

JunGL: a Scripting Language for Refactoring

Mathieu Verbaere
Programming Tools Group

Cakes Talk
November 10, 2005

What is refactoring?

- ▶ Refactoring is the process of gradually improving the design of existing code by performing behaviour-preserving program transformations.
- ▶ It can be done manually but most of the refactorings require tedious, error-prone manipulation of the code.
- ▶ IDEs now provide an automated support for performing some refactorings. For instance, *Rename*, *Extract Method*, *Extract Interface*.

The need of a scripting language for refactoring

- ▶ A wealth of refactorings has been proposed. A few of them are actually implemented.
- ▶ Support to allow developers to author their own refactorings is rudimentary in existing systems.
- ▶ Implementing refactoring transformations is quite complex:
 - ▶ it requires the same kind of analyses as in compiler optimisations;
 - ▶ even the most sophisticated IDEs have bugs in their refactoring support.

Extract Method bug in Visual Studio

```
public void F(bool b)
{
    int i;
    // from
    if (b)
    {
        i = 0;
        Console.WriteLine(i);
    }
    // to
    i = 1;
    Console.WriteLine(i);
}
```



```
public void F(bool b)
{
    int i;
    i = NewMethod(b);
    i = 1;
    Console.WriteLine(i);
}
private static int NewMethod(bool b)
{
    int i;
    if (b)
    {
        i = 0;
        Console.WriteLine(i);
    }
    return i;
}
```

Extract Method bug in Visual Studio

```
public void F(bool b)
{
    int i;
    // from
    if (b)
    {
        i = 0;
        Console.WriteLine(i);
    }
    // to
    i = 1;
    Console.WriteLine(i);
}
```



```
public void F(bool b)
{
    int i;
    i = NewMethod(b);
    i = 1;
    Console.WriteLine(i);
}
private static int NewMethod(bool b)
{
    int i;
    if (b)
    {
        i = 0;
        Console.WriteLine(i);
    }
    return i;
}
```

i in *NewMethod* is returned without necessarily being assigned.

The design of JunGL

- ▶ JunGL is a domain-specific language for refactoring.
- ▶ Hybrid of a functional language like ML and a logic query language akin to Datalog.
- ▶ The main data structure is a graph representing all information about the program we wish to transform:
 - ▶ ASTs,
 - ▶ variable binding,
 - ▶ control flow,
 - ▶ dataflow, ...

Lazy edge definitions

- ▶ Initially, the graph consists just of an AST, with edges indicating *parent* relationship.
- ▶ Additional information is added via lazy edge definitions, that are evaluated when their value is needed.
- ▶ For instance, the control flow edges emanating from an *if* statement are defined with:

```
let IfStmtCFSucc(node) =  
  match (node.thenBranch,node.elseBranch) with  
  | (null , null) → [DefaultCFSucc(node)]  
  | (t, null) → [t; DefaultCFSucc(node)]  
  | (null , e) → [DefaultCFSucc(node); e]  
  | (t,e) → [t; e] ;;
```

```
AddLazyEdge("IfStmt","cfsucc",IfStmtCFSucc) ;;
```

Streams, predicates and path queries

- ▶ In the implementation a refactoring, we need to retrieve some information about the program we wish to refactor.
- ▶ JunGL has the notion of streams (lazily evaluated lists) and predicates:

$$\{ ?x \mid P(?x) \}$$

returns a stream of all x that satisfy the predicate P .

- ▶ Predicates can be build via path queries, i.e. regular expressions that identify paths in the program graph.

Example: Variable binding for a subset of C#

```
let SimpleNameLookup(ref) =  
  let matches = { ?dec |  
  
    // Local variable  
    ([ ref ] pred+[?s:Kind("VariableDeclStmt")] child[?dec:Kind("Declarator")]  
      & ?dec.name == ref.name)  
    |  
    // Parameter  
    ([ ref ] parent+[?m:Kind("MethodDecl")] child[?dec:Kind("ParamDecl")]  
      & ?dec.name == ref.name)  
    |  
    // Field  
    ([ ref ] parent+[?c:Kind("ClassDecl")] child[?f:Kind("FieldDecl")]  
      child [?dec:Kind("Declarator")]  
      & ?dec.name == ref.name)  
  } in  
  FindFirst (matches) ;;
```

JunGL live (1)

- ▶ A prototype of JunGL is implemented on the .NET platform using both *C#* and *F#*.
- ▶ A basic structure editor facilitates the interactive development of the scripts.
- ▶ **Lazy edge demo**

Implementing refactorings

- ▶ Automating *Rename Variable* is far beyond a simple search-and-replace mechanism. We need to detect potential conflicting declarations of variables with a similar name.

Implementing refactorings

- ▶ Automating *Rename Variable* is far beyond a simple search-and-replace mechanism. We need to detect potential conflicting declarations of variables with a similar name.
- ▶ *Extract Method* is even more complex. They are four distinct phases:
 - ▶ Checking the validity of the user selection (it should be a single-entry single-exit block);
 - ▶ Classifying parameters into different sets: *value*, *out*, *ref* parameters;
 - ▶ Deciding on the place of variable declarations;
 - ▶ Performing the actual transformation.

Example: value parameters in detail

```
let value =  
  { ?x |  
    In(?x, variables) &  
    MayUseBeforeDefInSelection(?x) &  
    !( MayDefInSelection(?x) &  
      MayUseBeforeDefAfterSelection(?x) )  
  }
```

Most predicates have an elegant definition in JunGL. For instance,

```
let predicate MayUseBeforeDefAfterSelection(?x) =  
  [endNode]  
  (local ?z: cfsucc[?z] & ![?z]def[?x])+  
  [?u]use[?x]
```

holds if there is a path from the end node to a use of x with no intervening definition of x .

JunGL live (2)

- ▶ Refactoring demo:
 - ▶ *Rename Variable*
 - ▶ *Extract Method*

Future work

- ▶ Implement *Extract Interface* to cover a full spectrum of refactorings.
- ▶ Make JunGL workable on large programs using caching and incremental computation mechanisms, as well as BDDs for storing relations.
- ▶ Increase the confidence of JunGL users in their scripts, by:
 - ▶ providing a soft-typing system;
 - ▶ checking some properties of the transformations?

Thank you!

- ▶ Thanks also to:
 - ▶ Ran Ettinger and Oege de Moor
 - ▶ Microsoft Research for their support of the project
- ▶ More information available at
<http://progtools.comlab.ox.ac.uk/projects/jungl>