Interactive API Migration Framework

Krishna Narasimhan, Christoph Reichenbach
Goethe University, Frankfurt
krishna.nm86@gmail.com, creichen@gmail.com

Julia Lawall
INRIA, Paris
Julia.Lawall@lip6.fr

Abstract

This is the text of the abstract.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords keyword1, keyword2

1. Introduction

As source code evolves, large scale transformations are unavoidable. These can be a result of the need to introduce a new API, upgrade usages of existing API, perform code optimizations or simply do a code clean-up. Current support for such transformations comes in the form of refactorings from Integrated Development environments, transformation languages or simply domain specific libraries.

A scenario that occurs frequently during code evolution is migration of types, either to exploit enhanced capabilities as part of a library, or simply getting rid of a datatype that is hard to manage relative to the goals of the software. Examples include migrating from single precision numeric types to multiple precision types by introducing the GMP library, or explicitly vectorizing arrays and loops by introducing the VC library, or simply migrating usages of character pointer to strings. Transformations of such nature are generally not simple cut and replace of method calls or declarations. They require tracking of dependencies, transforming code belonging to different kinds of syntactic classes and tracking information between transformations. Partially performing such transformations could potentially result in semantically and/or syntactically broken code.

Consider the scenario of explicitly vectorizing a array of struct of floats using the VC API. VC is an api that allows explicit vectorization of c++ code at the source level. The VC API provides a custom type called float_v, which is a float vector, with capability to hold more than one float, depending on the hardware implementation. Transforming the array of struct of floats into an array of struct of float_vs would involve transforming the individual members of the struct definition into float vectors, using the information about the member types to construct a new reduced size for the array and transforming all usages of the array object if possible. Similar problems of dependency tracking and propagating information across transformations exist with the other examples we have observed as part of our motivational study.

The existing refactorings in IDEs are limited by the fixed features of the IDEs and require cleverly written scripts to be triggered as multiple refactorings at a time. For example, Eclipse supports a fixed set of refactorings like "Extract function" or "Rename method". IDE refactorings are not programmable, which means they do not support pattern matching and rewriting of different syntactic classes. It would require the developer to write extensive scripts to even trigger multiple refactorings in a specific order. Transformation languages like Stratego/XT and Coccinelle allow specifying transformation rules to express generic patterns matches and code transformations. Stratego/XT uses term rewriting and strategies to combine(or compose) the transformations. Although Stratego/Xt is very powerful, it is limited when it comes to propagating information between rules, as is required when expressing the transformation of the array of structs in the VC example. Coccinelle allows elegant expression of code fragment rewriting in the form of diffs and permits propagating information between rules with rule parameters that can be updated within the rules and queried by other rules. But, Coccinelle does not support applying transformation rules that are triggered by a search through the code dependencies.

We propose to design a language that can be used to express pattern matching and rewriting rules in the form of term equivalences. We specify how rules can be written in the language and introduce features that can be used to express propagating information across rules. The user can trigger the tool by providing a starting point, which is a combination of a code location and a set of rules (specifications) written in the language. The tool would attempt to apply a rule, with feedback from the user, as seems appropriate. The tool would then look through the dependencies of the transformed code and suggest applicable rules for those code locations. The process continues till the all code dependencies are exhausted or if the user chooses to stop the application of rules. The motivation of such a language and complementary tool is as follows:

- The expert user who writes the specifications has the freedom to write only partial specifications, which he can later update.
- The end user can use any set of specifications and has the freedom to choose where and how far to apply them and how much to manually transform.

The tool provides an option to roll back if the user is not happy with the state of transformations.

2. Motivating Examples

2.1 Using the GMP library to transform a single precision factorial into a multi precision factorial

Consider the scenario of converting a function that computes the factorial of a given integer **Figure 1** to be able to return a multi-precision numerical using the GNU's GMP library. Here is a sam-

```
int fact(int n){
  int i;
  int p = 1;
  for (i=1; i \textless = n ; ++i){
    p = p * i;
  }
  return p;
}
```

```
mpz_t fact(int n){
  int i;
  mpz_t p;
  mpz_init_set_ui(p,1);
  for (i=1; i \textless = n ; ++i){
     mpz_mul_ui(p,p,i);
   }
   return p;
}
```

Figure 1. Integer to BigInteger conversion using GMP

ple before and after code of such a scenario. The main steps a developer would perform in going from int to multi precision integer would do are:

- Flag the return value, find its initialization and transform the initialization **int p = 1**. He would then need to use two statements, one to declare and another to initialize the value, to transform the initialization into a vector version.
- Search for usages of this newly transformed value and transform it's operations to reflect multi precision usages. In the above case, the developer would need to convert the assignment **p = p *i;** into **mpz_mul_ui(p,p,i);**
- Change the return type to reflect the type of the new value being returned.

The two key features an automation framework would require to support transformations are : Ability to support expressing transformation rules for pattern matching and rewriting different syntactic classes. Ability to walk through dependencies of transformations performed and then trigger transformation rules on the dependencies, if applicable and roll back the whole transformation, if no transformation rules are applicable.

2.2 Introducing the VC API

Consider another scenario where C++ code undergoes explicit vectorization using an API called VC **2**. This api provides vector versions of primitive types and corresponding operations. Here is a simple example of a code undergoing transformation from scalar to a VC version. The developer would need to perform many steps in order to transform the code in the left hand side to the one in the right hand side. The challenging steps to perform that are:

- The developer would need to analyse the type of **aobj** , and transform the definition of the **struct A**, as it comprises only of float members.
- This transformation of the size of the array **aobj** also involves collecting the information about the type of the members inside the struct, **float** and using the vc version of it to construct the new size.
- Similarly, the user would also need to collect the information about the type of all array indexed elements inside the array in order to transform the limit of the for statement.

Based on our developer intuition, the feature that would be required to perform the above mentioned steps are:

- Ability to collect information from one transformation and pass it on to transformations in other parts of the code.

3. Features in the envisioned Language

We summarize the features that have been extracted out of the analysis of the two api migration examples.

1. Pattern matching and code rewriting
2. Automatic search of dependencies of the transformations
3. Information propagation across transformation patterns

We propose to use the following language features to support the generic features that we extracted out of the examples.

1. Pattern matching and rewriting - Transformation rules in the form of equivalences
2. Information propagation across rules - Tags
3. Search through dependencies - Implicit

In order to understand how the language introduces the equivalences and tags, let us look at the transformation of struct example **Figure 3**.

In the example specification at **Figure 4**, the first line is an example of a tag declaration. The tag, called vctype can be used to propagate a type information. The block B is an example of an pattern match and rewriting rule. It matches a declaration of a struct definition with size $sandbody$body. It is rewritten into a new struct definition, with the rewrite rules on the right hand side of === sign.

The scope vc_struct defines the rules that must be applied on each statement declaring a member type which is either an int or a float. It also populates the vc version of the type of the member declaration, if it matches with the existing value of vctype. This is referenced and used in the transformation of the size. The vc_struct also transforms the member declarations into the new vc types.

4. Informal Syntax and Semantics

Any specification in the language is any combination of :

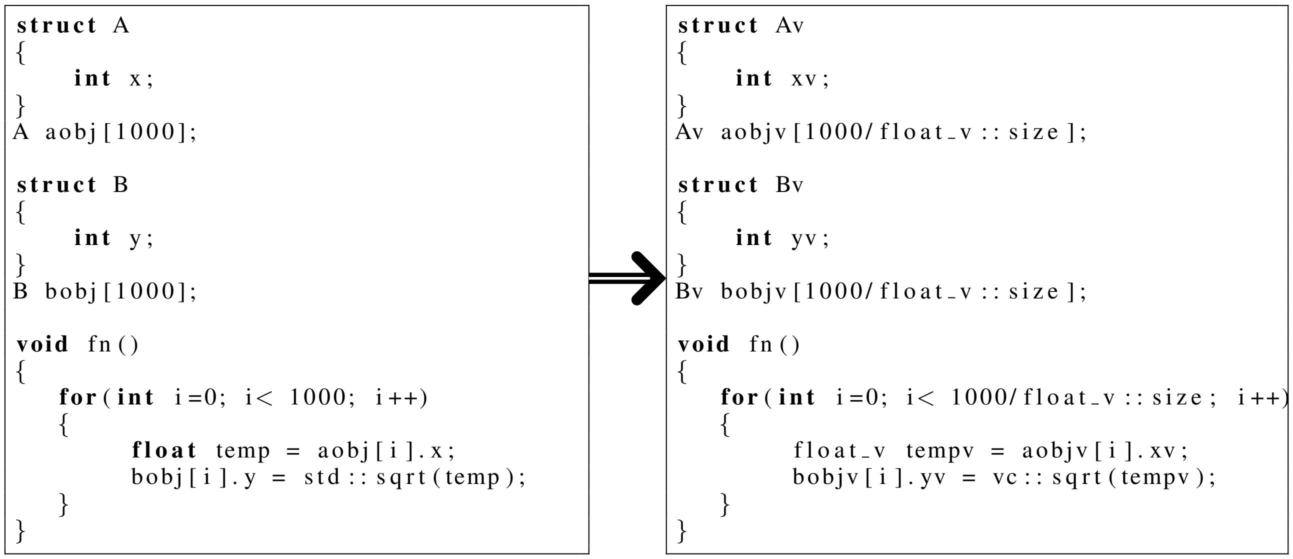
- Tag declaration
- Namespace declaration
- PMT rule (Pattern, Match Transform)
- Scope Block

4.1 Tag Declaration

4.1.1 Syntax

Tag <tagkind><tagname>;;

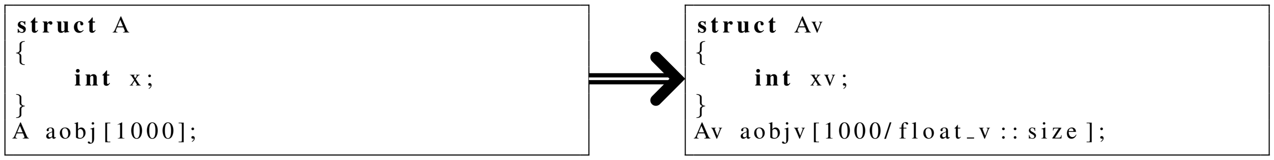
- **Tag** is a keyword specifying that the statement that follows is a tag declaration
- **Tagkind** specifies the kind of information that the corresponding tag can hold. This can be any C++ AST node type. For



```
struct A
{
    int x;
}
A aobj[1000];

struct B
{
    int y;
}
B bobj[1000];

void fn()
{
   for(int i=0; i< 1000; i++)
   {
        float temp = aobj[i].x;
        bobj[i].y = std::sqrt(temp);
   }
}
```

```
struct Av
{
    int xv;
}
Av aobjv[1000/float_v::size];

struct Bv
{
    int yv;
}
Bv bobjv[1000/float_v::size];

void fn()
{
   for(int i=0; i< 1000/float_v::size; i++)
   {
        float_v tempv = aobjv[i].xv;
        bobjv[i].yv = vc::sqrt(tempv);
   }
}
```

Figure 2. Introducing VC



```
struct A
{
    int x;
}
A aobj[1000];
```

```
struct Av
{
    int xv;
}
Av aobjv[1000/float_v::size];
```

Figure 3. Introducing VC

```
tag type  vctype;;

//TTL Expression skeleton to transform array of structs. - Block B
declaration struct $structname$ {$body$}[$s$] ===
          struct $structname$v {vc_struct($body$)}[$s$/(vc_struct,vctype)::size])

/*Block of rules to transform body of struct definition + updating the value of structtype
                   - Block C*/
scope  vc_struct{{
  statement  int $a$; === {{vctype:=int_v}}  $a$v;
  statement  float $a$; === {{vctype:=float_v}}  $a$v;
 }}
```

Figure 4. Specification sample

example, tagkind can be ”type” or ”stringliteral” or ”binaryexpression” or ”expression”. A complete list of possible tagkinds will be available in the formal specifications.

- **Tagname** is an identifier, to permit referencing the tag for update, or for querying. Rules on what qualify as an identifier will be available in the formal specifications.

4.1.2 Semantics

Tags are always updated inside scope blocks. Using the format

<tagname >:= <value >

where **Tagname** is an already declared tag and **Value** is a c++ code conforming with the type of the tag. For example, a tag of kind ”type” cannot be updated with the value ”x + y”, which can be updated into a tag of kind ”binaryexpression” or ”expression”. Tags are queried using the format

<scopename >. <tagname >

where scopename is the name of an existing scope block.

4.2 Namespace Declaration

A namespace declaration declares the two domains, of transformation. Providing a name for the two sides of any PMT rule.

4.2.1 Syntax

Namespace <LDomain >=== <RDomain>;;

- **Namespace** is the keyword specifying the line that follows declares two domains
- **LDomain** an identifier and associates all the left hand side code pattern of any PMT Rule to LDomain.
- **RDomain** an identifier and associates all the right hand side code pattern of any PMT Rule to RDomain.

4.2.2 Semantics

A namespace declaration is always the beginning of any specification file.

4.3 PMT Rule

4.3.1 Syntax

<SyntacticClass><LPMTExpression >=== <RPMTExpression >

- **SyntacticClass** indicates the type of code fragment being matched and transformed. This can be one of :
  - Declaration
  - Statement
  - Expression
- Both **LMPTExpression** and **RPMTExpressions** are **PMTExpressions**, which are explained here.

A **PMTExpression** is a C++ code fragment filled with one or more of the following :

- A **metaname**, enclosed with two $ signs, for example for(int i = 0; i <$limit$; i++) $body$ is a valid PMTExpression, corresponding to the syntactic class, STATEMENT, with three metanames, (i, body and limit) .
- A **tag query**, of the form **<scopename>.<tagname>**
- A **tag update** , of the form **<tagname>:= <value>**. Tag updates can have an optional conditional of the form **where <condition>** , where condition is any c++ expression that evaluates to boolean and can use metanames whose values have been parsed as part of an LMPTExpression.If the condition cannot be evaluated, the tag update fails.

  Tag updates can also happen at the end of a PMTRule enclosed within and . This indicates the updated value of the tag will not be used for the transformation, rather just to pass information.
- A **scope usage**, of the form <scopename>(<metaname>) .

4.3.2 Semantics

- Tag query, scope usage and tag updates can only happen on RPMTExpressions.
- Tag updates can only happen inside PMTRules that are part of a scope block.

4.4 Scope Block

4.4.1 Syntax

<PMT Rule >*

A scope block is any set of PMTRules.

4.4.2 Semantics

A scope rule applies the PMTRules inside it only on the AST node comprising of the code fragment that is passed as an argument when using the scope

4.5 General Semantics

- The initial inputs to tool are
  - A code location (declaration/ expression/ statement)
  - Specification file comprising of Set of PMTRules, Scopes and tag declarations
- Beginning with code location, all applicable rules are presented to the user and the user picks one.
- For an application of a PMTRule on a piece of code.
  - The LPMTExpression(left hand side) is used to gather all the metanames and the corresponding code fragments that match
  - If one metaname is used in multiple places, the code fragments must match, otherwise rule application is terminated.
  - During the pass on the right hand side, all the scope rules are processed, from left to right. If the scopes transform the internals of the code fragment being passed as a parameter, the transformation is reflected.
  - During application of the scope rules, if the value of the tags arent unique, then rule application is terminated
  - After application of all the scope rules, the tags are used to complete the rest of the transformation.

A. Appendix Title

This is the text of the appendix, if you need one.

Acknowledgments

Acknowledgments, if needed.

References

P. Q. Smith, and X. Y. Jones. ...reference text...