Java Type Migration Language - JTML

Ameya Ketkar

1 Introduction

Several scenarios might obviate the need for an automated refactoring where an existing type should be replaced with another one, i.e., *Type migration.* We encountered one such scenario when we studied the adoption of Java 8's lambda expressions (backed by functional interfaces) [1]. Other scenarios could be replacing a deprecated class, maintaining a common client code with multiple alternative APIs or updating the client code when the API gets updated.

Researchers [2, 3, 3, 4] suggest an approach to perform type migration by solving type constraint,and [5, 6] suggest a transformation language to perform this. While the transformation language approach has limitations namely the APIs need to be completely interchangeable and assumption of close world where the entire source code is available.The limitations of solving type constraints is that this approach does not scale beyond a few hundred thousand lines of code. Also, all these techniques are IDE-dependent making it unsuitable for adapting them as a API monitoring and correction tool.

Inspired from the misuse observed, we proposed technique for scalable type migration, and a tool for correcting these misuses.This tool integrates with the build system, making it feasible to run it on a cloud. For a source type T and destination type R, this technique : (1) translates AST Nodes of type T into an abstract semantic graph or term graph (2) merges these graphs, to discover type dependencies throughout the source code (3) enriches the graphs with mapping specifications (4) Finally, translates these enriched graphs into source code transformations. Our results show that, this tool can scale to analyzing source code of size 300 million lines in 33 mins.

This tool currently only works for migrating the misuses we observed in Java. The goal of this DSL - Java Type Migration Language (JTML) is to, generalize this tool to be used for multiple user scenarios of API breaking changes [7]. These can be simple changes like : Move, Rename, Pull Up, Remove Parameter, Parameters Reordered; and much more complex changes like : Change Type, Supertype Change, Add to interfaces.

2 Users

JTML can be used by any Java developer who intends to perform type based refactorings. Users of JTML could be :

- a source code owner who wants to continuously monitor and correct misuses of certain APIs,
- a library maintainer who want to propagate their API updates into client code,
- a developer who wants to maintain interchangeable APIs for common client code,
- a developer who just wants to perform a class migration through the source code (e.g. migrating to a new logging library).

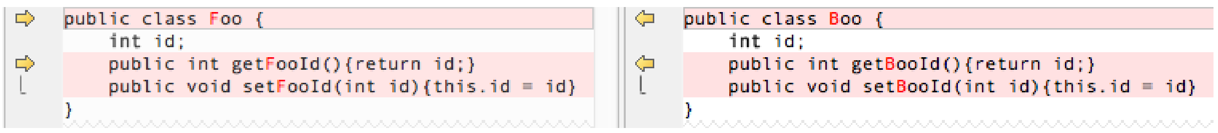
Listing 1: Output for Foo to Boo mapping

```
Call Site : (Allocation Foo ( getFooId ( int [])))
==> Call Site : (Allocation Boo ( getBooId ( int [])))
-- [Change: Name getBooId,Change: Receiver boo]

Call Site : Allocation Foo ( setFooId ([int]))
==> Call Site : Allocation Boo ( setBooId ([int]))
-- [Change: Name setBooId,Change: Receiver boo]

New Class : (Foo (Foo []))
==> (Boo (Boo []))
-- [Change: Name Boo]
```

Using JTML, the users will be able to maintain their migration mappings in a change-oriented fashion. Programs in this DSL will define two types `T` and `R`, and a description to transform the idioms of type `T` to type `R`. Changes to the idioms will be expressed by the operators in the JTML, and these transformations could be reused. Since, JTML focuses on the transformation changes, it can prove very fruitful when the transformation have certain patterns. Though JTML can be used in a variety of scenarios, it should be noted that JTML cannot be used to express changes which need contextual data from the source code. Therefore, refactorings like adding new argument are impossible to be expressed with JTML unless the arguments to be added have a pre-defined default value.

3 Outcomes

When the JTML program is executed it will produce a formatted string output encoding the transformation instructions to manipulate a AST Nodes. In the future, this output will be then fed into the tool described above, to perform these changes on the source code. A output for migrating from class Foo to Boo is shown in Figure 1.



```
public class Foo {
    int id;
    public int getFooId(){return id;}
    public void setFooId(int id){this.id = id}
}
```

```
public class Boo {
    int id;
    public int getBooId(){return id;}
    public void setBooId(int id){this.id = id}
}
```

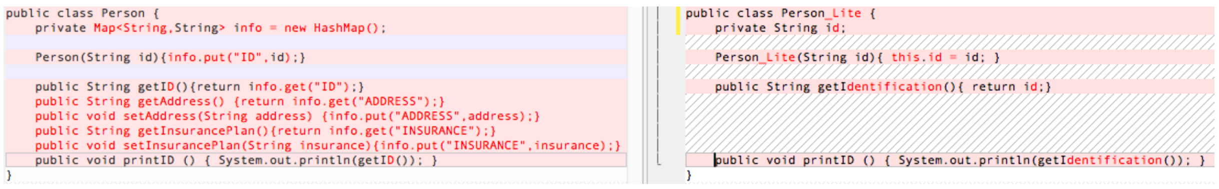
Figure 1: class Foo and class Boo

Listing 1 shows the output of JTML, when a correct program to migrate from `Foo` to `Boo` is executed. The output formatted as <AST Node Before> => <AST Node After> − [Transformation instructions]. CallSite denotes method invocations in the client code belonging to `Foo`, Allocation denote any variable declaration of type `Foo` or method declarations returning `Foo` in client code.

Every JTML program has two parts (1) *defining types* (2) *defining migrations*. For the above transformation, user will have to define `Foo`, `Boo` and `int` as `Type`. Listing 2 shows how types are defined using JTML. This part of JTML can easily be automated by parsing the Java files of the source and destination classes. In rest of the paper, *defining types* will not be included in sample programs.

Listing 2: Type Declaration and Transformation in JTML

```
---Type Declaration ---
getFoo = (Allocation "Foo",("getFoo", ("int", [])))
setFoo = (Allocation "Foo",("getFoo", ("void", ["int"])))
foo    = Ty ("Foo", [getFooId,setFooId])
int    = Ty ("int",[])

---Transformation description ---
fooToBoo = (("Foo","Boo"),[(CALLSITE ["setFooId"], (C  Name "setBooId")),
                            (CALLSITE ["getFooId"],(C Name "getBooId"))])
```



```
public class Person {
    private Map<String,String> info = new HashMap();

    Person(String id){info.put("ID",id);}

    public String getID(){return info.get("ID");}
    public String getAddress() {return info.get("ADDRESS");}
    public void setAddress(String address) {info.put("ADDRESS",address);}
    public String getInsurancePlan(){return info.get("INSURANCE");}
    public void setInsurancePlan(String insurance){info.put("INSURANCE",insurance);}
    public void printID () { System.out.println(getID()); }
}
```

```
public class Person_Lite {
    private String id;

    Person_Lite(String id){ this.id = id; }

    public String getIdentification(){ return id;}

    public void printID () { System.out.println(getIdentification()); }
}
```

Figure 2: class Person and class Person-Lite

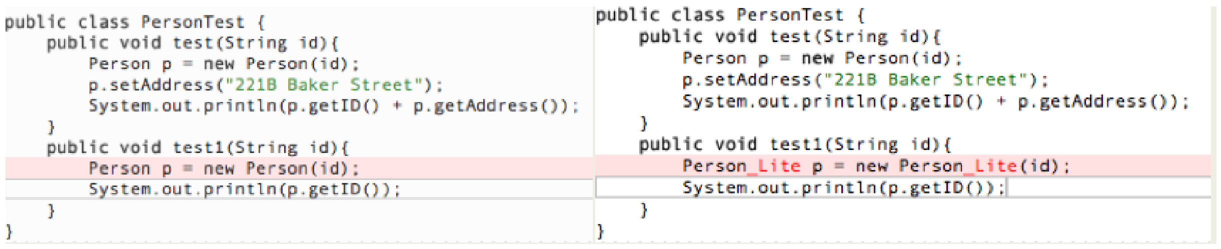
Defining types is a critical part of the JTML, since these are used for type checking the JTML migration program.Listing 3,shows a sample program for migrating from `Foo` to `Boo`. This program undergoes a certain type checks like, if the type `Foo` and `Boo` are defined by the user, `setFooId` and `getFooId` belong to `Foo`. After the program is executed, it is also checked if the resulting callsite after transformation belongs to `Boo` .

4 Use Cases

There are several use-cases for a type migration tool, it can be used for monitoring and correcting certain API uses, minimize the use of a deprecated class or updating client code when types change in APIs.

4.1 API Monitoring and Correcting tool

Imagine a scenario, where a company uses `Person` (Fig. 2) to store and retrieve information of its employees. The developer soon realize that most instances of `Person` are used to retrieve ID of the employee and the other methods were less used. So to prevent the overhead of passing around a heavy object, they come up with a Person-Lite object and want to maximize its use now and in the future.



```
public class PersonTest {
    public void test(String id){
        Person p = new Person(id);
        p.setAddress("221B Baker Street");
        System.out.println(p.getID() + p.getAddress());
    }
    public void test1(String id){
        Person p = new Person(id);
        System.out.println(p.getID());
    }
}
```

```
public class PersonTest {
    public void test(String id){
        Person p = new Person(id);
        p.setAddress("221B Baker Street");
        System.out.println(p.getID() + p.getAddress());
    }
    public void test1(String id){
        Person_Lite p = new Person_Lite(id);
        System.out.println(p.getID());
    }
}
```

Figure 3: Client code migration

Listing 3: Program and output for migrating Person

```
—— Program ——
(("Person","Person_Lite"),[((CALLSITE "getID"),
                    (C Name "getIdentification"))])

—— Output——

Call Site : Allocation "Person" (getID(String()))
==> Call Site : Allocation "Person_Lite"(getIdentification(String()))
 ... [Change: Name getIdentification ,Change: Receiver Person_Lite]
Call Site : Allocation "Person"(getAddress(String()))
==> X ... [DO NOT MIGRATE ]
Call Site : Allocation "Person"(getInsurance(String()))
==> X ... [DO NOT MIGRATE ]
Call Site : Allocation "Person"(setID(void(String)))
==> X ... [DO NOT MIGRATE ]
Call Site : Allocation "Person"(setAddress(void(String)))
==> X ... [DO NOT MIGRATE ]
Call Site : Allocation "Person"(setInsurance(void(String)))
==> X ... [DO NOT MIGRATE ]
Call Site : Allocation "Person"(printID(void()))
==> Call Site : Allocation "Person_Lite"(printID(void()))
 ... [IDENTITY,Change: Receiver Person_Lite]
Allocation : Person
==> Allocation : Person_Lite
 ... [Change: To Person_Lite]
```

For such a refactoring, as shown in Fig. 3, not all instances of `Person` will migrate to `Person_Lite` because not all methods of `Person` has a counterpart idiom in `Person_Lite`.

Doing this manually will involve a lot of effort, first to find all uses of `Person`, then to analyse them to narrow down on what could or could not migrate to `Person_Lite` and finally migrating the ones that are possible. Current transformation languages like *Twining* [5] and *SWIN* [6] do not support this kind of migration since they require the two classes `Person` and `Person_Lite` to be completely interchangeable. The program and output for `Person -> Person_Lite` is shown in Listing 3.

This output has mapping for `getID` to `getIdentification`, the static analysis figures that other methods have no mapping, thus marks them *DO-NOT-MIGRATE*. Also, the method `printID` is analyzed to have a *IDENTITY* mapping. Note that the user did not have to specify this identity mapping. This helps programmer to focus on changes in mappings only, in a expressive and succinct manner.

4.2 Replacing deprecated class

Consider, a developer wants to remove the usage of `Vector` with the new `ArrayList`. Assume `Vector` and `ArrayList` are interchangeable, and so the migration between them can be expressed using languages like *Twining* and *SWIN*. Listing 4 shows the mapping between `Vector` and `ArrayLists` for 6 methods in Twining and JTML syntax.

One can observe that, not a lot really changes in these 6 methods. Except that the receiver of the call

Listing 4: Migration program in Twingina and JTML

```
——Twining ———
[ Enumeration (Vector v) { return v.elements(); }
Iterator (ArrayList a) { return a.iterator(); } ]
[ boolean  (Vector v,Object i) { return v.add(i); }
  boolean (ArrayList a,Object i) { return a.add(i); } ]
[ boolean  (Vector v,Object i) { return v.remove(i); }
  boolean (ArrayList a,Object i) { return a.remove(i); } ]
[ Object  (Vector v,int i) { return v.get(i); }
  Object (ArrayList a,int i) { return a.get(i); } ]
[ boolean  (Vector v) { return v.removeAll(); }
  boolean (ArrayList a) { return a.clear(); } ]
[ int  (Vector v) { return v.size(); }
  int (ArrayList a) { return a.size(); } ]

——JTML ————
mapping =( ("Vector","ArrayList") ,
           [(CALLSITE ["element"],(C Name "iterator"
           `Seq` C ReturnType "Iterator"))
           , (CALLSITE ["removeAll"],(C Name "clear"))])
```

site change for all(which is a obvious change), return type changes for `elements() -> iterator()` and name changes for `removeAll() -> clear()`. It can be seen how, JTML program helps user focus towards changes and manage them better.The user first provides the change in types, and then provides call site mappings in that context.

The JTML program when executed, would result in output shown in Listing 5. The output shows that all 6 methods get correctly mapped just with those two lines of JTML. program.

5 Basic Objects

The basic syntax of JTML has been shown in Listing 6. `Mapping` denotes a JTML program, which is a list of transformation between two types. These transformations are expressed as a `((TypeName,TypeName),[LCTransform]` .The first element denotes the source type `T` and destination type `R` , while the second element contains the list of instructions needed to transform all idioms of `T` to `R`. Each instruction is a product of the idiom to transform(`ConstructKind`) and the changes to perform `Chng` upon that idiom.

Listing 6: JTML's abstract syntax

```
type TypeName = String

type Mapping = [((TypeName,TypeName),[LCTransform])]

type LCTransform = (ConstructKind,Chng)

data Chng = Seq Chng Chng
          | E SubConstruct (String -> String)
          | C SubConstruct String
```

Listing 5: Vector to ArrayList Output

```
Call Site : Allocation "Vector"(add(boolean(Object)))
==> Call Site : Allocation "ArrayList"(add(boolean(Object)))
 ... [IDENTITY,Change: Receiver ArrayList]
Call Site : Allocation "Vector"(add(boolean(Object)))
==> Call Site : Allocation "ArrayList"(add(boolean(Object)))
 ... [IDENTITY,Change: Receiver ArrayList]
Call Site : Allocation "Vector"(get(Object(int)))
==> Call Site : Allocation "ArrayList"(get(Object(int)))
 ... [IDENTITY,Change: Receiver ArrayList]
Call Site : Allocation "Vector"(element(Enumeration()))
==> Call Site : Allocation "ArrayList"(iterator(Iterator()))
 ... [Change: Name iterator ,Change: ReturnType Iterator
 ,Change: Receiver ArrayList]
Call Site : Allocation "Vector"(size(int()))
==> Call Site : Allocation "ArrayList"(size(int()))
 ... [IDENTITY,Change: Receiver ArrayList]
Call Site : Allocation "Vector"(removeAll(boolean()))
==> Call Site : Allocation "ArrayList"(clear(boolean()))
 ... [Change: Name clear ,Change: Receiver ArrayList]
Allocation : Vector
==> Allocation : ArrayList
 ... [Change: To ArrayList]
```

Listing 7: Semantics of JTML

```
type Name = String
type Mthd = (Construct,(Name,(TypeName, [(Int,TypeName)])))
data Construct = ClassDecl TypeName [Construct]
                | Allocation TypeName
                | CallSite Mthd
                | MethodDeclaration Mthd
type MappingSem = ((Type,Type),[(Construct,
        ((Construct->Construct),[Chng]))])
type Result =( (Type,Type),
      [((Construct,Maybe Construct),[Chng])])
```

```
        | DO_NOT_MIGRATE
        | IDENTITY

data ConstructKind = CLSDECL | ALLOCATION | CALLSITE [String]
      | NEW_CLASS

data SubConstruct = ReturnType| Param Int | Arg Int | To| Receiver
               | MethodInvc  | OveridenMthd String SubConstruct
               | SubType| Name
```

User gives `Mapping` as input, which is then checked for syntax and type correctness. If they check, this program is translated into `MappingSem` shown in Listing 7. The `MappingSem` are `Mapping` enriched with type information provided by the user(or auto-loaded in the future). This `MappingSem` manipulates the basic object of this domain `Construct`. `ClassDecl` denotes sub type declaration of `T` in the client code, `Allocation` denotes any variable or method declared of type `T` in the client code, `CallSite` denotes method invocations in client code and `MethodDeclaration` denotes the overidden methods during subtyping.

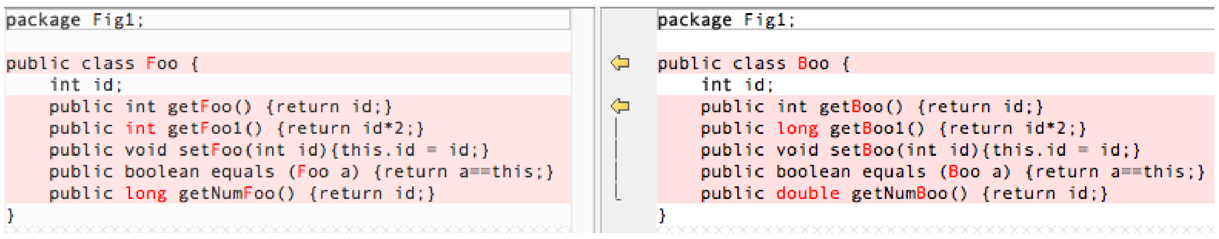
From the above example to migrate Vector to ArrayList,
`(CALLSITE ["removeAll"],(C Name "clear"))`
translates into
`(CallSite (Allocation "Vector",("removeAll",("boolean",[]))),(chngName clear,[C Name "clear"]))`
where `chngName` a function `String -> Construct -> Construct`.

The `MappingSem` is evaluated by applying each `Construct->Construct` function to the corresponding `Construct`. This evaluation results in `Result` Listing 7, which is then checked for correctness.

6 Operators and Combinators

JTML programs basically, transform `Construct` object(s) using `Chng`. `Chng` is the basic operator in JTML. For example in Listing 4, `(CALLSITE ["element"],(C Name "iterator" ‘Seq‘ C ReturnType "Iterator")` transforms the *CALLSITE* of method `element()` by changing the name and the return type. The operator C is defined by Constructors:

- Constructor `C` of `Chng` performs a replace operation of `type` or `name` on the corresponding subConstruct. For example, `C Name` changes name of the callsite, while `C ReturnType` changes the return type.



```
package Fig1;

public class Foo {
    int id;
    public int getFoo() {return id;}
    public int getFoo1() {return id*2;}
    public void setFoo(int id){this.id = id;}
    public boolean equals (Foo a) {return a==this;}
    public long getNumFoo() {return id;}
}
```

```
package Fig1;

public class Boo {
    int id;
    public int getBoo() {return id;}
    public long getBoo1() {return id*2;}
    public void setBoo(int id){this.id = id;}
    public boolean equals (Boo a) {return a==this;}
    public double getNumBoo() {return id;}
}
```

Figure 4:

- `DO_NOT_MIGRATE` construct, denotes that no corresponding idiom is available, so if this callsite is encountered, type migration should not happen.
- The `E` constructor, is one of the most powerful operator in JTML. It accepts a `String->String` function to alter the types and names of `Construct`. Using the `E` constructor, user can maintain a mapping of a particular subconstruct, separately.
- `Identity` denotes that the two constructs are identical in context of the change.

The `Seq` constructor in `Chng` maps multiple changes to a construct or a set of constructs as shown in Listing 4.

To demonstrate the expressiveness,re-usability and composition of JTML programs, we will consider another toy example which is a class library migration Fig. 4. Listing 8 shows a typical JTML program to migrate from `Foo` to `Boo` and its output.

But now imagine, the developer wants to maintain mapping from `Foo` to two versions of `Boo`. One which is shown in Fig. 13 (v1) and another version of `Boo` where the method `getBoo1` returns `Double` instead of `Long` (v2). Now to maintain mappings from `Foo` to v1 and `Foo` to v2, the developer will have to maintain duplicate mappings leading to maintenance overhead. Another way to go about this problem will be to decompose mappings by sub-constructs to be changed, as shown in Listing 9.

The temporary limitations of the JTML are :

- Adding chained method invocation to a callsite.
- Casting expressions is not possible yet
- Capturing expressions in new method invocations

7 Interpretation and Analysis

First, the input JTML program is tested for type and syntactic correctness. The syntax correctness ensures that all Constructs are paired correctly to their SubConstruct. The type correctness ensures that, every type name used is defined and if all the callsites name used belongs to the enclosing type `T`.This program is then interpreted, enriched and represented in the semantic domain `MappingSem`.

The static analysis of `MappingSem`, analyses the transformation code and the type information and collects the unmapped idoms for `T`. For each unmapped idiom it either (1) performs auto- mapping (2) or marks it `DO_NOT_MIGRATE`. Auto mapping, currently looks for identical method (based on method-signature), and maps them. This feature could be extended to use tools which discover mapping between methods using software repository mining techniques.

Finally, the `Result` is also checked for correctness. It is checked that every idiom of type `T` when transformed with the JTML program produces a idiom that belongs to destination Type `R`. Also, in the results it

Listing 8: JTML program and output to migrate Foo to Boo

```
—— PROGRAM ——
fooToBoo =(("foo","boo"), [
                        (CALLSITE ["getFoo"], C ReturnType "boo"
                                                   `Seq` C Name "getBoo"),
                        (CALLSITE ["setFoo"], (C  Name "setBoo")),
                        (CALLSITE ["getNumFoo"],C Name "getNumBoo"
                                                   `Seq` C ReturnType "Double"),
                        (CALLSITE ["getFoo1"],(C Name "getBoo1"
                                                   `Seq` C ReturnType "Long"))])
longToDouble = (("Long","Double") , [])
intToLong = (("int","Long") , [])

—— OUTPUT ——

Call Site : Allocation "foo"(getFoo(foo()))
==> Call Site : Allocation "boo"(getBoo(boo()))
 ... [Change: ReturnType boo,Change: Name getBoo,Change: Receiver boo]
Call Site : Allocation "foo"(setFoo(void(int)))
==> Call Site : Allocation "boo"(setBoo(void(int)))
 ... [Change: Name setBoo,Change: Receiver boo]
Call Site : Allocation "foo"(getFoo1(int()))
==> Call Site : Allocation "boo"(getBoo1(Long()))
 ... [Change: Name getBoo1,Change: ReturnType Long,Change: Receiver boo]
Call Site : Allocation "foo"(equals(bool(int)))
==> Call Site : Allocation "boo"(equals(bool(int)))
 ... [IDENTITY,Change: Receiver boo]
Call Site : Allocation "foo"(getNumFoo(Long()))
==> Call Site : Allocation "boo"(getNumBoo(Double()))
 ... [Change: Name getNumBoo,Change: ReturnType Double,Change: Receiver boo]
Allocation : foo
==> Allocation : boo
 ... [Change: To boo]
```

Listing 9: JTML program from Foo to Boo (v1 and v2)

```
get :: ([(a,b)] -> a -> b

callSiteNameMap = [("getFoo","getBoo"),("setFoo","setBoo"),
                   ("getFoo1","getBoo1"),("getNumFoo","getNumBoo")]

callSiteRetMap = [("getFoo","boo"),("getNumFoo","Double")
                  ,("getFoo1","Long")]

mapping1 =  (CALLSITE ["getFoo","setFoo","getFoo1","getNumFoo"],
                                  (E Name (get callSiteNameMap)) )

mapping2 = (CALLSITE ["getFoo","getNumFoo","getFoo1"],
            (E ReturnType (get callSiteRetMap)))

fooToBooMapV1 =(("foo","boo"), [mapping1,mapping2])

replace :: [(a,b)] -> (a,b) -> [(a,b)]

fooToBooMapV2 =(("foo","boo"),
 [mapping1 ++ (replace mapping2 ("getFoo1","Double"))]
```

is checked that, if any transformation changes the type of construct, the mappings for corresponding types is provided.

8 Implementation Strategy

A deep DSL is a very suitable approach to implement this DSL because this helped in making the syntax succinct. Because of the Deep DSL approach, the user can address callsites and types, just by their names. Though, this leads to a need for a checker to be implemented before the DSL is interpreted in the semantic domain `MappingSem`. A shallow DSL would lead to a very redundant and verbose syntax, making the DSL difficult to use and comprehend, but would need no such checker. Also, the deep DSL strategy gives a lot of scope to add syntactic sugar, and abstracting the user from the lower level details. For example, constructor `C` of `Chng` is a syntactic sugar over the semantics represented by `E`.

9 Future Work

In the future, JTML will be extended to support operations like adding chained method invocations, capturing expressions and casting. Though currently, JTML ensures that the idioms produced on executing a JTML program are type correct no analysis for subtyping is performed. This would also be a focus in the future. Since, JTML programs are just a set of predefined operators, a nice minimalistic GUI(like in emacs or Alpine mail) could be very useful in creating JTML programs.

References

[1] D. Mazinanian, A. Ketkar, N. Tsantalis, and D. Dig, "Understanding the use of lambda expressions in java," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 85:1–85:31, Oct. 2017. [Online]. Available: http://doi.acm.org/10.1145/3133909

[2] R. Fuhrer, F. Tip, A. Kieżun, J. Dolby, and M. Keller, "Efficiently refactoring java applications to use generic libraries," in *ECOOP 2005 - Object-Oriented Programming*, A. P. Black, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 71–96.

[3] F. Tip, R. M. Fuhrer, A. Kieżun, M. D. Ernst, I. Balaban, and B. De Sutter, "Refactoring using type constraints," *ACM Trans. Program. Lang. Syst.*, vol. 33, no. 3, pp. 9:1–9:47, May 2011. [Online]. Available: http://doi.acm.org/10.1145/1961204.1961205

[4] I. Balaban, F. Tip, and R. Fuhrer, "Refactoring support for class library migration," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '05. New York, NY, USA: ACM, 2005, pp. 265–279. [Online]. Available: http://doi.acm.org/10.1145/1094811.1094832

[5] M. Nita and D. Notkin, "Using twinning to adapt programs to alternative apis," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, - 2010, p. nil. [Online]. Available: https://doi.org/10.1145/1806799.1806832

[6] J. Li, C. Wang, Y. Xiong, and Z. Hu, "Swin: Towards type-safe java program adaptation between apis," in *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation*, ser. PEPM '15. New York, NY, USA: ACM, 2015, pp. 91–102. [Online]. Available: http://doi.acm.org/10.1145/2678015.2682534

[7] B. Cossette and R. J. Walker, "Seeking the ground truth: a retroactive study on the evolution and migration of software libraries," in *SIGSOFT FSE*, 2012.