On Universal Code Model Description

Dmitri Boulytchev

September 15, 2023

Abstract

We discuss a problem of specifying a form of program representation — *code model* — which would provide a convenient, consistent and scalable substrate for implementing various software development automation tools for navigation, refactoring, transformation, etc.

1 Motivation

No production development process nowdays can be managed without extensive use of software development automation tools which assist developers in various ways by providing navigation through the code base, performing refactorings and other program transformations, support testing, continuous integration, collaborative development, etc. All these tools require a certain intermediate program representation — *code model* — to be built and maintained in sync with the source code. However, different tools require different aspects of the code base to be reflected by a code model. Thus, a problem of universality arises — it would be great to have a single code model to serve all the development automation tools. Moreover, different programming languages often share similar features (for example, a relation between definitions of program's entities and their usages), so it would be desirable if a code model could also contain reusable fragments for different programming languages. Finally, an importand requirement for code model is *scalability*. Industrial-tier software projects can incorporate millions and even tens of millions lines of codes. The trade-off between the speed of code model extraction and its completeness becomed an issue — the more complete code model is the more time it takes to build from the sources. Sometimes it is profitable to start working on incomplete code model which enables only a part of software automation tools functionality while continue completing it in the background or building some part of the code model on-demand. Similarly, different concrete representations (in-memory structures, caches, indices, memory-mapped files, even databases) can be desirable in various settings. While implementing each concrete representation as a rule presents just a technical problem, repeating this work anew for various languages and varioue concrete representations is tedious and error-prone.

In this report we present a certain approach for describing code models. This description introduces a certain meta-model based on which all concrete

representations can be developed once and for all. Moreover, the meta-model would make it possible to implement the transitions between different concrete representations thus facilitating the scalability. Finally we argue that an essential part of meta-models can be reused by code models for different concrete languages.

2 Code Model Description by Examples

The general idea is to consider a concrete code model as a set of named abstract entities and relations between them, possibly equipped with some attributes. Both entities and relations are *typed*, and the set of types is equipped with subtyping relation making it a poset. We assume that both entities and relations can have more than one type (i.e. can belong to more than one hierarchy of properties). We consider types as fully abstract notions with no *ad hoc* properties and no hidden semantics.

The presence of types facilitates the reuse of code models. Indeed, many languages share the same features — control constructs, functions, procedures, object-oriented constructs, types, etc., — but not in exactly the same form. Thus, it is hard (or even impossible) to develop one universal code model for all languages. However the presence of types makes it possible to introduce abstraction layers in code models; these abstraction layers can be reused by the code models for different languages.

To specify code model we propose to use a certain specification language. Besides entities and relations, which constitute the *structural* part of the code model, the language provides means to specify *queries* which constitute its *procedural* part.

We consider a number of example specification in this (still virtual) language.

The first example is a simple entity type specification:

```
entity Named (
  name : String
)
```

Here an entity type `Named` is introduced; each entity of type `Name` hold an attribute `name` of type `String`. We stipulate that the atributes of entities (and relations) can only be of primitive types (integers, strings, enums, etc.), but not composite values like lists, arrays, structures, etc.

Based on this specification, the derived code model would consist of a collection of (unrelated) entity instances of type `Named`. Note, different instances can still hold the same values of attribute `name`, and yet be distinct.

The next example is similar:

```
entity Scoped (
  scope : Static | NonStatic
)
```

```
entity Visible (
  visibility: Public | Private | Protected
)
```

Here we introduced another two types of entities, each hold one attribute of specified enum type. Given these declarations we can form the following query:

```
modelSource | select X with X.name       = "main" &
                            X.visibility = Public &
                            X.scope      = Static
```

Here `modelSource` is the data source for a concrete code model instance, and `select` construct tells us that we extract only those entities which are `Named`, `Scoped` and `Visible` at the same time, and their corresponding attributes are equal to the values specified in the query. Note, the decription itself does not define such entities; but a concrete code model derived from it might.

In the next example we, for the first time, introduce a relation between two entities:

```
entity Type;
entity Typed;

typeOf : Typed*, Type;
```

Here we defined two entity types — `Type` and `Typed`, both with no attributes, and a *many-to-one* relation `typeOf`. With these definitions we can fetch all `Typed` and `Named` entities from a model which types are at the same time are `Named` and which names are equal to "`int`":

```
modelSource | select X with X.name = "x" &
                            (typeOf X).name = "int"
```

And, again, currently we have no entities with these properties in the model; however such an entities can appear in derived models.

Similarly, we can introduce a generic notion for declarations, usages and relations between them:

```
entity Declaration : Named;
entity Symbol : Named;

declarationOf : Symbol*, Declaration;
```

Note, here, for the first time, we declared one entity type to be a *subtype* of another: both `Declaration` and `Symbol` are separate subtypes of `Named`.

Now we can describe a derived code model for a subset of Java [1].

First we cover the object-oriented layer of the lanaguge:

```
entity Class  : Declaration, Scoped, Visible;
entity Method : Declaration, Scoped, Visible;
entity Field  : Declaration, Scoped, Visible, Typed;

methodOf     : Method*, Class;
fieldOf      : Field*, Class;
superClassOf : Class, Class?;
```

These definitions introduce the entities for classes, fields, and methods together with the relations connecting methods and fields with enclosing class and classes with their (optional) superclasses.

The next layer is control flow constructs of methods' bodies:

```
entity Statement;

entity Block       : Statement;
entity Assn        : Statement;
entity Conditional : Statement;
entity Loop        : Conditional;
entity If          : Conditional;
entity Seq         : Statement;
entity Return      : Statement;

entity Declaration : Named, Typed;

bodyOf        : Method, Block;
declarationOf : Declaration*, Block;
statementOf   : Statement, Block;
```

Here we introduced the entity types for a number of control flow constructs each of which is a subtype of a generic `Statement` entity type. The relations connect methods with their bodies, blocks with the list of declarations and blocks with enclosed statement.

The next section decyphers the structure of all statement types:

```
entity For     : Loop;
entity WhileDo : Loop;
entity DoWhile : Loop;

bodyOf      : Block, Loop;
conditionOf : Expr, Conditional;

initOf      : Block, For;
incrementOf : Block, For;

thenOf : Block, If;
elseOf : Block?, If;
```

```
srcOf : Expr, Assn;
dstOf : LExpr, Assn;

headOf : Statement, Seq;
tailOf : Statement?, Seq;

returnOf : Expr?, Return;
```

These entities and relations encode conventional control flow graph; note, the relation `bodyOf` is *overloaded* — we already had the relation with the same name but for entities of different types.

The next section encodes expressions:

```
entity LExpr : Typed;
entity Expr  : LExpr;

entity Var   : Symbol, LExpr;
entity Const : Expr (
  value : Int;
)
entity Elem : LExpr;

entity Binary : Expr (
  op : String
)

entity Unary : Binary;
entity MethodInvocation : Expr;

leftOf  : Expr, Binary;
rightOf : Expr, Binary;

opndOf : Expr, Unary;

arrayOf : LExpr, Elem;
indexOf : Expr, Elem;

instanceOf : MethodInvocation, Expr;
methodOf   : MethodInvocation, Symbol;
argsOf     : MethodInvocation, Expr*;
```

We distinguish here the general-form expressions `Expr` from those which can appear in the left-hand side of an assignment statement `LExpr`; all expressions are typed. The relations at expression level encode expression dags.

Next, we extend the definition of simplified JAVA code model with the layer of types:

```
entity PrimitiveType : Type;

entity Char    : PrimitiveType;
entity Integer : PrimitiveType;
entity Long    : PrimitiveType;
entity Float   : PrimitiveType;

entity TypeArgument;
entity TypeBound;

entity ReferenceType : Type, TypeArgument;
entity ClassType     : ReferenceType, TypeBound;
entity ArrayType     : ReferenceType;
entity NullType      : ReferenceType;

entity TypeParameter    : Type, TypeArgument, Named, TypeBound;

entity IntersectionType : TypeBound;

entity WildcardType : TypeArgument;

elementOf    : ArrayType, Type;
classOf      : ClassType, Class;
componentsOf : IntersectionType, (ClassType | TypeParameter)+;
boundOf      : TypeParameter, TypeBound;
bounfOf      : WildcardType, TypeBound;
argumentsOf  : ClassType, TypeArgument*;
```

These definitions encode a part of JAVA type system including generics following JLS.

The following final section completes the definition of simplified JAVA code model:

```
argumentsOf  : Method, Var*;
returnTypeOf : Method, Type;

typeParametersOf : Class, TypeParameter*;
typeParametersOf : Method, TypeParameter*;
superclassTypeOf : Class, ClassType;
```

These relations close the gap between object-oriented, expression and type levels of the description.

3 Some Observations

We can make a few important observations.

First, code model acquired using the approach we advocate will be on a higher abstraction level than a concrete representation used, for example, in compilers. Namely, the model itself does not ensure the satisfiability of some important constraints: for example, there is no way to say that in an assignment the types of source and destination must agree. The reason is that we suppose that even if a model could specify such coinstraints it would not automatically provide a way to satisfy them. Another point is that some tools (notably, IDEs) often deal with an imcomplete code in which such constraints are (temporarily) violated.

Another observation concerns the completeness of the meta-model specification language. A careful reader could notice that, in fact, given a conext-free specification of a subject language syntax the code meta-model can be automatically derived (or, more precise, can be derived the layer of the meta-model concerning the syntax of the subject language). Thus constitutes a strong argument in favor of the specification language completeness. Moreover, if other layers of the subject language (for example, typing) are described in a syntax-directed mannar (as it is done for the vast majority of conventional languages) then these layers also can be encoded in the meta-model specifiation language as well.

4 Levels of Representation

5 Historicity

References

[1] James Gosling *et al.* The Java Language Specification. *Java SE 19 Edition*, 2022-08-31 // `https://docs.oracle.com/javase/specs/jls/se19/jls19.pdf`