COOL

Audun Linjord Simonsen
Sander Jernquist Otterlei
Joel Andreas Hallheim Reiersøl

December 2018

Abstract

This report describes the implementation of the programming language Classroom Object-Oriented Language (COOL) in Jetbrains MPS, and generating it into Java. The approach used in this project was to first create the structure, and then the other aspects could be worked on independently. The results are a working editor for COOL in MPS, with a partial generation into Java.

1 Introduction

The task was to implement a chosen language in Jetbrains' Meta Programming System (MPS) [5], which in this case is Classroom Object-Oriented Language (COOL). MPS is a tool to create domain-spesific languages. The task was to use the various modules in MPS; structure, editor, typesystem, etc., in order to re-create the chosen language as accurately as possibly. In this project we also generate the chosen language into Java, in order to run it. The COOL manual [2] was used extensively in the making of COOL in MPS.

The code for the project is available in git.

Chapter 2 describes COOL in detail.
Chapter 3 describes the groups solution to the task.
Chapter 4 discusses the solution.
Chapter 5 describes the plan and the groups reflection on the project.

2 Language

The implemented language COOL, which was made by Alexander Aiken [1] in order to teach compiler construction, is a small language which retains many features of a modern programming language. Meaning, the language has objects, static typing and automatic memory management. COOL is an expression language, and most constructs in the language are expressions. Every expression has a value and a type. The expressions are:

- Constants
- Identifiers
- Assignment
- Dispatch
- Conditionals
- Loops
- Blocks
- Let
- Case
- New
- IsVoid
- Arithmetic and Comparison Operations

COOL is type safe, so every construct in the language has an explicit type. This will guarantee no runtime type errors.

A COOL program consist of one or several classes, which can be spread across several files, and in COOL every class identifies a type. A 'Main' class with a 'main' method is required in order to run a program. A few baseclasses are provided by COOL, these are Int, String, Bool, IO and Object.

All classes can inherit from maximum one class as long as they don't inherit from each other. Meaning class P inherits from class C, but class C cannot inherit from class P. If an inherited class is not specified the class inherits, from the baseclass Object. Classes can override methods from the classes it inherits from. The overridden method is used by default, unless the user specifies a super class when calling the method.

A class can contain one or several features, which is a combined term used for methods and attributes in the COOL manual. All features' names must begin with a lower case letter. For features defined in the same class, no two methods and no two attributes can have the same name, but a method can have the same name as an attribute.

An attribute is the creation of a new object of a given class. The attribute consists of an identifier, a type, and can be given an expression. The identifier is the attributes name which is used to access the attribute. The expression is used as an initialization when creating an object. If no expression is given, the types default initialization is used. The default initialization for the classes Int, String or Bool gives the values 0, empty string and false. Other classes give the value 'void' if the default initialization is used.

A method consists of the methods identifier, parameters and an expression. The identifier is used to refer to the method. The parameters can be zero or more.

3 Solution

3.1 Structure

The structure for our solution is given in the UML diagrams below. These diagrams are also available on git.

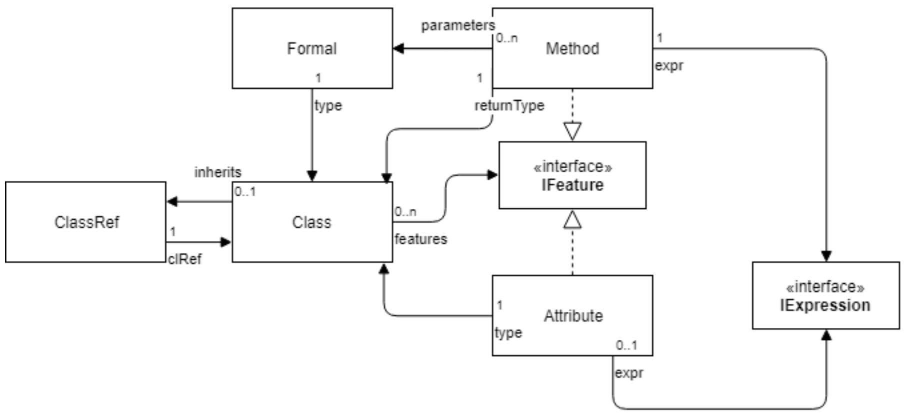


Figure 1: Concepts

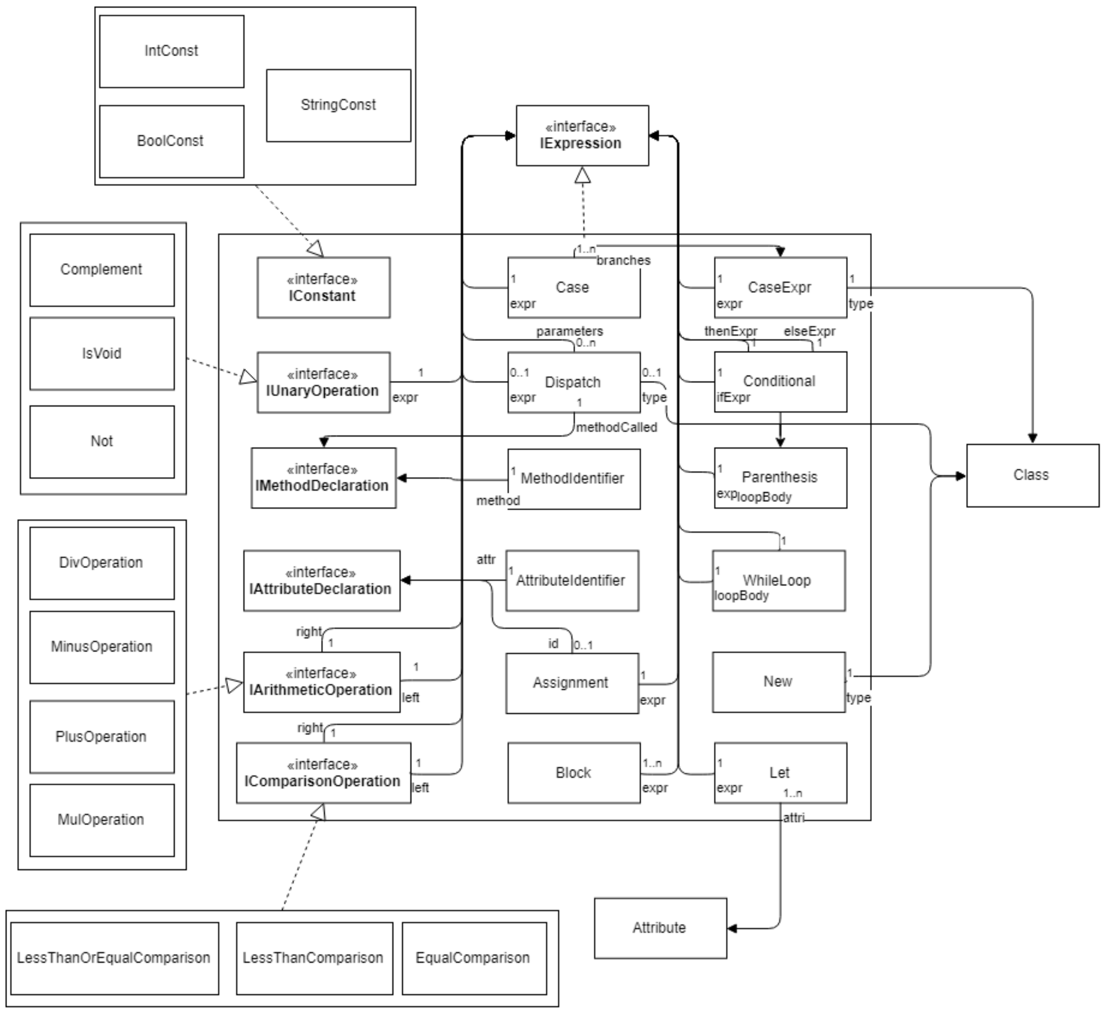


Figure 2: Expression Concepts

Most of the elements in the diagram are the ones given in the COOL manual, but some are introduced in order to have an easier time when creating the editor, and when functionality is introduced, since the elements are easier to separate. The introduced elements are:

- ClassRef
  Which is a reference to a class. Is used when referring to a classes type
- IFeature
  Which is an interface that both method and attribute implements. This is done as Class can have a single feature, and since a method and an attribute are both formals.
- IExpression
  Is an interface that all expressions implements. Which makes it easier when an expression can have another expression in itself.
- IConstant
  An interface which all of the basic constants implements.
- IMethodDeclaration
  An interface which holds the actual reference of a method
- IAttributeDeclaration
  An identifier unique for attributes, such that they are easier to access and reference when writing Cool.
- IArithmetichOperation
  An identifier unique for methods, such that they are easier to access and reference when writing Cool.
- IUnary
  An interface which all the unary operations implements. this gathers all the unaries and makes them easier to handle as they work quite similarly.
- IComparisonOperation
  An interface which all the comparison operations implements. this gathers all the comparsions and makes them easier to handle as they work quite similarly.
- AttributeIdentifier
  As identifiers are also considered expressions, the attributeIdentifier allows to access names as expressions.
- MethodIdentifier
  As identifiers are also considered expressions, the methodIdentifier allows to access names as expressions, though for methods their names are used in dispatches, and aren't normally referenced outside of this normally.
- CaseExpr
  An expression unique for the expression Case.

3.1.1 Constraints

The following constraints was added to the language:

Class: In a class the name has to be different from the other classes, and start with an uppercase letter followed by a combination of either letters, numbers or underscores. A class name cannot be one of the keywords of the language. A class cannot contain an inheritance loop. Lastly, there must be a class named Main and this class must contain a method named main.

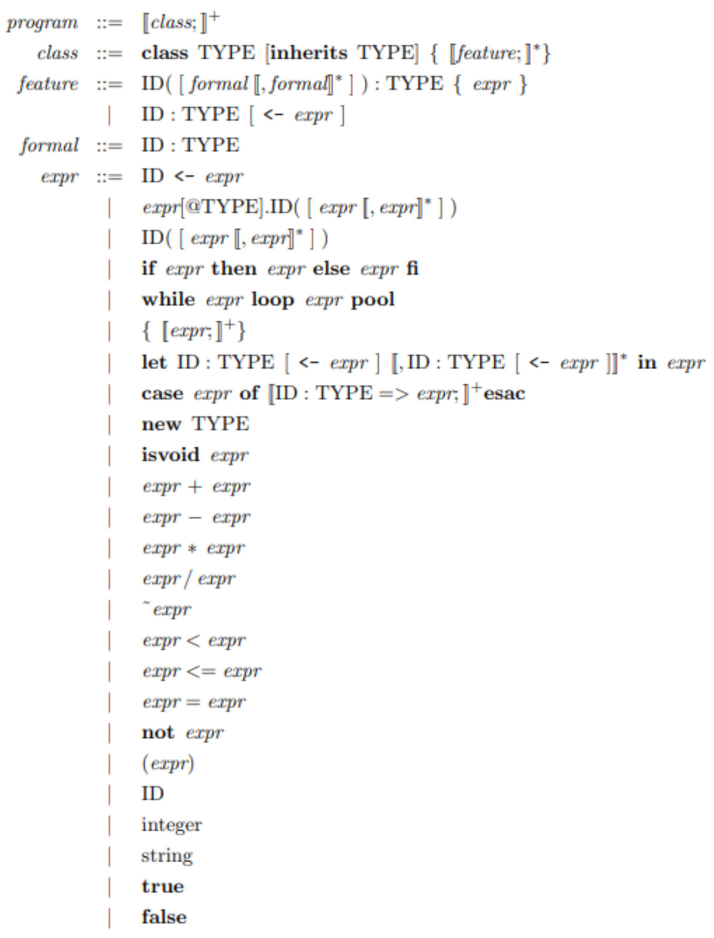
Attribute and method: In both of these the names can not be in the list of keywords. Both has to start with a lowercase letter followed by a combination of letters, numbers or underscores. An attribute can not have the same name as an attribute, and a method can not have the same name as a method.

3.1.2 Typesystem

The typesystem uses COOLs static checking rules, in order to constrain wrong usage of types. Since COOL is generated into Java, Java handles the dynamic checking of the types. Thus, only static checking is needed in the typesystem. Static checking means the COOL code is checked for errors that can be seen before runtime. The typerules has been implemented via MPS' typesystem. The typerules for the static checking can be found in the COOL Reference Manual.

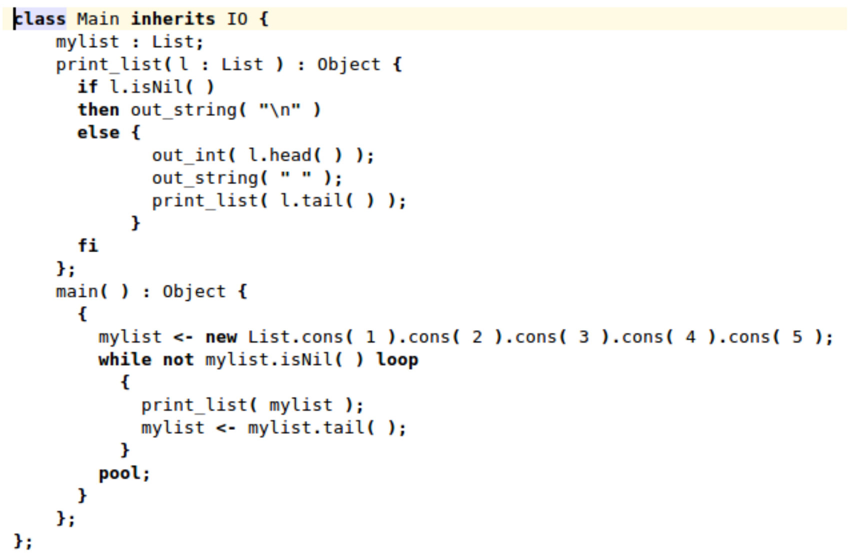
3.2 Syntax

For the syntax of the COOL language, there is an already defined way of how the different elements should be structured, which is defined in the COOL manual. Also defined in the manual is the syntax of each of the elements. This syntax was easy to port into MPS' highly customizable editor module.



```
program ::= [[class; ]]+
  class ::= class TYPE [inherits TYPE] { [[feature; ]]*}
feature ::= ID( [ formal [[, formal]]* ] ) : TYPE { expr }
          | ID : TYPE [ <- expr ]
 formal ::= ID : TYPE
   expr ::= ID <- expr
          | expr[@TYPE].ID( [ expr [[, expr]]* ] )
          | ID( [ expr [[, expr]]* ] )
          | if expr then expr else expr fi
          | while expr loop expr pool
          | { [[expr; ]]+}
          | let ID : TYPE [ <- expr ] [[, ID : TYPE [ <- expr ]]]* in expr
          | case expr of [[ID : TYPE => expr; ]]+esac
          | new TYPE
          | isvoid expr
          | expr + expr
          | expr − expr
          | expr ∗ expr
          | expr / expr
          | ˜expr
          | expr < expr
          | expr <= expr
          | expr = expr
          | not expr
          | (expr)
          | ID
          | integer
          | string
          | true
          | false
```

Figure 3: The syntax for most COOL structures can be written with this grammar [2]



```
class Main inherits IO {
    mylist : List;
    print_list( l : List ) : Object {
      if l.isNil( )
      then out_string( "\n" )
      else {
             out_int( l.head( ) );
             out_string( " " );
             print_list( l.tail( ) );
           }
      fi
    };
    main( ) : Object {
      {
        mylist <- new List.cons( 1 ).cons( 2 ).cons( 3 ).cons( 4 ).cons( 5 );
        while not mylist.isNil( ) loop
          {
            print_list( mylist );
            mylist <- mylist.tail( );
          }
        pool;
      }
    };
};
```

Figure 4: An example of how the COOL editor looks

3.3 Semantics

For the execution of a COOL program, it is translated it Java. In order to generate COOL to Java the MPS' model-to-model transformation is used. This translates COOL concepts into Java concepts, with the help of templates.

In order to include the baseclasses in the generation and running of the Java program, we added a runtime module with pre-generated Java code for the baseclasses.

3.4 What works

The parts of the COOL project that works and the group is proud of is the following:

- Structure

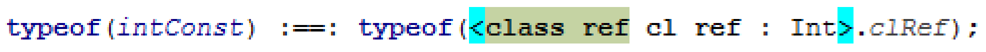
  The structure captures the elements of COOL thoroughly. This is the aspect that has been worked on the most and is well defined. A structure is required for working with the other aspects of MPS so a good structure is important.

- Accessory Models

  Using accessory models is a great way to implement the COOL baseclasses in MPS.

- Typesystem

  The group is proud of the typing rules for Int, Bool and String constants. These use <quotation> in MPS to create a ClassRef node that can reference the Int, Bool and String classes made with accessory models.

  ```
  typeof(intConst) :==: typeof(<class ref cl ref : Int>.clRef);
  ```

- Editor

  The editor that is implemented captures the syntax of COOL accurately. This is based on the COOL manual, so should be the correct syntax of COOL. Side transformations are added to enhance the experience of the editor, and to make it feel more like a text editor without the use of the completion menu.

- Generator

  The generator is not complete, but the group is proud of the model-to-model transformations that works.

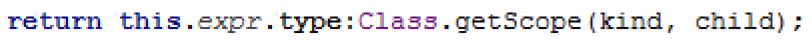
3.5 Known problems

These are the currently known problems that have not yet been fixed:

- Assignment does not properly put in the written expression. The side transformation when typing '<-' does not add what was before the arrow as the identifier.

- Methods and attributes cannot be created from the completion menu. Side transformations must be used for these.

- Subtyping does not work, including that all types are subtype of Object
- The dispatch scope when specifying an expression works in the completion menu, but gives an error about 'out of search scope' when selecting it.

  ```
  return this.expr.type:Class.getScope(kind, child);
  ```

  The fix for the code above was to include every scope, so programs could be written correctly.
- Scope for static dispatch has not been implemented
- Static dispatch, Case and Let is not included in generation
- Stringing dispatches together is tricky. They also do not generate properly.
- The return of a block does not work properly in the generator with nested if statements
- Equal comparisons does not work with Strings because Java use the equals() method instead of '=='. This is not captured by the generator at this time.
- Models needs to manually add the runtime module dependency. This is not added automatically by the generator.

4 Discussion

For covering the language, the implementation has a fully defined structure. Structure is well defined with the use of interfaces grouping the elements nicely. Concepts and editors for all the elements of COOL are captured so COOL programs can be written in MPS.

The implementation has static typing with typesystem, and should cover the cases described in the COOL manual. However, subtyping is not working currently, assumingly a minor issue that has an easy solution.

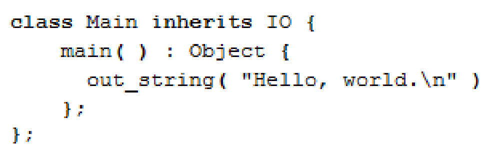
The editor has optional elements like the 'inherits' text in class, that is hidden unless an inherited class is specified. The group decided to use side transformation to alter these elements, instead of intentions, because side transformations also makes the editor feel more like a real text editor. Currently, no side transformation has been added for static dispatch, and an intention must be used. Using side transformations feels more fluent when one knows how to write COOL. However, for others this can be a disadvantage as choosing from a list helps them when they do not know what is expected.

The scope for dispatch is supposed to return the scope from the specified expression. A working solution for this was not found, so the scope includes every scope from the model and baseclasses. This was done in order to still allow correct programs to be written.

In the generator we cannot generate the required COOL method named 'main' because that is reserved for the main method in Java. Thus, we altered it in generation to be named 'cool_main'. The main method in Java then calls the cool_main. This means that users cannot write a function called cool_main in their program since this is used in the generation of the Java code.

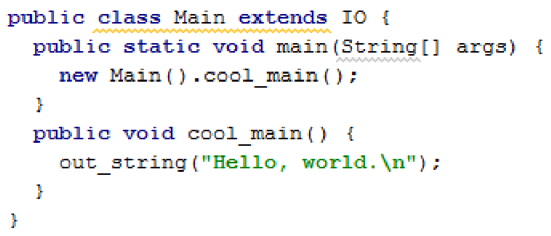
4.1 Correctness

In order to check if the COOL structure could be captured, examples from CoolToJs [3] was used. Below is an example of the Hello World program.



```
class Main inherits IO {
    main( ) : Object {
      out_string( "Hello, world.\n" )
    };
};
```

Figure 5: Hello World
written in COOL inside MPS



```
public class Main extends IO {
  public static void main(String[] args) {
    new Main().cool_main();
  }
  public void cool_main() {
    out_string("Hello, world.\n");
  }
}
```

Figure 6: Hello World
generated into Java inside MPS

Unfortunately, the group could not get the generated Java code to run in MPS, so the code was copied over to Eclipse [4]. Running the code in Eclipse gives "Hello World." in the output.

All the examples we tried to enter could be captured in MPS. However, some can not be generated as Case and Let is not implemented. Also, stringing method calls together does not work when generating.

4.2 Future steps

- Focus on fixing the known problems
- Improve the type system such that there is proper sub-typing.
- Fix scoping for dispatches with expression and adding scoping for static dispatches.
- Add Static dispatch, Case and Let to the generator.
- Fix return values when generating Blocks.
- Run the generated Java files correctly inside MPS.

5 Plan and reflection

Over the project period, the group had weekly meetings with supervisor. Therefore, the group decided to work at least one day every week, when all members had time, to improve the project for next meeting.

Working with MPS was a new experience for all the group members, thus the efficiency of the work was reduced by also learning the tool. Problems often occurred and was frustrating, as sometimes the MPS documentation does not cover aspects in-depth. Near the deadline of the project the group had more time to work, but also stumbled upon last minute problems. The project could be improved upon indefinitely, so the group had to settle at some point.

6 References

[1] Alexander Aiken. “Cool: A portable project for teaching compiler construction”. In: *ACM Sigplan Notices* 31.7 (1996), pp. 19–24.

[2] Alexander Aiken. “Cool: A portable project for teaching compiler construction”. In: (2000). URL: `http://theory.stanford.edu/~aiken/software/cool/cool-manual.pdf`.

[3] *CoolToJs*. `https://nathanfriend.io/cooltojs/`. Accessed: 2018-12-05.

[4] *Eclipse IDE*. `https://www.eclipse.org/downloads/packages/release/mars/r/eclipse-ide-java-developers`. Accessed: 2018-12-05.

[5] *Meta Programming System*. `https://www.jetbrains.com/mps/`. Accessed: 2018-12-05.