Rules for Type-checking of Parametric Polymorphism in EMF Generics

Miguel Garcia

Institute for Software Systems (STS)
Hamburg University of Technology (TUHH), 21073 Hamburg, Germany
miguel.garcia@tuhh.de

Abstract: The Eclipse infrastructure for modeling is based on EMF, an implementation of Essential MOF, the OMG standard for metamodeling. A recent addition to EMF has been parametric polymorphism (also known as "parameterized types" or "generics") thus achieving the same benefits realized in other generic type systems (most notably, Java 5). To our knowledge, this is the first formal account in the literature of the resulting type system, including the well-formedness rules (WFRs) for type expressions. We offer an initial formalization of the type system of EMF Generics by formulating those WFRs in OCL (Object Constraint Language). The insights thus gained are applicable to solve the type-checking problem for custom DSLs (Domain Specific Languages).

1 Introduction

The EMF infrastructure allows the programmatic manipulation of models as first-class citizens by making available (among others) reflection and persistence services that significantly increase the productivity of the development of modeling tools. A cornerstone of such infrastructure is the language in which models are expressed (Ecore) whose type system adds generics to the type system of Essential MOF [Obj06]. Models expressed in Ecore can be seen in turn as a shorthand notation for types in Java, thanks to the code generation capabilities delivered with EMF. It is these generated classes that are used as a software component in tools for DSLs (Domain Specific Languages), with the Ecore-based model specifying the abstract syntax for the DSL in question [Gar06]. Ecore can equally well be used to capture the data model of a general-purpose application, although we focus on the modeling-tool scenario throughout this paper.

The modeling abstractions supported by Ecore language constructs can be conceptualized as a subset of those available for UML class models. With this limitation no significant loss in expressivity is incurred, as every datamodel expressed in UML can be reformulated as a corresponding Ecore + OCL model. Several advantages result from focusing on a well defined set of constructs. For example, an implementation supporting Object-Relational Mapping (ORM) of Ecore-based models must deal with considerably fewer special cases than its UML counterpart.

The structure of this paper is as follows. Sec. 2 reviews definitions around type systems, serving as background for the characterization of EMF Generics in Sec. 3, including the OCL formulation of well-formedness for type expressions. Subtyping between parameterized types is formalized, after discussing conformance of a parameterized type to its declaration. Related work, e.g. tooling support for modeling with generics, is covered in Sec. 4. Conclusions and plans for future work are discussed in Sec. 5. Knowledge is assumed from the reader about metamodeling and Java 5 Generics, while experience in OCL is helpful but not required.

2 Type systems of programming and modeling languages

The grammar of a programming or modeling language specifies the set of syntactically legal programs. Not all of them are useful: those (programs, models) on which the translator would fail are discarded by the well-formedness rules (WFRs) for sentences in the language. For example, requiring all usages of an identifier to refer to a declaration that is in scope is a common WFR. Typing rules add another hurdle that well-formed programs should overcome: they allow determining (preferably at compile-time) the most specific type of each expression, with the purpose of rejecting those programs that are type unsafe, i.e., those which may cause at runtime the assignment of a value of type T_1 to a location declared to hold values of type T_2, with T_1 not a subtype of T_2. Type safety alone does not rule out all unwanted runtime behaviors: correctly typed programs may crash, never terminate, or produce incorrect results. The decision procedures for analyzing properties of interest beyond type-safety are the realm of Hoare-style program verification [Gor89] or model-checking tools [Lam06]. In contrast to these specialized techniques, type systems have a successful track record in terms of cost/benefit (specification effort vs. variety of unsafe situations detected), thus explaining all the effort that goes into their engineering. The designers of a type system must balance natural tensions among expressiveness, performance of type inference, and usability of the resulting type system.

More in detail, specifying a type system involves:

1. Making explicit the rules by which new types can be defined. Together with the set of built-in types, types so constructed constitute the *universe of types* for a given program or model.

2. Building upon this vocabulary of valid types, the link to the grammar of the language is established in the form of *typing rules*, i.e., a procedure to assign (sometimes infer) a unique type for each well-formed expression in the language. This inference is not performed in isolation but taking into account an *environment* of visible $\langle identifier, type \rangle$ pairs.

3. Finally, the subtyping relationship between types allows determining, in conjunction with the type annotations from the previous item, whether the program is type safe. This algorithm is embodied in the *type checker*

The mechanism of choice to define type construction expressions is usually an EBNF grammar with additional WFRs, generating a set of valid types instead of valid sentences. The need for WFRs for type expressions can be seen for example in the context of *generic type declarations*, i.e., those owning one or more type parameters which may impose constraints for type arguments to fulfill. Not every syntactically-correct *parameterized type* (listing *type substitutions*) conforms to its type declaration. Types are generally compile-time entities, but EMF always reifies them at runtime for reflection. Java 5 supports reflection of typing information in the form of `java.lang.reflect` classes, with much improved support available in Java 6 (`javax.lang.model.util.Types`).

Similar to Java before generics, a classifier declaration in Ecore before 2.3 did not own any type parameters and thus was a constructor for just one type. Instead, a classifier with type parameters defines a *set of types*, one for each conformant substitution by a type argument. For example, a list that keeps its items automatically sorted may be declared as

```
class OrderedList<T extends Comparable<? super T>> {...}
```

making clear that `T` admits any type argument conforming to `Comparable<? super T>`. Interface `Comparable` allows comparing two values of type `T` or supertypes thereof. For example, if an ordered list is to contain `String` items a comparator for `Object` will also do, as strings are objects. Bounded generics allow writing generic algorithms which minimally depend on the type of the input, while preserving static type-safety. The idea of factorizing object capabilities into fine-granular types was first introduced by ML [Gil97].

Angle brackets are used for two different purposes in Java 5. In the example above, the outermost pair of angle brackets encloses a *type parameters section*, while the innermost pair encloses a *type arguments section*. A type parameters section occurs in the context of type construction, while no new type is introduced by type arguments (they act like queries returning types already defined). In the example, the fragment `? super T` denotes a set of types, each of them lower bounded by `T` (*lower bounded by* being generics-speak for *supertype of*). Subtyping is formalized in Sec. 3.3. The `?` is not actually a type argument but a wildcard standing for any of several possible type arguments. A wildcard can only be used in places where a type argument is expected, however it should not be considered to be the name of a specific type. For instance, each occurrence of `?` in `Pair<?,?>` in general stands for different types, and `?` is *not* a subtype of `?`. By convention, the unqualified wildcard `?` is a shorthand for `? extends Object`.

3 Well-formed Type Expressions in EMF Generics

Two main constructs are subject to well-formedness checking: (a.1) the declaration of a generic type, and (a.2) a parameterized type, i.e., an invocation of (a.1) with type arguments. The OCL WFR for (a.1) is given in Sec. 3.1, as a prerequisite for answering:

b.1. whether a given parameterized type is a valid invocation of its declaration (Sec. 3.2)

b.2. subtyping relationship between two parameterized types (Sec. 3.3)

The precise formulation of these queries over ASTs is not straightforward given the rich structure of references that (a.1) and (a.2) may exhibit. For example, determining (b.1) must take into account two different scopes for type parameters (that of the declaration and that of the invocation) where moreover wildcards may occur (except in bounds of type parameters in a.1). For the same reason, the case-analysis of transformation algorithms operating on ASTs of type expressions is intricate.

Figure 1 shows the new classes (`ETypeParameter` and `EGenericType`) added to the metamodel of Ecore to support genericity. Only legal instantiations of this metamodel will result in legal Java types. The reader is invited to compare the readability of the OCL formulation vs. the current realization in EMF, commented procedural code in `org.eclipse.emf.ecore.util.EcoreValidator` ("specification by reference implementation"). Moreover, OCL invariants can be compiled into Java, allowing the automatic detection of violations of WFRs during AST tree-building or transformation, before further processing takes place. The WFRs for the non-generics fragment of Ecore are covered elsewhere [BSM$^+$03, Obj06].

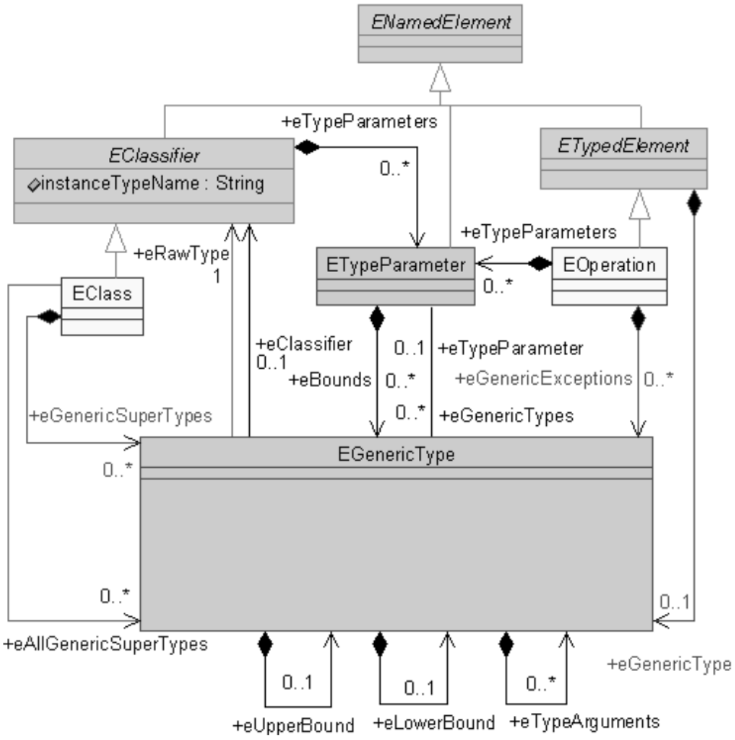

Figure 1: Fragment of the Ecore metamodel dealing with Generics

3.1 Declaration of a Generic Type

Informally, well-formed ASTs of (a.1) consist of an `EClassifier` with a non-empty list of `ETypeParameter`, which from then on are in scope for the whole type declaration (e.g. `class C <T extends C<T>>` is legal, §8.1.2 in [GJSB05]). A type parameter owns a possibly empty list of upper bounds, later used to answer (b.2). Each such bound is represented in the AST as an `EGenericType` which may represent:

- a reference to a type variable (in scope for a.1), acting as a terminal node, as no type arguments can be specified
- a reference to a non-generic type
- a reference to a generic type, either with or without type arguments (in the latter case a so called *raw* type reference).

Listing 1 contains the OCL formulation of the sketched WFRs.

Listing 1: WFRs for generic type declarations

```
context EClassifier
inv consistentTypeParameters :
  allDifferent( eTypeParameters.name ) and
  eTypeParameters->forAll( tp | tp.isConsistent(eTypeParameters) )

context ETypeParameter::isConsistent( tpsInScope :
  Collection(ETypeParameter)) : Boolean
def: self.name <> '' and ( self.eBounds->isEmpty() or
     self.eBounds->forAll( tr |
        tr.isConsistentTypeReference(tpsInScope) ) )

context EGenericType::isConsistentTypeReference(
  tpsInScope : Collection(ETypeParameter)) : Boolean
def: not isWildcard() and (
( self.isReferenceToTypeParameter()
  and tpsInScope->includes(self.eTypeParameter) )
xor
(self.isReferenceToClassifier() and
  -- self.eClassifier is the declaration of the generic type
 self.eClassifier.isValidTypeInvocation(self.eTypeArguments) ) )

context EGenericType::isReferenceToTypeParameter() : Boolean
def: eClassifier->isEmpty() and
     not eTypeParameter->isEmpty() and eTypeArguments->isEmpty()

context EGenericType::isReferenceToClassifier() : Boolean
def: not eClassifier->isEmpty() and eTypeParameter->isEmpty()
```

3.2 Type invocation

Again informally, a parameterized type (a.2) consists syntactically of a name reference to a generic type (a.1), followed by one or more type arguments. At the AST level, all these constructs are instances of `EGenericType`, with conventions on their connectivity used to determine the role played by the instance (conventions fixed when the abstract syntax was chosen). Details are given below as individual WFRs are reviewed. Syntactically valid type arguments can be any of:

- references to type variables in scope for the type invocation. This scope is introduced by a generic type or operation declaration in which the type invocation occurs, and is not to be confused with the scope of type variables for the referred generic type declaration.
- references to types (raw type reference, parameterized type reference, or reference to non-generic type)
- wildcards (`?`, `? extends` *`oneUpperBound`*, `? super` *`oneLowerBound`*). Each of these bounds, just like bounds in a generic type declaration, cannot be a wildcard itself. Instead, it can assume any of the forms described in the two previous items.

The start situation when answering (b.1) are ASTs for a parameterized type $P_1 = G < T_1, ..T_n >$ and its declaration $D_1 = G < A_1, ..A_n >$. P_1 and D_1 are not directly comparable as usages of type variables belong to disjoint scopes and as P_1 may contain wildcards. A process termed *capture conversion* ([GJSB05], §5.1.10) rewrites those wildcards into appropriately bounded fresh type variables (whose lifetime is limited to answering b.1). The resulting type invocation has the form $P_2 = G < X_1, ..X_n >$. The next rewriting takes place on the declaration D_1 by substituting all occurrences of type variables declared in D with the type arguments from P_2. Each $A_i, i = 1..n$ is rewritten to $A_i[V(A_j) \leftarrow X_j, j = 1..n]$, where $V(A_j)$ stands for the type variable introduced by A_j. The resulting working expression D_2 is not a declaration anymore but a parameterized type (a.2) sharing the same scope of type variables as P_2. The final step involves iterating over each actual type argument tp_i in P_2, checking if it is a subtype of each upper bound of the type argument at the ith position in D_2. In case any of these comparands refers to a type variable, *type argument containment* (explained in the next subsection) is used.

Listing 2: WFRs for type invocations

```
context EClassifier::isValidTypeInvocation( -- see 4.5 in JLS3
  -- typeArgs contains the type arguments of the type invocation
  typeArgs : Sequence(EGenericType) ) : Boolean
def: if not self.isGenericTypeDeclaration()
     then typeArgs->isEmpty()
     else typeArgs->isEmpty() -- raw type
          or isValidTypeInvocationAfterCaptureConversion(
                 captureConversion(typeArgs) )
     endif
```

```
context EClassifier::isValidTypeInvocationAfterCaptureConversion(
  typeArgs : Sequence(EGenericType) ) : Boolean
pre: typeArgs->isEmpty()
     or typeArgs->forAll( ta | not ta.isWildcard() )

context EClassifier::isValidTypeInvocationAfterCaptureConversion(
  typeArgs : Sequence(EGenericType) ) : Boolean
def: -- this operation has an OCL-specified precondition, see above
  typeArgs->isEmpty()
  or Sequence( 1..typeArgs->size() )->forAll(index |
       eTypeParameters->at(index).isValidTypeSubstitution(
         typeArgs->at(index), typeArgs ) )

context ETypeParameter::isValidTypeSubstitution( -- 4.10.2 in JLS3
  -- ccta is a capture-converted type argument, i.e. not a wildcard
  ccta : EGenericType,
  typeArgsForAllTypeParams : Sequence(EGenericType) ) : Boolean
def: self.eBounds->forAll(si |
        si.isSuperTypeOf(ccta,typeArgsForAllTypeParams) )
```

3.3 Subtyping between two parameterized types

$Subtype$ and $Supertype$ are binary relations on types. They are partial orders (i.e., reflexive, antisymmetric, and transitive) and transpose of each other. It is sufficient to define $directSubtype$ (or $directSupertype$) to have $Subtype$ and $Supertype$ univocally determined. Direct subtype, symbolized $<_1$, consists of:

- an enumeration of pairs of predefined types (i.e. for the $BuiltInType \times BuiltInType$ subset of $Type \times Type$)
- a partition of the remaining cases into categories with a membership condition for each. For example, in the type system of Java 5, the partition with pairs of the form $\langle NullType, C\rangle$ (where C is a class or interface type) has the constant membership condition $True$. As a consequence, `null`, the only value conforming to the type $NullType$, can always be assigned to a location declared to hold instances of C, for any declared type C.

Besides answering whether subtyping holds between two types, it is sometimes possible to conclude whether two type expressions T_1,T_2 (involving type variables and wildcards) exhibit subtyping, i.e., whether the set of types denoted by T_1 is necessarily a subset of that denoted by T_2. This procedure, *type argument containment* (§4.5.1.1 in [GJSB05]) is depicted visually in Figure 2 and formalized for the Ecore metamodel in terms of OCL as shown in Listing 3.

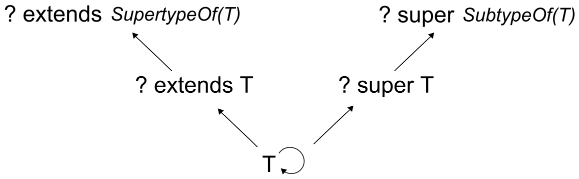

Figure 2: Graphical display of the rules for Type Argument Containment. Arrows point at the container argument

Listing 3: Type Argument Containment

```
context EGenericType::contains(TA : EGenericType) : Boolean
def: (self = TA) or
     if self.isUpperBoundedWildcard()
     then -- left branch in Figure 2
          if TA.isUpperBoundedWildcard()
          then self.eUpperBound.isSuperTypeOf(TA.eUpperBound)
          else self.eUpperBound = TA
          endif
     else if self.isLowerBoundedWildcard()
          then -- right branch in Figure 2
             if TA.isUpperBoundedWildcard()
             then self.eUpperBound.isSuperTypeOf(TA.eUpperBound)
             else self.eUpperBound = TA
             endif
          else false
     endif
```

A type argument is either a wildcard, a raw type invocation, a parameterized type invocation, or a reference to a type variable. Therefore, type argument containment alone does not provide an answer in all cases where two arbitrary type arguments are compared for subtyping. An important special case is the comparison of two parameterized types without wildcards or type variables: it succeeds only when they refer to the same generic type and their arguments match exactly: none of `List<String>`, `List<Object>` is subtype of the other, for otherwise type-unsafe updates would be allowed. The rules in Figure 2 imply that upper-bounded wildcards exhibit covariant subtyping with respect to their bounds. In contrast, lower-bounded wildcards give rise to contravariant subtyping.

4 Related Work

Emfatic [Dal05] is an Eclipse-based IDE that greatly simplifies the creation of Ecore-based models thanks to a well-designed textual notation. We have extended Emfatic to handle generics, allowing in particular the translation from `.emf` into `.ecore`. This work is part of a larger project (supported by a 2006 Eclipse Innovation Grant) involving extensions to the Eclipse Modeling infrastructure. Up-to-date progress reports are available on this author's homepage, `http://www.sts.tu-harburg.de/~mi.garcia/`

Although Ecore so far does not include executable statements, this situation will change as ideas developed originally for Executable UML [RFW+04] find their way into Ecore extensions. Generics-aware type checking is relevant for the Eclipse MDT implementation of OCL, with UML2 support available from the 1.1 release (UML2 brings its own notion of polymorphic types). A discussion of typing for Generics OCL can be found in [Kya05].

Bruce explains in [Bru02] the formal machinery required for analyzing the type system of OO languages, including a discussion of the design decisions made for several languages currently in use. Determining whether the rules of a proposed type system (defined as described in Sec. 2) actually reject all type-unsafe programs is a topic on its own, with researchers increasingly relying on mechanized proofs, for example for Java [ON99].

Before generics, EMF developers would routinely reuse the Java code generated by EMF, adding potentially unsafe downcasts. A refactoring to automatically parameterize Java classes [KETF07] can be used to remove the need for such downcasts. The APIs used in its Eclipse implementation are explained in a tutorial by Fuhrer [Fuh05]. Another discussion of type inference in Java 5 can be found in [PB06].

5 Conclusions and Future Work

Type systems are an often neglected area in DSL design. This state of affairs can be traced back to the perception that the specification and implementation of type-checking is too time consuming, given its interdependence with all the language constructs of the DSL in question. Additionally, most DSLs are translated into an statically type-safe language which provides a "safety net" to catch type unsafe programs. We have motivated the need for declarative and machine-processable encodings of type-checking rules by showing productivity and quality advantages in comparison to a manual implementation for an industrially-relevant DSL, Essential MOF extended with Generics, which moreover has a rich type system. The resulting OCL expressions can be used as test oracles in conjunction with inputs produced by model-based test generation [CGN+05], and in the implementation of Design-by-Contract [MMS98]. With it, runtime checks during the execution of transformations on ASTs are instrumented automatically, reporting whenever a WFR is broken. As previously isolated transformations are aggregated into *model compilers*, the ability to detect a contract violation before further processing takes place is essential.

Future work building upon the formulation of type-checking rules in terms of OCL at the DSL metamodel level involves Certified Model Transformations. This certification comprises the design-time (symbolic) analysis of behavior of a transformation algorithm, for all possible executions that satisfy stated preconditions, to ensure termination and the establishment of stated postconditions. This capability is necessary if model compilers are to be considered on par with their 3GL counterparts.

References

[Bru02] Bruce, K. B. *Foundations of Object-Oriented Languages: Types and Semantics*. MIT Press, Cambridge, MA, USA, 2002.

[BSM$^+$03] Budinsky, F, Steinberg, D, Merks, E, Ellersick, R, and Grose, T. J. *Eclipse Modeling Framework*. Addison-Wesley Professional, Boston, MA, USA, 2003.

[CGN$^+$05] Campbell, C, Grieskamp, W, Nachmanson, L, Schulte, W, Tillmann, N, and Veanes, M. Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer. Technical Report MSR-TR-2005-59, Microsoft Research, 2005.

[Dal05] Daly, C. Emfatic Language for EMF Development, http://www.alphaworks.ibm.com/tech/emfatic, 2005.

[Fuh05] Fuhrer, R. Dagsthul Seminar Types for Tools, Static Analysis for Java in Eclipse, http://kathrin.dagstuhl.de/files/Materials/05/05251/05251.FuhrerRobert1.Slides.ppt, 2005.

[Gar06] Garcia, M. Formalizing the well-formedness rules of EJB3QL in UML + OCL. In Kühne, T, editor, *Reports and Revised Selected Papers, Workshops and Symposia at MoDELS 2006, Genoa, Italy*, LNCS 4364, pages 66–75. Springer-Verlag, 2006.

[Gil97] Gilmore, S. Programming in Standard ML '97: A Tutorial Introduction. Technical Report ECS-LFCS-97-364, Laboratory for Foundations of Computer Science, Department of Computer Science, The University of Edinburgh, 1997.

[GJSB05] Gosling, J, Joy, B, Steele, G, and Bracha, G. *Java(TM) Language Specification, The (3rd Edition) (Java Series)*. Addison-Wesley Professional, July 2005.

[Gor89] Gordon, M. J. C. Mechanizing programming logics in Higher Order Logic. In *Current trends in hardware verification and automated theorem proving*, pages 387–439, New York, NY, USA, 1989. Springer-Verlag New York, Inc.

[KETF07] Kieżun, A, Ernst, M. D, Tip, F, and Fuhrer, R. M. Refactoring for parameterizing Java classes. In *ICSE'07, Proceedings of the 29th International Conference on Software Engineering*, Minneapolis, MN, USA, May 23–25, 2007.

[Kya05] Kyas, M. An extended type-system for OCL supporting templates and transformations. In *M. Steffen and Gianluigi Zavattaro (Eds), Formal Methods for Open Object-Based Distributed Systems (FMOODS 2005), Lecture Notes in Computer Science, number 3535*, pages 83–98. Springer-Verlag, 2005.

[Lam06] Lamport, L. The +CAL Algorithm Language. In *NCA '06: Proceedings of the Fifth IEEE International Symposium on Network Computing and Applications*, page 5, Washington, DC, USA, 2006. IEEE Computer Society.

[MMS98] Meyer, B, Mingins, C, and Schmidt, H. Providing Trusted Components to the Industry. *Computer*, 31(5):104–105, 1998.

[Obj06] Object Management Group. Meta Object Facility (MOF) Core Specification, formal/06-01-01, http://www.omg.org/docs/formal/06-01-01.pdf, Jan 2006.

[ON99] Oheimb, D. v and Nipkow, T. Machine-checking the Java Specification: Proving Type-Safety. In Alves-Foss, J, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*, pages 119–156. Springer, 1999. http://isabelle.in.tum.de/Bali/papers/Springer98.html.

[PB06] Plümicke, M and Bäuerle, J. Typeless programming in Java 5.0. In *PPPJ '06: Proceedings of the 4th international symposium on Principles and practice of programming in Java*, pages 175–181, New York, NY, USA, 2006. ACM Press.

[RFW$^+$04] Raistrick, C, Francis, P, Wright, J, Carter, C, and Wilkie, I. *Model Driven Architecture with Executable UML*. Cambridge University Press, Cambridge, UK, 2004.