Generation of Inductive Types from Ecore Metamodels

Jérémy Buisson, Seidali Rehab

Generation of Inductive Types from Ecore Metamodels

Jérémy Buisson[1] and Seidali Rehab[2]

[1] IRISA, Écoles de Saint-Cyr Coëtquidan, Guer, France,
jeremy.buisson@irisa.fr
[2] MISC, University of Constantine 2 - Abdelhamid Mehri, Nouvelle ville Ali Mendjeli, Constantine, Algeria,
seidali.rehab@misc-umc.org

Abstract. When one wants to design a language and related supporting tools, two distinct technical spaces can be considered. On the one hand, model-driven tools like Xtext or MPS automatically provide a compilation infrastructure and a full-featured integrated development environment. On the other hand, a formal workbench like a proof assistant helps in the design and verification of the language specification. But these two technical spaces can hardly be used in conjunction. In the paper, we propose an automatic transformation that takes an input Ecore metamodel, and generates a set of inductive types in Gallina and Vernacular, the language of the Coq proof assistant. By doing so, it is guaranteed that the same abstract syntax as the one described by the Ecore metamodel is used, *e.g.*, to formally define the language's semantics or type system or set up a proof-carrying code infrastructure. Improving over previous state of the art, our transformation supports structural elements of Ecore, with no restriction. But our transformation is not injective. A benchmark evaluation shows that our transformation is effective, including in the case of real-world metamodels like UML and OCL. We also validate our transformation in the context of an ad-hoc proof-carrying code infrastructure.

Keywords: Model-Driven Engineering, Model Transformation, Inductive Type, QVT-Operational, Ecore, Xtext, Coq

1 Introduction

In this paper, we present our work that is specifically related to the implementation of support tools for a formal architecture description language for system of systems engineering, named SosADL [1]. In this paper, we do not intend to describe this novel architecture description language. We would like to put the emphasis on difficulties that arise when we want to benefit from convenient tools like Xtext [6] and, at the same time, formally ensure language properties by means of proofs.

Nowadays the creation of a language and its infrastructure becomes easier thanks to the tools that model-driven engineering offers. Among these effective tools we find MPS [24] or Xtext [6]. A complete editing environment can be generated from a combined description of concrete and abstract syntax. This can include syntax-highlighting, auto-completion and elaborated error reporting. A compilation or interpretation framework accompanies these tools in order to smoothly interact with the generated editing

environment. For SosADL we choose Xtext to benefit from the mature Ecore/EMF ecosystem.

In the formal side, principled language design has been promoted by language theory. This is done using well-established techniques to specify a language in terms of, *e.g.*, semantics and type system, and then prove that this specification is sound. Proof techniques, relevant properties and proof techniques have been proposed in the meta-theory related to language theory. And several proof assistants as Coq [5] or Isabelle/HOL [19] have been successfully used to mechanize such specifications and proofs. For SosADL we choose Coq.

The problem in our work is how to ensure the automatic transformation between the two technical spaces in order to benefit from their crossed contributions, so that the disadvantages of one can be overcome thanks to the contributions of the other. This question comes from the fact that there is a difficulty of integration between the model-driven engineering tools like Xtext or MPS and the proof assistants. Some (model-driven engineering) researches often rely on graph-based modeling, while the others (proof assistants) use inductive data types, despite some exceptions such as Rascal [13]. In other words, we need to make sure that both sides, that is, informal implementation in the Ecore/EMF technical space, and the formal specification in the Coq technical space are consistent. To overcome this problem, we base SosADL tools on the proof-carrying code approach [18]. Therefore, we aim to generate automatically large parts of the infrastructure for proof-carrying code.

In this paper, we study how to generate Coq types for the abstract syntax tree from an Ecore metamodel, such that the abstract syntax of the language is shared in the two technical spaces. Our contribution is an improved transformation in comparison to prior state of the art: constraints on the input metamodel are relaxed, especially with respect to inheritance, to the detriment of not being injective.

The work described in this paper is an extended version of our earlier publication [8]. In addition to more detailed description of the transformation, we better describe its implementation, and more specifically the transformation framework we have designed. We extend the test suite that we use to validate our transformation with real-world metamodels. And we extend the validation with some discussion of execution time.

Section 2 describes the overall context of our work. Section 3 gives a brief description of Ecore and of inductive types. Section 4 presents related work about transforming from metamodels to inductive types. Section 5 describes a running example that we use in the subsequent description of the transformation, given in Section 6. Section 7 discusses specific points, noticeably why we consider having an injective transformation is not that important in our case. Section 8 gives indications about implementation issues. Section 9 summarizes how we validate the transformation. Finally, section 10 concludes the paper with perspectives.

2 Context and Motivations

When one wants to use model-driven techniques to implement the supporting tools for a formally-defined language, the question arises how to ensure the implementation

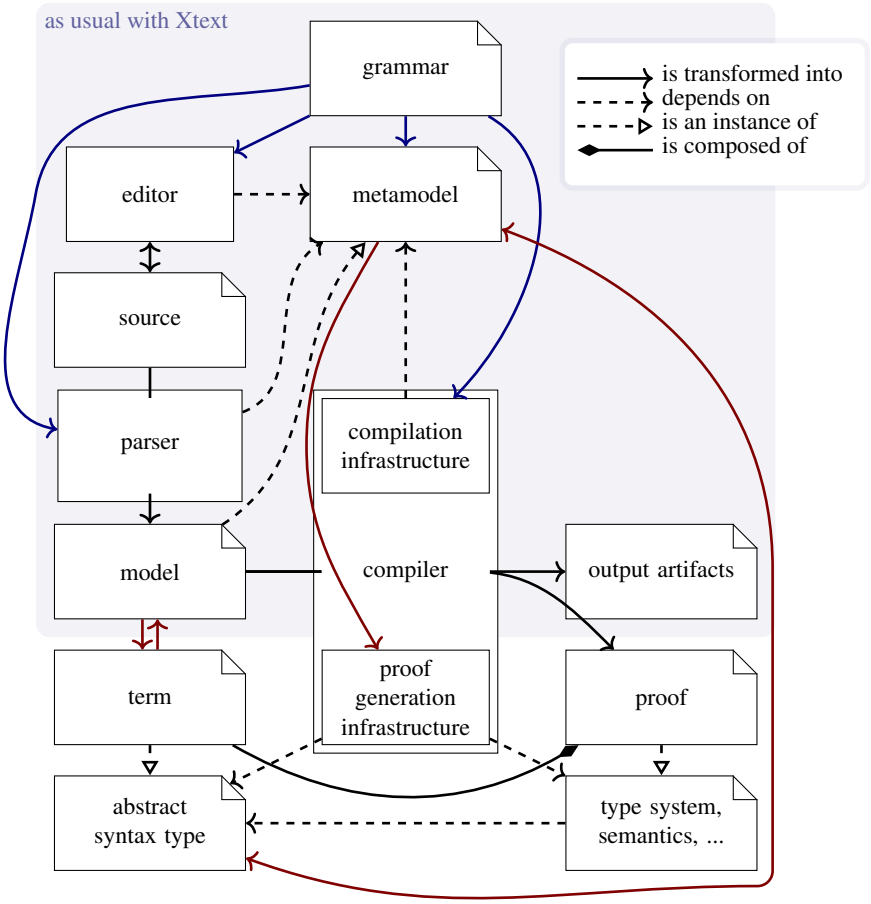


Fig. 1. The big picture of our general approach.

actually conforms to the formal definition of the language. In this paper, we consider that this issue is solved by means of proof-carrying code [18].

With the proof-carrying code, the compiler (or any supporting tool for the language) not only generates compiled code. It also generate a verifiable proof of properties about the source or compiled code. By checking the validity of the proof, one can easily ensure that the compiler performed correctly. For instance, the Java compiler generates annotations in the compiled byte-code such that the Java virtual machine can easily ensure that the generated byte-code has been correctly generated, without having to perform complete type checking.

Our general approach described in Figure 1 explains the proof-carrying code approach in the way we apply it to the SosADL [1] context. This one is divided into two parts. The first (upper) part of the figure, which aims to produce at the end *output artifacts*, starts with the generation of our complete editing environment from a concrete

grammar. This operation, which generates a *metamodel*, an *editor*, a *compilation infrastructure* and a *parser*, is performed using Xtext [6]. Thanks to this generated code, the *model* that is an instance of the metamodel is obtained from the textual *source* thanks to the parser. Then, we develop a *compiler* using the compilation infrastructure, which (the compiler) is in turn the tool for transforming the object-oriented model to the output artifacts. The second (bottom) part of the figure gives the explanation of the proof-carrying code. The *proof generation infrastructure* is used by our compiler to provide a *proof*. This proof is an instance of the language's specification, *i.e.*, of the *type system* or *semantics*. *Terms* compose this proof, which represent instances of *abstract syntax type*. In addition, these terms map with models, and thus we have to ensure a strong consistency between the metamodel and the abstract syntax type, that is, any object that is an instance of a metamodel class maps to a term whose type come from the abstract syntax, and conversely.

3 Background

In this section, we introduce Ecore and inductive types, which are the two languages which we consider to express an abstract syntax.

3.1 Ecore

An Ecore [22] metamodel is an object-oriented description of the abstract syntax of a modeling language. An Ecore metamodel consists in hierarchically-nested *packages*, where the root packages are the Ecore files. Each package contains several *classes*, which describe the types of objects that can exist in instance models. As usual in object-oriented modeling languages, Ecore supports inheritance and subtyping by means of a specialization-generalization relationship between classes. A specific kind of classes, *abstract* classes, denote classes that cannot be instantiated in models; only their concrete, i.e., non-abstract specialization classes can be.

An Ecore class contains *structural features*, namely, the fields declared by the class. In Ecore, fields can be either *attributes* or *references*. The former (attributes) denote fields whose type is a plain Java type imported as a *data type*; while the latter (references) are fields whose type is described by an Ecore type. References are further refined as either *containment* references or non-containment references, where the semantics of containment is the same as UML's composition.

An instance model, which is an instance of a metamodel, is a tree of objects (according to the containment references) whose classes are the classes described in the metamodel. Additional non-containment references allow to describe arbitrary graphs. An instance model either resides in memory or is serialized to an XMI file.

Orthogonal to the attribute/reference classification, structural features have additional properties. To transparently represent collections, each structural feature has multiplicity indication. The actual kind of collection is further refined by uniqueness and ordering properties. In addition, a feature is said *derived* when its value is computed on-demand; *transient* when it is omitted from XMI serialization; or *volatile* when it is omitted from the in-memory object.

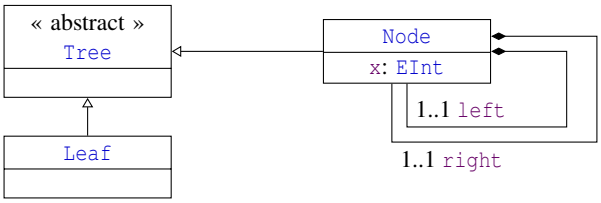


Fig. 2. Example of a simple Ecore metamodel.

Ecore classes also contain *operations*. But operations are not relevant in the context of this work.

Figure 2 shows an example metamodel, which contains an abstract class named `Tree` specialized by two concrete classes `Leaf` and `Node`. So only `Leaf` and `Node` can be instantiated in models. While `Leaf` has no structural feature, `Node` contains an attribute `x`, whose type is data type `EInt`, the data type that imports Java's `int` primitive type, and two containment references (containment is denoted by the diamond) named `left` and `right`. The attribute and the references all have here 1..1 multiplicity, meaning that each instance of `Node` contains exactly one `left Tree` and one `right Tree`.

3.2 Inductive Types

Inductive type is a usual approach to the definition of data types in the context of functional programming. Without lack of generality, in the following, we consider the specific case of Gallina and Vernacular, Coq's languages for terms and for module-level commands.

Each inductive type is a set of *constructors*, each of which declares a variant of the type. The data structure associated with a given constructor is specified by the formal parameters of that constructor. Types are gathered in hierarchical *modules*, where each file is a module too.

To illustrate, consider the following example in Gallina and Vernacular:

```
Inductive Tree: Set :=
| Leaf: Tree
| Node: Tree → nat → Tree → Tree.
```

This code reads as the definition of an inductive type named Tree, for which two constructors are defined:

- When a Tree is built by the Leaf constructor, its does not contain any data: the type of Leaf is Tree.
- When a Tree is built by the Node constructor, it contains a Tree, a nat (a natural integer), and another Tree. The type of Node states that it is a function that takes 3 parameters and returns a Tree.

Any value is the result of calling the constructors. A term is therefore inherently a tree that mimics the abstract syntax tree of itself, like:

```
Definition x := Node (Node Leaf 2 Leaf) 6 Leaf.
```

4 Related Works

Language theory provides background on how one specifies a language and what properties of such specification should be investigated for in order to convince that the language specification is sound. The idea to extend the theoretical approach to languages with effective tools to help in the implementation of language supporting tools has already been investigated, like witnessed by several projects. For instance, Ott [21] is a tool that, given a single description of inference rules, *e.g.*, for semantics or type systems, generates boilerplate code, as well as definitions suitable for mechanization with a proof assistant and LaTeX-based documentation. Lem [17] goes one step further and generates in addition executable functions from inductive relations encoding operational semantics or type systems. The $\mathbb{K}$-framework [2] is a comprehensive approach to generate interpreters and tools from the executable semantics of a language.

More than just interpreters or tools, it may be convenient to generate compilation infrastructure, as well as full-featured editors and development environments. The idea is not new like witnessed by older projects like Centaur [7] and ASF+SDF [12]. It has been renewed in the context of model-driven engineering and domain-specific languages, that rise the issue of routinely producing supporting tools and editors for many languages. In this perspective, Xtext [6] generates a full-featured text editor and a compilation framework based on the Eclipse IDE, from a combined description of concrete and abstract syntax. MPS [24] is based on projectional editor, *i.e.*, edition is made directly at the level of the abstract syntax. MPS also provides a declarative language for executable type systems. Other language workbenches like Spoofax [11] and Rascal [14] have similar objectives of generating both compilation infrastructure and IDE services.

There is therefore a need for bridging the gap between model-driven engineering, which enables automatic generation of convenient IDE from the language definition, and formal methods that allows precise specification and analysis of the language and its semantics. To bridge this gap, one possible approach consists in generating a formal description of the language from a metamodel, or conversely in generating a metamodel from a formal language description.

Several previous work have studied how one can analyze a metamodel thanks to tools coming from formal methods, *e.g.*, [4,3,16,15]. These work usually address the question whether a metamodel is consistent, especially when the metamodel is partly specified by constraints such as OCL constraints. These work propose approaches to decide whether the metamodel is inhabited and contradiction-free, *i.e.*, whether some instance model exists and conforms to the metamodel. Since they try to prove properties of the metamodel, the main object made available in the proof assistant is the metamodel itself.

In Section 2 and Figure 1, we have depicted a slightly different issue. What we are interested in is ensuring, here by means of proof-carrying code, that an analysis or a transformation conforms to its specification, and, of course, to encode this specification. In this context, the main object we would like to manipulate in the proof assistant is *instances* of the metamodel, and not the metamodel itself. So the metamodel has to be transformed into types, such that instances of this metamodel could be transformed into manipulable terms conforming to these types.

Such a transformation has already been studied in [9]. Authors have defined a bidirectional transformation between Ecore and Isabelle's inductive type: each abstract class `A` is mapped to an inductive type `t`; and each concrete class `C` is transformed to a constructor `c` of the inductive type `a` mapped from the super class `A` of `C`. Fields of class `C` are mapped to formal parameters of constructors `c`. Examples given in Section 3 illustrate this transformation scheme on a simple case. To apply this transformation, the metamodel must conform to the strict pattern of having only abstract classes without any super type nor any field, and each concrete class must specialize exactly one abstract class. The transformation is injective. But multiple inheritance, or even having a class that specializes another class that specializes yet another class is prohibited.

Rascal [13] comes with another similar transformation. Instead of restricting the metamodel, the transformation described in [13] leverages a preprocessing step of the metamodel before the scheme of [9] is used. First, structural features are pushed to concrete classes (same as our step ⑥ in Figure 4); then references are generalized, *i.e.*, a reference to any class `C` is replaced with a reference to `C`'s most general super class, hence flattening the inheritance tree to two levels. After this preprocessing step, the metamodel obviously conforms to the restrictions required by [9]. The second step of preprocessing intrinsically assumes that a most general super class exists for any class. But in presence of multiple inheritance, this assumption may not hold, unless a class implicitly generalizes all the other classes, *e.g.*, like Ecore [22]'s `EObject` or Java's `Object` classes. And if such a class exists, it turns out that the transformation issues a single inductive type for this class, to which all the constructors belong. In the end, terms manipulated in the proof assistant are therefore untyped, what is undesirable.

This last comment is the issue we address in this paper. In Section 6, we follow the same principles as [9,13], but without any restriction on the input metamodel. Furthermore, when multiple inheritance is not used, our transformation generates narrower types than the transformation of [13] by duplicating constructors. As counterpart, generating a term from a model is going to be harder, because the right constructor has to be selected with respect to the expected type for the term. Like in [13], our preprocessing steps break injectivity.

5 A Running Example

To illustrate the discussion in subsequent sections, we use the metamodel for λ terms of Figure 3. A `File` is composed of `Definition`s, each containing a `Term`. A term is either an `Abstraction`, an `Application` or a `Variable`. In order to avoid issues related to naming and scopes, the abstract syntax assumes variables have already been resolved, hence `Variable` has a non-containment reference to `Binder`, which is either an abstraction or a definition. Classes are generic such that terms can be annotated, *e.g.*, with types.

We expect that our transformation generates the following Coq script (or equivalent), *i.e.*, inductive types such that any model that is an instance of the source metamodel can be written as a term whose type is one resulting from the transformation.

```
Inductive _Term: Type → Type :=
| Term_Abstraction: ∀ (A: Type) (type: option A) (boundName: string)
    (boundType: option A) (body: _Term A), _Term A
```

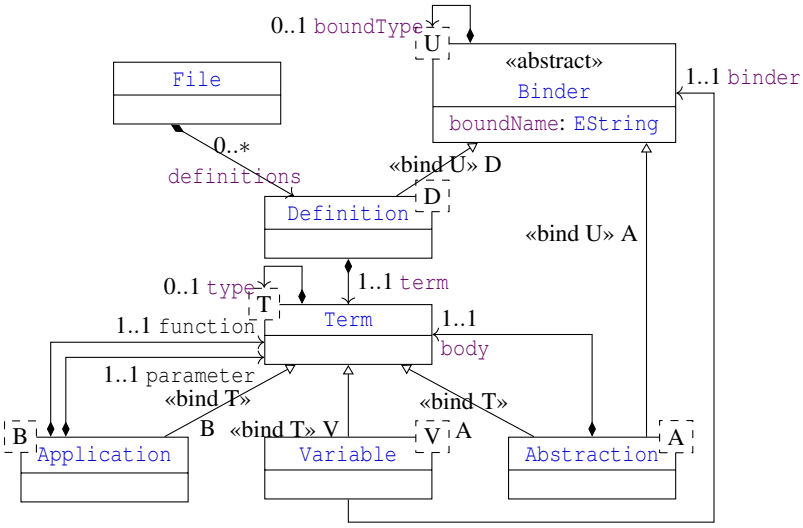


Fig. 3. Ecore metamodel of the example. Source [8].

```
| Term_Application: ∀ (A: Type) (function: _Term A) (parameter: _Term A), _Term A
| Term_Variable: ∀ (A: Type) (binder: _URI (_Binder A)), _Term A
with _Binder: Type → Type :=
| Binder_Definition: ∀ (D: Type) (boundName: string) (boundType: option D)
    (term: _Term D), _Binder D
| Binder_Abstraction: ∀ (A: Type) (type: option A) (boundName: string)
    (boundType: option A) (body: _Term A), _Binder A.
Inductive _Definition: Type → Type :=
| Definition_Definition: ∀ (D: Type) (boundName: string) (boundType: option D)
    (term: _Term D), _Definition D.
Inductive _File: Type :=
| File_File: ∀ (definitions: list (_Definition _Type)), _File.
```

6 The Transformation

The transformation that we propose is decomposed into 12 steps, like shown in Figure 4. It follows the same principle as [9,13]: it maps classes to inductive types, concrete classes to constructors, and structural features to constructor parameters. In our transformation, the improvements lie the preprocessing of the metamodel, especially to handle multiple inheritance in the source metamodel, as well as in the more elaborate post-processing specifically targeted at Gallina and Vernacular. Our transformation does not consider any behavioral element, such as operations and derived, transient or volatile features, as these elements are irrelevant with respect to the context depicted in Section 2.

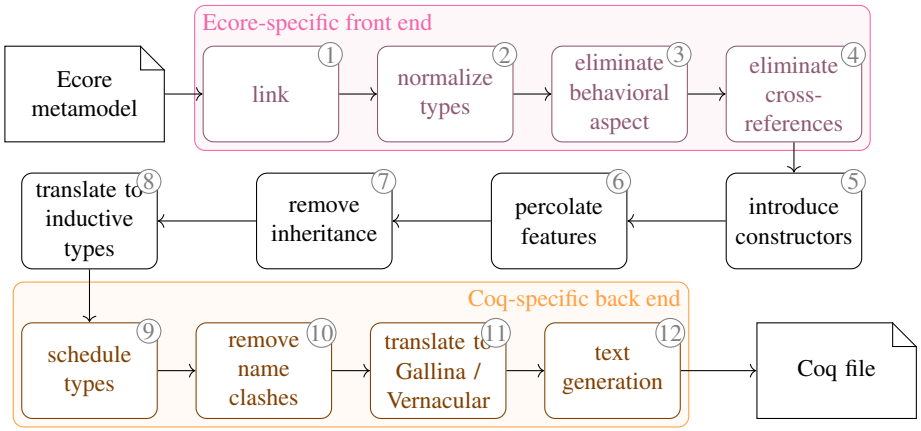


Fig. 4. Decomposition of the Ecore-to-Coq transformation.

To describe the transformation, we begin first with the Ecore-specific front-end, i.e., steps ① to ④ in Section 6.1. Then, in Section 6.2, we present the main steps ⑤ to ⑧ of the transformation. Last, Section 6.3 describes the Coq-specific back-end containing steps ⑨ to ⑫.

6.1 Ecore-specific Front-end

The goal of the front-end step is to first transform the Ecore metamodel in order to normalize and simplify its representation. The first step ① erases the concept of package: all the classes are gathered in a single top-level package and a single file. Indeed, it would be tempting to map packages to modules. However, while each class can be member of any package, regardless of the relationships between classes, the situation is different in the context of inductive types. The constructors of an inductive type belong to that type, and therefore they are all members of the same module. But like explained in Section 6.2, constructors are mapped from classes. If the packages were not ignored, our transformation would attempt to generate the constructors of a single inductive type within several distinct modules, which is forbidden in the target language.

To illustrate more concretely, consider an abstract class `A` in package `p1`, specialized by concrete class `C` of package `p2`. In Section 6.2, we will see that class `C` maps to constructor C, which belongs to type A, itself being mapped from class `A`. If packages were mapped to modules, type A should be in module `P1` and constructor C should be in module `P2`. This is impossible since constructors belong to types, not to modules.

To show the effect of step ①, Figure 5 gives an excerpt of the result when applied to the metamodel of Figure 3. The metamodel is almost unchanged. Still, classes from the Ecore metamodel are pulled into the single root package, starting with `EObject` because it is used as the raw type of `type` in `Term`. Because of `boundName` in `Binder`, `EString` is pulled too. Any other class that `EObject` depends on is pulled as well, and the process is repeated until all the dependencies are gathered. Like shown in the XMI excerpt of Figure 5, the transformation encodes the name of the originating package in

```
1 <LEPackage>
2   <eClassifiers name="ecore_EString" ... />
3   <eClassifiers name="lambda_Binder"
4                 eSupertypes="//ecore_EObject" ...>
5     <eTypeParameters name="U" />
6     <eStructuralFeatures name="boundName" ... />
7     <eStructuralFeatures name="boundType" ...>
8       <eGenericType eTypeParameter="//lambda_Binder/U" />
9     </eStructuralFeatures>
10  </eClassifiers>
11  ...
12 </LEPackage>
```

Fig. 5. XMI excerpt after step 1 *link*. Source [8].

the class name. By using this systematic renaming scheme, the transformation avoids name clashes after all the classes are gathered in a single package.

In step ②, the transformation deals with the representation of types in Ecore. In its early versions, Ecore did not support generic classes. In this context, types and classes were confused in a single concept. Thus, the type of, *e.g.*, a reference or an attribute were given as a direct reference to the data type or to the class. Since the introduction of generic classes in Ecore, types and classes have been made two distinct concepts, yielding to the introduction of `EGenericType` to represent types. But due to backward compatibility, old-style types are still allowed when referring to non-generic data types or classes. Figure 5 contains examples of both:

- The super class of `lambda_Binder` (line 4) is given as a direct reference to the `ecore_EObject` class, which is indeed a non-generic class.
- The type of `boundType` (line 8) is given as an instance of `EGenericType`, here to represent type variable `U`.

Step ② translates all the types to a simpler uniform representation, whose abstract syntax is given in Figure 6, regardless the types were initially given old-style or new-style. We define five kinds of types:

- `GEClassifierType` is a type built by applying effective parameters (`eArguments`) to a possibly generic classifier, *i.e.*, either a data type or a class. If the referenced classifier is not generic, *i.e.*, if it does not have any formal type parameter, then `eArguments` is empty.
- `GEVariableType` is a type variable `eTypeParameter`, *e.g.*, bound by an enclosing generic classifier.
- A `GEAnyType` represents the any-type wildcard.
- `GELowerBoundType` is a wildcard type with a lower bound.
- `GEUpperBoundType` is a wildcard type with an upper bound.

Figure 7 shows an excerpt of the resulting XMI file, in our running example. For instance, the super-class of `lambda_Binder`, which where initially given as an old-style

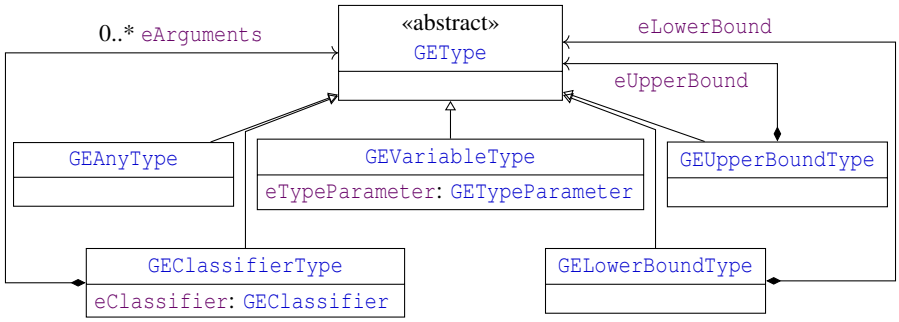


Fig. 6. Abstract syntax for types.

direct reference to `ecore_EObject`, is now encoded as an instance of `GEClassifierType` (lines 5 and 6); and the type of `boundType` is an instance of `GEVariableType` referring to formal type parameter `U` of `lambda_Binder` (lines 14 and 15).

Steps ③ erases behavioral elements, that is, operations and derived, transient or volatile features. In step ④, all the non-containment references are replaced with attributes of type `_URI`. These attributes are intended to store identifiers of the referred objects. Feature multiplicities are expanded to appropriate collection types at the same time, hence completing the simplification of the types:

- Features with 0..1 multiplicity are mapped to type `_Option`.
- Features with 1..1 multiplicity keep their original type.
- Features with $m..n$ multiplicity where $n \geq 2$ are mapped to types `_List`, `_Set` or `_Bag`, depending on uniqueness and ordering properties stated in the source Ecore metamodel.

In subsequent steps of the transformation, these collection types are suitably interpreted such that they are ultimately mapped to corresponding Coq types.

6.2 Core of the Transformation

During steps ⑤ to ⑦, constructors are added to the classes, before they can be turned into inductive types.

First, in step ⑤, each concrete class of the metamodel is mapped to a newly-created constructor. Figure 8 illustrates the result of step ⑤: in this excerpt, two constructors are created for the two concrete classes `Abstraction` and `Definition`. Each constructor refers to the structural features defined or inherited by the corresponding class.

Then in step ⑥, structural features are pulled down and cloned into the constructors, through inheritance. Like shown in the excerpt of Figure 9, the structural features are duplicated to all the constructors that inherit from them. Step ⑥ also ensures correct handling of generic classes by substituting type variables when the structural features are inherited, like illustrated by `type` and `boundType` in Figure 9.

The last step that deals with constructors is step ⑦. This step duplicates constructors previously built up at each level of the generalization relation, including at abstract

```
<GEPackage>
  <eClassifiers xsi:type="GEClass"
                name="lambda_Binder" abstract="true">
    <eTypeParameters name="U"/>
    <eSuperTypes xsi:type="GEClassifierType"
                 eClassifier="//ecore_EObject"/>
    <eStructuralFeatures xsi:type="GEAttribute"
                         name="boundName" lowerBound="1">
      <eType xsi:type="GEClassifierType"
             eClassifier="//ecore_EString"/>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="GEReference"
                         name="boundType" containment="true">
      <eType xsi:type="GEVariableType"
             eTypeParameter="//lambda_Binder/U"/>
    </eStructuralFeatures>
  </eClassifiers>
  ...
</GEPackage>
```

Fig. 7. XMI excerpt after step 2 *normalize*.

classes. For instance, because the `Abstraction` class inherits from the `Term` and `Binder` abstract classes, the `Abstraction` constructor is duplicated at these two classes. Because `Binder` is also a generalization of the `Definition` class, the `Binder` class is also made containing a duplicate of the `Definition` constructor. At the end of step ⑦, the generalization / specialization relation can be discarded. For each constructor duplicate, an assignment stores the precise type of the value built, hence taking into account of generic classes correctly.

Step ⑧ straightforwardly turns each class into an inductive types, without any further transformation.

6.3 Coq-specific Back-end

In Gallina and Vernacular, it is forbidden that an inductive type refers to another type that is not previously defined or that does not belong to the same group of definitions. Inductive type definitions must be ordered accordingly to their dependencies. To do so, step ⑨ computes strongly connected components in the dependency graph in order to build groups of types, then this step sorts the groups according to topological order.

For instance, in the result of step ⑨, types `Binder` and `Term` shall be in the same group, since they refer each other. Indeed, constructor `Variable` (in `Term`) has a parameter of type `_URI<Binder<V>>`; and constructor `Definition` (in `Binder`) has a parameter of type `Term<D>`. This group is put before `Definition`, which refers to `Term`.

Step ⑩ ensures that each name is unique, as well as conforms to lexical constraints of Gallina and Vernacular. Step ⑪ introduces Vernacular commands (like `Inductive`) and builds Gallina terms for each type to build a correct script. Step ⑫ generates the

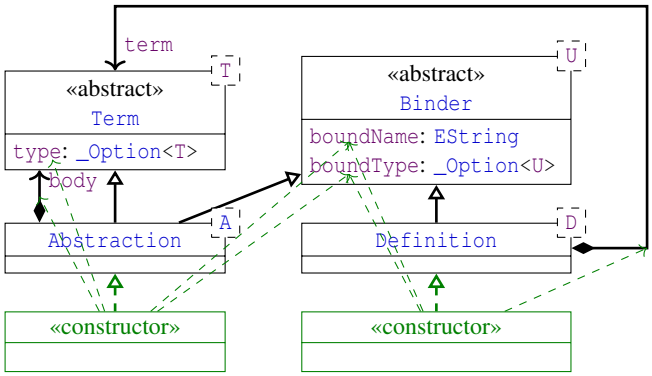


Fig. 8. Metamodel excerpt after step 5 *introduce*. Source [8].

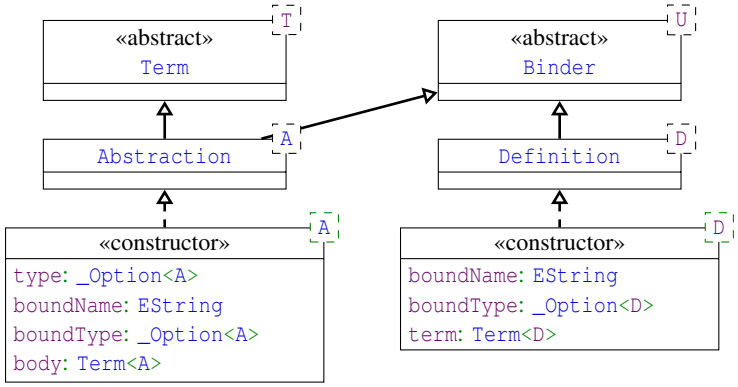


Fig. 9. Metamodel excerpt after step 6 *percolate*. Source [8].

text file. For our running example, the result is equivalent to the desired ones given at Section 5:

```
Definition ecore_EString: Type := string.
Definition ecore_EInt: Type := Z.
Definition ecore_EEList: (Type → Type) := list.
    (* ... *) Inductive lambda_Term: (Type → Type) :=
| lambda_Term_lambda_Abstraction: (∀ (A: Type), (∀ (body: (lambda_Term A)),
    (∀ (boundName: ecore_EString), (∀ (boundType: (_Option A)),
    (∀ (type: (_Option A)), (lambda_Term A))))))
| lambda_Term_lambda_Application: (∀ (B: Type), (∀ (function: (lambda_Term B)),
    (∀ (parameter: (lambda_Term B)), (∀ (type: (_Option B)), (lambda_Term B))))))
| lambda_Term_lambda_Variable: (∀ (V: Type), (∀ (binder: (_URI (lambda_Binder V))),
    (∀ (type: (_Option V)), (lambda_Term V))))
with lambda_Binder: (Type → Type) :=
    (* and so on *)
```

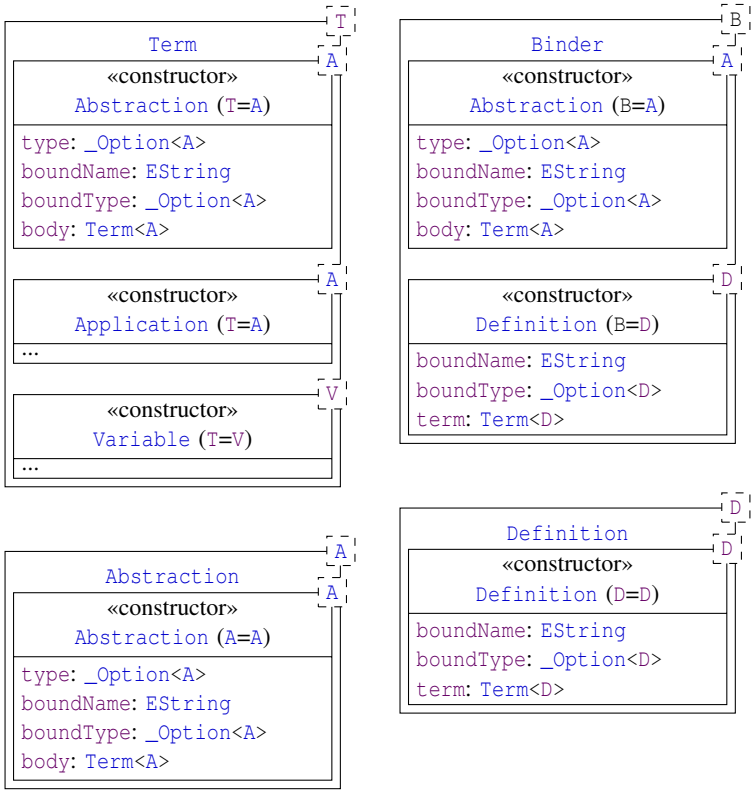


Fig. 10. Metamodel excerpt after step 7 *flatten*. Source [8].

6.4 Traceability

Orthogonal to the previously described steps, the transformation ensures traceability. That is, it records mapping information between objects in the source metamodel, objects in intermediate steps and objects in the Gallina model. This information is subsequently used when instances of the source metamodel have to be transformed into Gallina terms.

7 Discussion

Our transformation restricts to metamodels that strictly conform to Ecore constraints as implemented in EMF, including constraints at the *warning* level.

In addition, the following patterns are ignored:

– When a multi-valued reference in an Ecore metamodel refers to a class that contains two features named `key` and `value`, and when this class's instance class name is `java.util.Map$Entry`, the Ecore tool chain handles this reference as a hash map. Our transformation ignores the pattern, yielding to a collection of pairs.

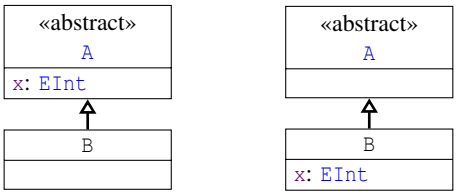


Fig. 11. Two different metamodels yielding to identical Coq scripts. Source [8].

– When a multi-valued attribute has type `EFeatureMapEntry` and if it is suitably annotated, the attribute declares a feature map, that is, a structural feature that merges several subset structural features in a single one. Values in the collection are indexed by subset features. Because the subset features must be volatile, transient and derived, they are erased at step ③ of our transformation. But the attribute's type is not decoded to the correct type, which should be the union of the types of all the grouped subset features.

Except the above-described limitations, Ecore is fully supported. Hence the transformation cannot be injective. To illustrate the reason, consider Figure 11: the two metamodels, despite different, result in identical Coq scripts. When the structural features are pulled from classes to constructors at step ⑥, our transformation does not track the class they originate from. By the way, this information is irrelevant in Gallina.

The fact that the two metamodels yield to the same Coq script is not an issue: any model that is an instance of any metamodel of Figure 11 is also an instance of the other one. Indeed, these models contain only instances of `B` that contain an integer named `x`, regardless `x` is declared in `B` or inherited from one of its super classes.

Still, we can further analyze what steps of the transformation are injective, and what steps are not:

– Steps ⑤, ⑧, ⑪ and ⑫ are straightforwardly injective.
– When the naming scheme is robust enough, steps ① and ⑩ are obviously injective too.
– Step ④ is not injective, mainly because multiplicities are simplified as one of 0..1, 1..1 or 0..∗. Though, Gallina would allow to preserve any exact multiplicity if desired.
– Step ② is trivially not injective because Ecore's representation of non-generic types is not unique. But making the distinction between old-style representation and `EGenericType` is irrelevant as both are considered as interchangeable by the Ecore tool chain.
– Step ⑨ discards the order of metamodel elements.
– Steps ③, ⑥ and ⑦ discard information from the metamodels.

Like described in Section 6, the transformation is a pipeline decomposed in many steps. This approach makes each individual step simpler as each one focuses on a single issue (or few related issues). The first steps ① to ④ of the transformation are specific to Ecore, and steps ⑨ to ⑫ are specific to Gallina and Vernacular. If other source

or target languages were considered, only the related steps would have to be changed. Consider MOF [20] for instance: according to [10], older versions of MOF and Ecore are convertible to one another. To adapt the transformation to MOF, step ① has to deal with nested classes, e.g., similarly to the way packages are merged; step ④ has to deal with MOF's richer reified associations; and data type mapping has to be updated, since MOF is not based on Java types. Likewise, switching to, say, Isabelle/HOL would require changing steps ⑩ and ⑪ in order to take into account the different abstract and concrete syntax.

8 Implementation

The decomposition of the transformation in multiple steps allows us to mix several implementation technologies. Our overall choice is in favor of industrial-strength freely-available technologies: Eclipse's mature EMF-and-Java ecosystem. Model-to-model transformations are implemented in plain Java and QVT-Operational. The final model-to-text step ⑫ uses Acceleo.

Steps ① to ⑩ perform only local modification. At each of these steps, when considered individually, the transformation rebuilds the overall structure of the source (meta)model into the target one, only substituting some specific subtrees. Rather than using a general-purpose approach to model transformation, we design an ad-hoc framework that duplicates an EMF (meta)model with hooks to customize its behavior at some features of some classes. Our framework is a mixture of EMF's `EcoreUtil.copy`, ATL's refining mode [23] and Rascal's `visit` operation. But unlike these inspiring approaches, our framework is not restricted to endogeneous transformations, and therefore adopts a copy-based strategy.

The overall algorithm of our framework is given, in pseudo-Java code, in Figure 12. At lines 2 and 3, the algorithm uses EMF's reflection to clone the object at the root of the subtree that must be transformed. The `remapClass` hook lets the transformation change the class of the object on the fly. Then at line 4, the algorithm records the mapping between source and target objects, in order to later resolve references. Each structural feature of the target class (line 5), is mapped back to a structural feature of the source class (line 6) by the `remapFeatureBack` hook.

- If the target feature is a containment feature, values are assigned immediately (line 10). By default, the `remapValue` hook calls the `clone` function in order to recursively copy model objects.
- If none of the source feature and target feature is a containment feature, the task is registered for post-processing (lines 14 and 15). After the source model has been fully handled, post-processing tasks call the `remapReference` (lines 26 and 27), which, by default, looks in the mapping table to set the reference in the target object (line 28).
- With our framework, it is non-sense when the source feature is a containment feature, while the target feature is not. Line 17, our algorithm raises an error in this case.

```
1 clone(sourceObject) {
2   targetClass = remapClass(class of sourceObject)
3   targetObject = new object of type targetClass
4   mapping.put(sourceObject, targetObject)
5   for (targetFeature: targetClass.features) {
6     sourceFeature = remapFeatureBack(targetFeature)
7     if(targetFeature.isContainment) {
8       for (v: remapValue(values of sourceFeature
9                                  from sourceObject)) {
10        add v to targetFeature of targetObject
11      }
12    } else if(!sourceFeature.isContainment
13              && !targetFeature.isContainment) {
14      register (sourceObject,sourceFeature,
15                targetObject,targetFeature)
16    } else {
17      error
18    }
19  }
20  return targetObject
21 }
22
23 post-processing {
24  for(sourceObject,sourceFeature,
25      targetObject,targetFeature: registrations) {
26    for (v: remapReference(values of SourceFeature
27                                  from sourceObject)) {
28      add v to targetFeature of targetObject
29    }
30  }
31 }
```

Fig. 12. Pseudo-code of our Java-based transformation framework.

On top of this framework, we provide an implementation of the hook functions (`remapClass`, `remapFeatureBack`, `remapValue` and `remapReference`) based on Java reflection, yielding to simpler description of the transformation steps. Figure 13 illustrates our framework with the real source code for step ③. In the constructor at lines 3 and 4, it directs the framework to remap classes of the `Generic` package to classes with same name (except a prefix) of the `Structural` package. The rule at lines 6 to 17 implements a `remapValue` hook for `eStructuralFeatures` of a `SEClass` target object, when it is mapped from a `GEClass` source object. Lines 8 to 11 retrieve the default behavior of the hook, that is, the cloning of objects as a function named `transformer`. Then, the rule retrieves the `eStructuralFeatures` of the source object (line 11), filters those that are neither derived nor transient nor volatile (lines 12 and 13), applies the default behavior `transformer` to the retained objects (line 14), and put the resulting objects

```
1 public class G2S extends GenericRemapper {
2   public G2S() {
3     super(GenericPackage.eINSTANCE, "GE",
4           StructuralPackage.eINSTANCE, "SE");
5   }
6   @Rule public void eStructuralFeatures(
7         GEClass source, SEClass target) {
8     Function<EObject, EObject> transformer =
9       maybeTransform(GE_CLASS__ESTRUCTURAL_FEATURES,
10                     SE_CLASS__ESTRUCTURAL_FEATURES);
11      source.getEStructuralFeatures().stream()
12      .filter((s) -> !s.isDerived() && !s.isTransient()
13                     && !s.isVolatile())
14      .map(transformer)
15      .map(cast(SEStructuralFeature.class))
16      .forEach(target.getEStructuralFeatures()::add);
17  }
18 }
```

Fig. 13. Java code for step 3 *eliminate*.

in `eStructuralFeatures` of the target object (line 16). Other structural features in any other class are just copied by the framework. By not having any structural feature named `eOperations` in `SEClass`, operations are omitted in the target (meta)model, without any further programming.

At step ⑪, QVT-Operational seems a good choice as it avoids most of the notation burden. Disjunct mapping and collection operations like `iterate` remind pattern-matching and higher-order functions, like usually found in functional programming. For the purpose of the comparison, we have also manually translated the QVT-Operational transformation into strictly equivalent Java code.

Our implementation records mapping information between source objects and generated ones, including intermediate ones, using a generic one-to-one correspondence metamodel. In order to deal with the size of mapping information, especially in the context of EMF's XMI serialization, our implementation produces two records:

- Summarized mapping information record only indirect mapping from each source Ecore object to the corresponding Gallina objects, hence omitting intermediate steps.
- Full mapping information is split into fragments in order to keep the size of each fragment below a fixed threshold. This trick lowers memory consumption by EMF's XMI serialization code. Synthetic objects, which do not belong to any intermediate model, are saved alongside mapping information.

Most of the transformation steps are exogeneous transformations, with the only exception of step ⑩ that is endogeneous. By making this choice, the abstract languages for target metamodels are fitted to the needs. Doing so is required to for implicit removal of objects by our framework, such as the removal of operations at step ③ like

	Ecore files (downloaded)	Ecore files (after build)	Ecore files (validated)	Coq scripts (generated)
raw count	279	338	319	319
unique files	241	241	226	197

Table 1. Summary of the benchmark suite.

previously described. The whole transformation involves 11 different metametamodels. In the pipeline of the whole transformation, each two consecutive metametamodels must remain consistent for the elements that are not directly affected by the concerned step. Managing such 11 different metametamodels while maintaining their consistency appears to us to be a tedious and error-prone task during development, despite the small size of the metametamodels[3]. We would have appreciated lightweight mechanisms for automatic application of some editing commands to a group of several metametamodels. Using transformations engines, *e.g.*, based on QVT or ATL, for such operations appeared inconvenient: such transformations would have had to be generic enough to be applicable to several similar-yet-different metametamodels, while not offering any reuse opportunity because editing commands are one-shot.

The Java code for steps ① to ⑩ contains 1555 SLOC, and the underlying Java-based framework contains 320 SLOC according to `ohcount`. The QVT-Operational script for step ⑪ is made of 218 SLOC and its Java translation contains 655 SLOC. The Acceleo template for step ⑪ contains 41 SLOC. The code is available at https://bitbucket.org/jbuisson/ecore2coq.

9 Validation

To evaluate our transformation and our implementation, we build a benchmark suite from four third-party open-source projects, in addition to the metamodels involved in our transformation itself: EMF, Eclipse's OCL, Xtext extras, and Dresden OCL. In total, we gather 80 Eclipse projects from the repositories of these projects. Like summarized in Table 1, these projects contain 279 Ecore metamodel files, of which 241 are unique files. At the completion of Eclipse's builder task, some of these metamodels are duplicated in target directories, yielding to 338 Ecore metamodel files. In this suite, 319 files pass EMF's validator without any error nor warning, of which 226 are unique files. Our transformation produces 319 Coq scripts, of which 197 are unique files, hence illustrating that our transformation is not injective.

9.1 Metrics on Source Metamodels and Generated Scripts

Table 2 summarizes some metrics about few source metamodels and generated scripts. The biggest metamodel is the UML metamodel in its version coming from the Dresden OCL project: it contains 247 classes, of which 199 are concrete and 48 are abstract. It

[3] Each metametamodel contains approximately 20 classes, 50 references, 30 attributes, and 30 data types

	Ecore source				after step ① *link*				Gallina target		
	classes				classes						
File	concr.	abs.	type	enum.	concr.	abs.	type	enum.	ind.	cons.	def.
	20				20						
`Ecore`	15	5	33	0	15	5	33	0	20	61	41
	247				265						
`UML`	199	48	4	13	212	53	15	13	278	1964	23
					...						
	4113				11578						
Total	3421	692	943	210	8777	2801	4092	367	11945	43584	6675

Table 2. Metrics on source metamodels and generated scripts.

also contains 4 data types and 13 enumeration types. Not shown on Figure 2, the classes of the source metamodel contain 508 references and 110 attribute; increased to 554 references and 140 attributes after step ①. After step ①, additional classes are pulled in, increasing the size to 265 classes (212 abstract classes and 53 concrete classes), 15 data types and 13 enumeration types. Because one inductive type is generated for each class and for each enumeration type, 278 inductive types are generated, which altogether define 1964 constructors. The 23 generated type definitions come from the 15 data types after step ①, and 8 predefined types definitions, *i.e.*, types like `_URI` and `_Option` mentioned at Section 6.1.

In total, the source metamodels of our benchmark contains 3421 concrete classes and 692 abstract classes containing 5529 references and 2996 attributes, 943 data types and 210 enumeration types. After step ①, it contains 11578 classes (8777 concrete classes, 2801 abstract classes, 23743 references and 13589 attributes), 4092 data types and 367 enumeration types. Our transformation produces 11945 inductive types made of 43584 constructors, as well as 6675 type definitions.

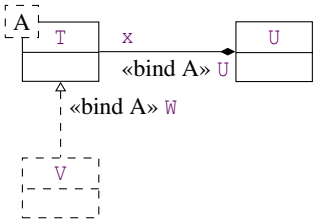
The metamodels of our benchmark involve multiple inheritance. At most, one class has 8 immediate super types, increased to 9 after step ①. The depth of inheritance is up to 11 levels.

9.2 Validity of the Generated Scripts

As a first validation, we ensure that the transformation produces correct Coq scripts. To do so, we invoke the Coq compiler on each of them. Of the 319 generated scripts, 315 pass successfully compilation. Manual inspection lets us ensure that the generated scripts correspond to the source Ecore metamodels.

The remaining 4 scripts raise Coq's *non strictly positive occurrence* error. To better understand this error, consider the anti-pattern and generated script of Figure 14. The generated type `U` recursively uses itself in the type of parameter *x*, as a parameter of generic type `T`. By doing so, and depending on the type of `T`'s constructors, one can make the logic inconsistent, but, in the case of Figure 14, this is not the case. Still Coq[4] restrict definitions of inductive types to conform to a syntactical criterion, named *strictly*

[4] Other proof assistants based on dependent types, such as Agda or Lean behave similarly.




```
Inductive T: Type → Type := | T_T: ∀ (A: Type), (T A).
Inductive U: Type := | U_U: ∀ (x: T U), U.

(* alternate definition of T *)
Inductive T (A: Type): Type := | T_T: (T A).
```

Fig. 14. Anti-pattern leading to Coq error.

positive recursive occurrences in order to ensure that any inconsistent definition is rejected. However, the syntactical criterion is a conservative one, hence ruling out some consistent definitions too. In Figure 14, the alternate definition of T, which differs only in Coq's internal treatment, allows to work-around the criterion by moving parameter *A* to the inductive type itself instead of the constructor. This alternate definition shows that, in this example, our transformation generates only consistent definitions.

Figure 14 is representative of the 4 failing scripts. Because the problematic parameters are type parameters of generic classes, one could think the error could be avoided by simply changing our transformation to put type parameters at the inductive type, rather than at constructors. But doing so would prevent a non-generic class, *e.g.*, `V` to inherit from, *e.g.*, `T<W>` (dashed in Figure 14, where `W` is yet another class), while such a construct is perfectly permitted by our transformation and yields no error in the generated Coq script.

The error could be worked-around by specializing generic classes, that is, by doing partial application of type parameters. We have not yet included this strategy in our transformation.

9.3 Execution Time of the Transformation

Using the same benchmark, we have measured the execution time spent in the transformation with several platforms. In this perspective, we instrument the transformation in order to measure the time spent in each step, including serialization of the result but excluding XMI deserialization of the input. To mitigate intrinsic variations of execution time, each step is preceded by an invocation of the garbage collector. We also do our best to make the transformation reproduce identical artifacts regardless the execution environment, sometimes at the cost of sub-optimal implementation, *e.g.*, to ensure identical order of elements even in unordered collections. In order to avoid effects of cache and JIT, every source metamodel is transformed several times before execution time is measured.

Table 3 gives the details of the execution platforms we use. For each platform, we enable G1 garbage collector and string deduplication in the Java virtual machine. In each run, we use a headless Eclipse product.

Measured execution time are given in Table 4. We first use platform D to compare Java and QVT-Operational implementations of step (11): we observe that the Java implementation is 30 times faster than the QVT-Operational implementation.

Platform	CPU	RAM	OS	JDK	Heap size
A	2x Xeon L5640	48GiB	Debian 9.4	Oracle 8_162 64-bits	16GiB
B	4x Xeon E5-2630 v4	128GiB	Debian 9.2	Oracle 8_162 64-bits	16GiB
C	4x Xeon E5-2630 v4	128GiB	Debian 9.2	Oracle 10+46 64-bits	16GiB
D	1x i7-4702HQ	16GiB	Windows 10 1709	Oracle 10+46 64-bits	8GiB

Table 3. Execution platforms used in experiments.

	A		B		C		D	
Step	Java 1	Java 2	Java 1	Java 2	Java 1	Java 2	Java	QVTo
① *link*	7.7	7.5	2.8	2.9	2.7	2.7	9.4	9.4
② *normalize*	5.8	5.7	2.1	2.1	1.9	2.0	6.9	6.6
③ *eliminate*	3.9	3.8	1.3	1.2	1.2	1.2	5.0	5.0
④ *eliminate*	4.8	4.6	1.8	1.8	1.6	1.7	5.6	5.4
⑤ *introduce*	5.8	5.6	2.2	2.2	2.1	2.1	6.4	6.1
⑥ *percolate*	14.8	15.4	6.2	6.0	5.7	5.9	10.7	10.5
⑦ *flatten*	495.4	509.4	202.1	196.6	187.3	197.9	207.6	202.9
⑧ *translate*	43.8	45.0	22.9	22.4	21.5	22.3	31.7	30.5
⑨ *schedule*	146.7	146.4	67.0	68.2	61.0	61.0	82.9	81.2
⑩ *rename*	65.5	65.9	34.3	34.0	33.5	33.5	61.6	60.9
⑪ *generate*	39.2	39.1	19.5	19.5	19.6	19.1	42.1	1332.9
⑫ *generate*	94.1	95.3	52.6	50.2	45.1	45.1	49.6	49.0
save mapping summary	193.1	199.2	119.3	116.5	114.5	115.5	123.8	123.3
Total (transformation)	1120.5	1142.9	534.1	523.6	497.7	510.0	643.2	1923.6
save synthetic objects	579.5	579.1	358.6	353.8	357.2	370.3	437.5	433.1
save full mapping	2731.9	2737.4	1658.4	1727.5	1748.7	1709.9	1817.1	1792.9

Table 4. Measured execution time (seconds) of each individual step.

After this first observation, we measure execution time only for the Java implementation using the other platforms, hence showing that, even with real-world metamodels, our transformation is compatible with use in interactive IDE. By comparing two runs on each platform, we evaluate variability of the execution time, here up to 2.6%, despite efforts to mitigate. Because the transformation is intended to be used in interactive environments, we do not feel necessary to have more accurate measures.

To assess the relevance of summarized mapping information, the two last lines of Table 4 give the time spent when full mapping information is saved, including synthetic objects and intermediate objects generated at each step. We observe the time spent in these tasks is large in comparison to the time spent in the transformation. To better explain, in the case of Dresden OCL's UML metamodel, the biggest one of our suite, 59793 synthetic objects are created and 7414201 correspondences are recorded, while summarized mapping contains only 6287 correspondences. For the complete benchmark suite, 20GiB are generated for synthetic objects and mapping, to be compared with 6.9MiB input Ecore files, 2.5GiB generated models and 359MiB summarized mapping. This observation confirms that saving full mapping information should be avoided if unnecessary.

9.4 Usability in the Context of Proof-Carrying Code

To ensure that the generated scripts are actually usable in the context of proof-carrying code [18], we fully applied the approach depicted in Figure 1 in the implementation of tools supporting SosADL [1]. This paper focuses on automating only the transformation of the Ecore metamodel towards Gallina inductive types. Hence the others tasks either involve preexisting tools such as Xtext [6] or are done by hand. Noticeably, the transformation of SosADL architectures into Gallina terms has been manually written. Then we manually instrument SosADL's type checker in order to produce proofs that witness that the architecture under consideration is actually well-typed. Last we use Coq to check the correctness of all the generated artifacts.

This experiment witnesses that the types generated by the transformation described in this paper conform to the requirements of a proof-carrying code infrastructure.

10 Conclusion

In this paper, we describe a transformation from Ecore metamodels to inductive types. This transformation allows to set up a model-driven language engineering chain, *e.g.*, involving Xtext and, at the same time, to specify the language using a proof assistant, such as Coq, and then prove properties of this specification. In comparison to previous work [9,13], our transformation has fewer constraints on the source Ecore metamodel and ensures stronger typing in the generated inductive types, but it is not injective.

To validate our proposal, we implement the transformation using QVT-Operational, Acceleo, and EMF-and-Java. We fetch of 226 Ecore metamodels from open source projects, in order to gather both synthetic and real-world metamodels for the purpose of benchmarking. For all of them, except 4 metamodels, our transformation produces valid Coq scripts. The 4 erroneous cases correspond to a specific pattern that infringes a syntactic criterion that prevents Coq from accepting potentially inconsistent type definitions. Still, this criterion is a conservative approach, and we analyze that, in our case, none of the generated type definitions may introduce any contradiction in the underlying logic. We propose to further study this point, investigating how partial application of type parameters may work-around Coq's restriction.

Our experience shows that, regardless the execution platform, our transformation scales to real-world metamodels.

In the near future, we will continue automatic generation of the proof-carrying code infrastructure from an Ecore metamodel. Our next step will be to produce a second transformation, that will generate the transformation from instance models to terms, such that the terms have the types generated by the transformation described in this paper.

References

1. The SoS Architect Studio: Toolchain for the formal architecture description and analysis of software-intensive systems-of-systems with SosADL

2. K overview and SIMPLE case study. Electronic Notes in Theoretical Computer Science **304**, 3–56 (2014). DOI 10.1016/j.entcs.2014.05.002. Proceedings of the Second International Workshop on the K Framework and its Applications (K 2011).
3. On the verification of UML/OCL class diagrams using constraint programming. Journal of Systems and Software **93**, 1–23 (2014). DOI 10.1016/j.jss.2014.03.023
4. Barbier, F., Cariou, E.: Inductive UML. In: Proceedings of the 2nd International Conference on Model and Data Engineering, MEDI'12, pp. 153–161. Poitiers, France (2012). DOI 10.1007/978-3-642-33609-6_15
5. Bertot, Y., Castran, P.: Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions, 1st edn. Springer Publishing Company, Incorporated (2010)
6. Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend. Packt Publishing (2013)
7. Borras, P., Clement, D., Despeyroux, T., Incerpi, J., Kahn, G., Lang, B., Pascual, V.: Centaur: The system. In: Proceedings of the Third ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SDE 3, pp. 14–24. Boston, Massachusetts, USA (1988)
8. Buisson, J., Rehab, S.: Automatic transformation from Ecore metamodels towards Gallina inductive types. In: Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD,, pp. 488–495. INSTICC, SciTePress (2018). DOI 10.5220/0006608604880495
9. Djeddai, S., Strecker, M., Mezghiche, M.: Integrating a formal development for DSLs into meta-modeling. In: Proceedings of the 2nd International Conference on Model and Data Engineering, MEDI'12, pp. 55–66. Poitiers, France (2012)
10. Gerber, A., Raymond, K.: MOF to EMF: There and back again. In: Proceedings of the 2003 OOPSLA Workshop on Eclipse Technology eXchange, eclipse '03, pp. 60–64. Anaheim, California (2003)
11. Kats, L.C., Visser, E.: The Spoofax language workbench: Rules for declarative specification of languages and IDEs. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '10, pp. 444–463 (2010)
12. Klint, P.: A meta-environment for generating programming environments. ACM Trans. Softw. Eng. Methodol. **2**(2), 176–201 (1993)
13. Klint, P., van der Storm, T.: Model Transformation with Immutable Data, pp. 19–35. Springer International Publishing, Cham (2016). DOI 10.1007/978-3-319-42064-6_2
14. Klint, P., van der Storm, T., Vinju, J.: EASY meta-programming with Rascal. In: Proceedings of the 3rd International Summer School Conference on Generative and Transformational Techniques in Software Engineering III, GTTSE'09, pp. 222–289 (2011)
15. Lano, K., Clark, D., Androutsopoulos, K.: UML to B: Formal Verification of Object-Oriented Models, pp. 187–206. Springer, Berlin, Heidelberg (2004). DOI 10.1007/978-3-540-24756-2_11
16. Meyer, E., Souquières, J.: A systematic approach to transform OMT diagrams to a B specification. In: Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems-Volume I - Volume I, FM '99, pp. 875–895 (1999). DOI 10.1007/3-540-48119-2_48
17. Mulligan, D.P., Owens, S., Gray, K.E., Ridge, T., Sewell, P.: Lem: Reusable engineering of real-world semantics. In: Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming, ICFP '14, pp. 175–188. Gothenburg, Sweden (2014)
18. Necula, G.C.: Proof-carrying code. In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '97, pp. 106–119. Paris, France (1997)

19. Nipkow, T., Wenzel, M., Paulson, L.C.: Isabelle/HOL: A Proof Assistant for Higher-order Logic. Springer-Verlag, Berlin, Heidelberg (2002)
20. OMG: OMG Meta Object Facility (MOF) Core Specification (2016). URL http://www.omg.org/spec/MOF/2.5.1/
21. Sewell, P., Nardelli, F.z., Owens, S., Peskine, G., Ridge, T., Sarkar, S., StrniŠa, R.: Ott: Effective tool support for the working semanticist. J. Funct. Program. **20**(1), 71–122 (2010)
22. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0, 2nd edn. Addison-Wesley Professional (2009)
23. Tisi, M., Martínez, S., Jouault, F., Cabot, J.: Refining models with rule-based model transformations. Research Report RR-7582, INRIA (2011). URL https://hal.inria.fr/inria-00580033
24. Voelter, M.: Language and IDE Modularization and Composition with MPS, pp. 383–430. Springer, Berlin, Heidelberg (2013). DOI 10.1007/978-3-642-35992-7_11