automatically by Xtext and when using F3 (or Ctrl+click) to jump to the definition of a symbol, if that symbol is in an imported file, we will be automatically redirected to the file (i.e., another editor window will be automatically opened for the imported file). Furthermore, the corresponding dependencies among source files are handled by Xtext itself. Thus, the modification of an included file f automatically triggers the re-validation of all the files including f.

The code generation phase can be dealt with in Xtext by relying on XPand [4], a code generation framework based on "templates", specialized for code generation based on EMF models (however, other generation frameworks can be used). In our implementation of TraitRecordJ, code generation produces standard Java code, which does not need any additional libraries to be compiled and executed. Our code generation phase basically implements the flattening procedure sketched in Section 2.2. However, by providing different templates, we could also generate into different target languages (this is subject of future work).

Xtext generates three plugin projects: one for the language parser and corresponding validators, one for the code generator, and one for the user interface IDE parts. The first two plugins do not depend on the third. Thus, it is straightforward to build a stand-alone compiler for TraitRecordJ to be executed outside Eclipse on the command line, which we also provide.

6.4. Xtext evaluation

Our experience with Xtext was definitely positive. Xtext seems to be the right tool to experiment with language design and to develop implementations of languages. Furthermore, experimenting with new constructs in the language being developed can be handled straightforwardly. It requires to modify the grammar, regenerate Xtext artifacts and to deal with the cases for the new constructs.

Even though Xtext hides many internal details of IDE development with Eclipse, still it lets the programmer customize every aspect of the developed language implementation by specialized code (which is flexibly "injected" in Xtext using Google Guice). Even EMF mechanisms are still open to adaptation. For instance, we developed a customized EMF resource factory for synthesizing the interfaces and classes of the internal library described in Section 2, such as `IList` and `CArrayList`, which will then be transparently available in every program (represented as an EMF model), without having to treat them differently in program validation.

As we described in Section 6, by using our own validator, Xtext can create the error markers in the editor (and also in the "Problem View") related to the EMF element which caused the error. We can provide our own "quickfixes" by simply deriving from an Xtext default class and providing a method which refers to a specific error detected by our validator. Since we deal with EMF model objects, which are connected by Xtext to the program text in the editor, we can simply manipulate the EMF model and the program in the text editor will reflect this change; in particular, we do not need to manipulate the eclipse text editor document at all.

Xtext also provides some useful functionalities to write Junit tests for many language development components; it generates a stand-alone class for the developed language with all the functionalities to correctly initialize all the EMF mechanisms so that the language can be tested as a stand-alone application (i.e., outside from Eclipse). This way, we can easily test our language with no need of: manually running an Eclipse application, writing a code snippet in our language, and checking that the error mark shows up. This speeds up the development of the language.

Summarizing, Xtext lets the language developer concentrate on the aspects that are typical of his own language, while relying on the framework for all the other recurrent jobs.

7. Related work

Traits are well suited for designing libraries and enable clean design and reuse which has been shown using Smalltalk/Squeak (see, e.g., [19,25]). Recently, Bergel et al. [11] pointed out limitations of the trait model caused by the fact that methods provided by a trait can only access state by accessor methods (which become required methods of the

trait). To avoid this, traits are made *stateful* (in a SMALLTALK/SQUEAK-like setting) by adding private fields that can be accessed from the clients possibly under a new name or merged with other variables. In TRAITRECORDJ traits are stateless. By their required fields, however, it is possible to directly access state within the methods provided by a trait. Moreover, the names of required fields (in traits) and provided fields (in records) are unimportant because of the field rename operation. Since field renaming works synergically with method renaming, exclusion and aliasing, TRAITRECORDJ has more reuse potential. Concerning field requirements, they are not present in most formulations of traits in the SMALLTALK/SQUEAK-like and JAVA-like settings. They were introduced in the formulation of traits in a structurally typed setting by Fisher and Reppy [34].

TRAITRECORDJ requires that the summed traits must be disjoint. The disjoint requirement for composed traits was proposed by Snyder [62] for multiple class-based inheritance (see also Bracha's JIGSAW framework [22]). According to other proposals, two methods with the same name do not conflict if they are syntactically equal (Ducasse et al. [33,49]) or if they originate from the same subtrait (Liquori and Spiwack [46]). When a recursive method is aliased in our language, its recursive invocation refers to the original method (as proposed by Schärli et al. [60,33]). The variant of aliasing proposed by Liquori and Spiwack [46] (where, when a recursive method is aliased, its recursive invocation refers to the new method) can be straightforwardly encoded by exclusion, renaming and symmetric sum. Instead, exclusion, renaming, symmetric sum and the variant of aliasing are not able to encode aliasing. Concerning method renaming and required field renaming, they are not present in most formulations of traits in the SMALLTALK/SQUEAK-like and JAVA-like settings. Method renaming has been introduced in the formulation of traits in a structurally typed setting by Reppy and Turon [57]. Renaming operations were already present in the JIGSAW framework [22] in connection with module composition and in the EIFFEL language [47] in connection with multiple class-based inheritance.

Reppy and Turon [58] proposed a variant of traits that can be parametrized by member names (field and methods), types and values. Thus, the programmer can write *trait functions* that can be seen as code templates to be instantiated with different parameters. This enhances the code reuse provided by traits already. It could be interesting to adapt this approach to our context and to extend the parametrization functionalities also to interfaces, records and classes. This will be the subject of future work. However, an important difference between our proposal and the one by Reppy and Turon [58] is that, in the latter, traits play also the competing role of type, which is avoided in TRAITRECORDJ. Another feature of TRAITRECORDJ is that structural types are used only "internally" on `this`, i.e., the programmer works with nominal types (interfaces) alone. We believe this is an important feature from a practical point of view, as it reduces the distance between the classical JAVA-like languages and our linguistic constructs, from the perspective of the programmer.

Quitslund et al. [54] proposed an Eclipse plugin that supports JAVA programmers with using trait-like mechanisms for JAVA classes. JAVA is not extended: traits are modeled as (possibly) abstract classes and trait method requirements as abstract methods. The plugin also aims at helping the programmer to refactor JAVA hierarchies with traits: the programmer can select a set of methods to be extracted into stateless classes that play the role of traits. When a class uses a trait, the methods provided by the trait are copied into that class; the programmer then has to use this plugin to keep track of the actual source of a method, in order not to accidentally change the copy of a method and to be aware of what to change: either the copy of the class or the original method in the trait. Thus the programmer has to rely on this plugin completely, while in our case the TRAITRECORDJ compiler can be used also outside from Eclipse.

The work by Quitslund et al. [54] starts from an earlier work by Quitslund [53] where traits are implemented directly as an extension of a subset of the JAVA language, and a compiler translates such programs into standard JAVA programs. The authors, however, state that such an approach requires a strong effort to be conservative with respect to JAVA since, in order to generate well typed JAVA code, many type checking functionalities for JAVA related code must be implemented in their compiler. Such extension, moreover, lacks a formalization, thus no property of type safety for such language extension is available.

In our approach, since we deal with a JAVA dialect, we can concentrate on the linguistic parts that are characteristic of our own language, without re-implementing JAVA checks. However, the code generated by our compiler is standard JAVA code, and our formalization [14] guarantees that such code will be type correct with respect to the JAVA compiler (starting from a well-typed TRAITRECORDJ program).

There are other tools for implementing both domain specific and general purpose languages and their text editors and IDE functionalities (we also refer to Pfeiffer and Pichler [52] for a comparison). Tools like IMP (The IDE Meta-Tooling Platform) [28] and DLTK (Dynamic Languages Toolkit) [1] only deal with IDE functionalities and leave the parsing mechanism completely to the programmer, while XTEXT starts the development cycle right from the grammar itself. TCS (Textual Concrete Syntax) [3] is similar to XTEXT since it enables the specification of textual concrete syntaxes for DSLs by attaching syntactic information to metamodels; however, with XTEXT one describes the abstract and concrete syntax at once, and it is completely open to customization of every part of the generated IDE (besides, TCS seems to be no longer under active development). Another framework, closer to XTEXT is EMFText [38]. EMFText basically provides the same functionalities. But, instead of deriving a meta-model from the grammar, it does the opposite, i.e., the language to be implemented must be defined in an abstract way using an EMF meta model. (A meta model is a model describing a model, e.g., an UML class diagram describing the classes of a model.) Note that XTEXT can also connect the grammar rules to an existing EMF meta model, instead of generating an EMF meta model starting from the grammar. Furthermore, XTEXT also has a wizard to generate an XTEXT grammar starting from an EMF meta model. XTEXT seems to be better documented than EMFText (indeed, both projects are still young and always under intense development), and more flexible, especially since it relies on Google Guice. On the other hand, EMFText offers a "language zoo" with many examples that can be used to start the development

of another language. In this respect, the examples of languages implemented using XTEXT, that we found on the web, are simpler DSLs, and not programming languages like TRAITRECORDJ. Thus, this paper can also be seen as a report of effective usage of XTEXT for implementing more complex programming languages.

EriLex [67] is a software tool for generating support code for embedded domain specific languages and it supports specifying syntax, type rules, and dynamic semantics of such languages. EriLex does not generate any artifact for IDE functionalities, and it concentrates on other aspects of language development such as type systems and operational semantics. MPS (Meta Programming System) [36] is another tool for developing a DSL, and it also provides IDE functionalities, but it does not target Eclipse.

Neverlang [27] is based on the fact that programming language features can be easily plugged and unplugged. A complete compiler/interpreter can be built in Neverlang as the result of a compositional process involving several building blocks. With respect to composition functionalities, XTEXT allows the programmer to mix grammars (so called "grammar mixins") and also to reuse recurrent syntax artifacts (like the standard terminal definition grammar, typically included at the beginning of an XTEXT grammar definition). However, these compositional functionalities are not yet as powerful as the ones provided by Neverlang.

Finally, we just mention here other tools and frameworks for implementation of DSLs, such as, e.g., JTS [9], MetaBorg [24] and MontiCore [41] which are based on language specification preprocessors and XMF [29] and Language Boxes [56] which target host language extensions and internal DSLs.

8. Conclusions and future work

We presented the programming language TRAITRECORDJ and its implementation. The language isolates each linguistic entity in a separate linguistic construct, namely, types into interfaces, behaviors into traits, states into records, and instance generators into classes. TRAITRECORDJ programs may look more verbose than standard class-based programs. However, as illustrated by the case study in Section 5, the degree of reuse provided by records and traits is higher than the reuse potential of standard static class-based hierarchies.

TRAITRECORDJ is based on the calculus presented in [13,14]. In that paper, we considered mechanisms for code reuse for implementing Software Product Lines (a set of software systems with well-defined commonalities and variabilities [30]). We explored a novel approach to the development of SPL, which provides flexible code reuse with static guarantees. In order to be of effective use for SPL, the type checking has to facilitate the analysis of newly added parts without re-checking already existing products. The TRAITRECORDJ type system (see Section 2.3) satisfies this requirement since it supports type checking records and traits in isolation from classes that use them.

A special form of *reuse* is at the base of the contemporary *agile* software development methodologies, which are based on an iterative approach, where each iteration may include all of the phases necessary to release a small increment of a new functionality: planning, requirements analysis, design, coding, testing, and documentation. Another example is the use of *Extreme Programming* [10], where team members work on activities simultaneously. While an iteration may not add enough functionality to guarantee the release of a final product, an agile software project intends to be capable of releasing new software at the end of every iteration. However, this means that the next iteration will *reuse* the software produced in the previous ones. We believe that an interesting future research direction is to investigate whether the programming language features proposed in this paper may help in writing software following an agile methodology.

Bettini et al. [12] present a tool for identifying the methods in a JAVA class hierarchy that could be good candidates to be refactored in traits (by adapting the SMALLTALK analysis tool by Lienhard et al. [44] to a JAVA setting). It will be interesting to investigate the application of this approach also for detecting possible candidates for records and traits in the context of porting existing JAVA code to TRAITRECORDJ code.

Acknowledgements

We are grateful to the developers of XTEXT, in particular, Sven Efftinge and Sebastian Zarnekow, for their prompt help and support during the development of TRAITRECORDJ. We thank Viviana Bono and Stéphane Ducasse for many interesting discussions on the design of TRAITRECORDJ and Alexandre Bergel for insightful comments on the trait operations supported by TRAITRECORDJ. We also thank Thomas Sauer for helping with the case example. Finally, we thank the anonymous SCP referees for many suggestions for improving the presentation.

References

[1] DLTK. http://www.eclipse.org/dltk.
[2] Google guice. http://code.google.com/p/google-guice.
[3] TCS (Textual Concrete Syntax). http://www.eclipse.org/gmt/tcs.
[4] Xpand. http://www.eclipse.org/modeling/m2t/?project=xpand.
[5] Xtext — a programming language framework. http://www.eclipse.org/Xtext.
[6] A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman (Eds.), Compilers: Principles, Techniques, and Tools, 2nd edition, Addison Wesley, 2007.
[7] E. Allen, D. Chase, J. Hallett, V. Luchangco, G.-W. Maessen, S. Ryu, G. Steele, S. Tobin-Hochstad, The Fortress Language Specification, V. 1.0, 2008.
[8] D. Ancona, G. Lagorio, E. Zucca, Jam—designing a Java extension with mixins, ACM TOPLAS 25 (5) (2003) 641–712.

[9] D. Batory, B. Lofaso, Y. Smaragdakis, JTS: tools for implementing domain-specific languages, in: ICSR, IEEE, 1998, pp. 143–153.
[10] K. Beck, Extreme Programming Explained: Embrace Change, Addison-Wesley, 1999.
[11] A. Bergel, S. Ducasse, O. Nierstrasz, R. Wuyts, Stateful traits and their formalization, Computer Languages, Systems & Structures 34 (2–3) (2008) 83–108.
[12] L. Bettini, V. Bono, M. Naddeo, A trait based re-engineering technique for Java hierarchies, in: PPPJ, ACM, 2008, pp. 149–158.
[13] L. Bettini, F. Damiani, I. Schaefer, Implementing Software Product Lines using Traits, in: OOPS, Track of SAC, ACM, 2010, pp. 2096–2102.
[14] L. Bettini, F. Damiani, I. Schaefer, Implementing type-safe software product lines using records and traits, Technical Report RT 135/2011, Dipartimento di Informatica, Università di Torino, 2011. Available at: http://www.di.unito.it/~damiani/papers/tr-135-2011.pdf.
[15] L. Bettini, F. Damiani, I. Schaefer, F. Strocco, A prototypical Java-like language with records and traits, in: PPPJ, ACM, 2010, pp. 2096–2102.
[16] G. Bierman, M. Parkinson, A. Pitts, MJ: An imperative core calculus for Java and Java with effects, Technical Report 563, University of Cambridge, Computer Laboratory, April 2003.
[17] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, M. Denker, Squeak by Example, Square Bracket Associates, 2007.
[18] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, M. Denker, Pharo by Example, Square Bracket Associates, 2009.
[19] A.P. Black, N. Schärli, S. Ducasse, Applying traits to the Smalltalk collection classes, in: OOPSLA, ACM, 2003, pp. 47–64.
[20] V. Bono, F. Damiani, E. Giachino, Separating type, behavior, and state to achieve very fine-grained reuse, in: Electronic Proceedings of FTfJP, 2007.
[21] V. Bono, F. Damiani, E. Giachino, On traits and types in a Java-like setting, in: TCS (Track B), in: IFIP, vol. 273, Springer, 2008, pp. 367–382.
[22] G. Bracha, The programming language JIGSAW: mixins, modularity and multiple inheritance, Ph.D. Thesis, Department of Comp. Sci., Univ. of Utah, 1992.
[23] G. Bracha, W. Cook, Mixin-based inheritance, in: OOPSLA, in: SIGPLAN Notices, vol. 25(10), ACM, 1990, pp. 303–311.
[24] M. Bravenboer, R. de Groot, E. Visser, MetaBorg in action: examples of domain-specific language embedding and assimilation using stratego/XT, in: GTTSE, in: LNCS, vol. 4143, Springer, 2006, pp. 297–311.
[25] D. Cassou, S. Ducasse, R. Wuyts, Redesigning with traits: the Nile stream trait-based library, in: ICDL, ACM, 2007, pp. 50–75.
[26] D. Cassou, S. Ducasse, R. Wuyts, Traits at work: the design of a new trait-based stream library, Computer Languages, Systems and Structures 35 (1) (2009) 2–20.
[27] W. Cazzola, D. Poletti, DSL evolution through composition, in: ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE), ACM, 2010.
[28] P. Charles, R. Fuhrer, S. Sutton Jr, E. Duesterwald, J. Vinju, Accelerating the creation of customized, language-Specific IDEs in Eclipse, in: OOPSLA, ACM, 2009, pp. 191–206.
[29] T. Clark, P. Sammut, J. Willans, Superlanguages, developing languages and applications with XMF, in: Ceteva, 1st edition, 2008.
[30] P. Clements, L. Northrop, Software Product Lines: Practices and Patterns, Addison Wesley Longman, 2001.
[31] W. Cook, W. Hill, P. Canning, Inheritance is not subtyping, in: POPL, ACM, 1990, pp. 125–135.
[32] J. Dovland, E.B. Johnsen, O. Owe, M. Steffen, Incremental reasoning with lazy behavioral subtyping for multiple inheritance, Science of Computer Programming 76 (10) (2011).
[33] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, A. Black, Traits: a mechanism for fine-grained reuse, ACM TOPLAS 28 (2) (2006) 331–388.
[34] K. Fisher, J. Reppy, A typed calculus of traits, in: FOOL, 2004.
[35] M. Flatt, S. Krishnamurthi, M. Felleisen, Classes and mixins, in: POPL, ACM, 1998, pp. 171–183.
[36] M. Fowler, A language workbench in action — MPS, 2008. http://martinfowler.com/articles/mpsAgree.html.
[37] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
[38] F. Heidenreich, J. Johannes, S. Karol, M. Seifert, C. Wende, Derivation and refinement of textual syntax for models, in: ECMDA-FA, in: LNCS, vol. 5562, Springer, 2009, pp. 114–129.
[39] J. Hendler, Enhancement for multiple-inheritance, in: SIGPLAN Workshop on Object-Oriented Programming, ACM, 1986, pp. 98–106.
[40] A. Igarashi, B. Pierce, P. Wadler, Featherweight Java: a minimal core calculus for Java and GJ, ACM TOPLAS 23 (3) (2001) 396–450.
[41] H. Krahn, B. Rumpe, S. Völkel, Monticore: a framework for compositional development of domain specific languages, STTT 12 (5) (2010) 353–372.
[42] G. Lagorio, M. Servetto, E. Zucca, Featherweight Jigsaw — a minimal core calculus for modular composition of classes, in: ECOOP, in: LNCS, 5653, Springer, 2009, pp. 244–268.
[43] G. Lagorio, M. Servetto, E. Zucca, Flattening versus direct semantics for Featherweight Jigsaw, in: FOOL, 2009.
[44] A. Lienhard, S. Ducasse, G. Arévalo, Identifying traits with formal concept analysis, in: ASE, IEEE, 2005, pp. 66–75.
[45] M. Limberghen, T. Mens, Encapsulation and composition as orthogonal operators on mixins: a solution to multiple inheritance problems, Object Oriented Systems 3 (1) (1996) 1–30.
[46] L. Liquori, A. Spiwack, FeatherTrait: a modest extension of featherweight Java, ACM TOPLAS 30 (2) (2008).
[47] B. Meyer, Object-Oriented Software Construction, 2nd edition, Prentice-Hall, 1997.
[48] E.R. Murphy-Hill, P.J. Quitslund, A.P. Black, Removing duplication from java.io: a case study using traits, in: OOPSLA, ACM, 2005, pp. 282–291.
[49] O. Nierstrasz, S. Ducasse, N. Schärli, Flattening traits, JOT 5 (4) (2006) 129–148.
[50] M. Odersky, The Scala Language Specification, version 2.4, Technical report, Programming Methods Laboratory, EPFL, 2007.
[51] T. Parr, The Definitive ANTLR Reference: Building Domain-Specific Languages, Pragmatic Programmers, May 2007.
[52] M. Pfeiffer, J. Pichler, A comparison of tool support for textual domain-specific languages, in: DSM, 2008, pp. 1–7.
[53] P.J. Quitslund, Java traits — improving opportunities for reuse, Technical Report CSE-04-005, OGI School of Science & Engineering, Beaverton, Oregon, USA, Sept. 2004.
[54] P.J. Quitslund, R. Murphy-Hill, A.P. Black, Supporting Java traits in eclipse, in: ETX, ACM, 2004, pp. 37–41.
[55] D. Remy, J. Vouillon, Objective ML: an effective object-oriented extension to ML, Theory and Practice of Object Systems 4 (1) (1998) 27–50.
[56] L. Renggli, M. Denker, O. Nierstrasz, Language boxes: bending the host language with modular language changes, in: SLE, in: LNCS, vol. 5969, Springer, 2009, pp. 274–293.
[57] J. Reppy, A. Turon, A foundation for trait-based metaprogramming, in: FOOL/WOOD, 2006.
[58] J. Reppy, A. Turon, Metaprogramming with traits, in: ECOOP, in: LNCS, vol. 4609, Springer, 2007, pp. 373–398.
[59] T. Sauer, K. Maximini, R. Maximini, R. Bergmann, Supporting collaborative business through integration of knowledge distribution and agile process management, in: MKWI, GITO-Verlag, 2006, pp. 349–361.
[60] N. Schärli, S. Ducasse, O. Nierstrasz, A. Black, Traits: composable units of behavior, in: ECOOP, in: LNCS, vol. 2743, Springer, 2003, pp. 248–274.
[61] C. Smith, S. Drossopoulou, Chai: traits for Java-like languages, in: ECOOP, in: LNCS, 3586, Springer, 2005, pp. 453–478.
[62] A. Snyder, Encapsulation and inheritance in object-oriented programming languages, in: OOPSLA, vol. 21(11), ACM, 1986, pp. 38–45.
[63] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks, EMF: Eclipse Modeling Framework, 2nd edition, Addison Wesley Professional, 2008.
[64] R. Strniša, P. Sewell, M. Parkinson, The Java module system: core design and semantic definition, in: OOPSLA, ACM, 2007, pp. 499–514.
[65] F. Strocco, A Java dialect oriented to fine-grained software reuse, Bachelor thesis, Dip. di Informatica, Università di Torino, 2009.
[66] D. Ungar, C. Chambers, B.-W. Chang, U. Hölzle, Organizing programs without classes, Lisp and Symbolic Computation 4 (3) (1991) 223–242.
[67] H. Xu, Erilex: an embedded domain specific language generator, in: TOOLS, in: LNCS, vol. 6141, Springer, 2010, pp. 192–212.