JunGL: a Scripting Language for Refactoring

Mathieu Verbaere, Ran Ettinger and Oege de Moor

Programming Tools Group
University of Oxford
United Kingdom
{Mathieu.Verbaere,Ran.Ettinger,Oege.de.Moor}@comlab.ox.ac.uk

ABSTRACT

Refactorings are behaviour-preserving program transformations, typically for improving the structure of existing code. A few of these transformations have been mechanised in interactive development environments. Many more refactorings have been proposed, and it would be desirable for programmers to script their own refactorings. Implementing such source-to-source transformations, however, is quite complex: even the most sophisticated development environments contain significant bugs in their refactoring tools.

We present a domain-specific language for refactoring, named JunGL. It manipulates a graph representation of the program: all information about the program, including ASTs for its compilation units, variable binding, control flow and so on is represented in a uniform graph format. The language is a hybrid of a functional language (in the style of ML) and a logic query language (akin to Datalog). JunGL furthermore has a notion of demand-driven evaluation for constructing computed information in the graph, such as control flow edges. Borrowing from earlier work on the specification of compiler optimisations, JunGL uses so-called 'path queries' to express dataflow properties.

We motivate the design of JunGL via a number of non-trivial refactorings, and describe its implementation on the .NET platform.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.3.4 [**Programming Languages**]: Processors; D.2.6 [**Software Engineering**]: Programming Environments

General Terms

Languages, Design

Keywords

Refactoring, scripting language, source code transformation, language workbenches

ICSE'06, May 20–28, 2006, Shanghai, China.

1. INTRODUCTION

A *refactoring* is a program transformation that improves the design of a program while preserving its behaviour. Often the purpose is to correct existing design flaws, to prepare a program for the introduction of new functionality, or to take advantage of a new programming language feature such as generic types. A remarkable number of refactorings have been proposed, and a wealth of examples can be found in *e.g.* [11, 12, 17].

Most refactorings require tedious, error-prone manipulation of the codebase, and it is therefore desirable to provide automated support for applying them. Indeed, most interactive development environments now provide such support, in the form of a fixed menu of transformations that may be applied, for instance for renaming, extracting a method, extracting an interface, and so on.

In view of the large number of refactorings that have been proposed, it is natural that developers wish to author their own refactorings. Support for doing that is however extremely rudimentary in existing systems. In Eclipse, it requires writing a new plugin, and mastery of a number of complex APIs. In IntelliJ IDEA, there is a facility called *Structural Search and Replace* that enables limited transformations by pattern matching on the syntax tree.

The reasons for this paucity of features to build new refactorings lies in the inherent complexity of implementing correct program transformations. To illustrate, consider the *Extract Method* refactoring, where the programmer selects a contiguous block of code, which is then extracted into a new method. The tool needs to determine what parameters are passed. Eclipse, IntelliJ IDEA and Visual Studio provide this refactoring, and all three implementations are incorrect.

An example of such a flaw in Visual Studio 2005 is shown in Figure 1. On the left is the original program, and the region to be extracted is indicated by the 'from' and 'to' comments. On the right is the resulting code: note that in the new method, the variable i is returned without necessarily being assigned. This kind of bug goes to the heart of the difficulty of implementing new refactorings: it requires dataflow analysis (in particular variable liveness), of the same kind as in compiler optimisations. From this and similar examples, we conclude that a framework for refactoring must provide dataflow analysis facilities as well as other, perhaps more obvious, features such as pattern matching, rewrite rules and mechanisms for variable binding. We shall show the correct way to refactor this example in Section 3. More examples of faulty refactorings are documented in [10].

```
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;
}
```

Figure 1: *Extract Method* bug in Visual Studio.

Over the past fifteen years, there has been an upsurge of activity in the formal specification of compiler optimisations, and in generating program transformers from such specifications *e.g.* [2, 6, 8, 13, 18–22, 28, 29, 34, 40]. All these works contrast with research that seeks to express transformations in purely syntactic terms, without recourse to dataflow analysis. As illustrated above, such an approach is not feasible when scripting refactoring transformations.

In this paper, we wish to suggest that the techniques which have proven successful in specifying compiler optimisations form an appropriate basis for scripting refactoring transformations – with the important difference that in refactoring one transforms source code, and not some convenient intermediate representation. The principal contributions of this paper are:

- The identification of the need for a scripting language for refactoring transformations.
- The formulation of requirements for such a language, in particular:
  - functional features (borrowed from ML, such as higher order functions and pattern matching) for manipulating ASTs and graphs more generally,
  - logical queries (akin to Datalog) for expressing complex relationships between program elements, and
  - path queries as a convenient shorthand for Datalog queries that capture, for instance, dataflow properties.
- The integration of all these features in a clean, coherent design.
- The validation of that design via a number of non-trivial examples.
- An implementation of the language on the .NET platform.

The remainder of this paper is organised as follows. In Section 2, we identify the requirements for a language to script refactorings, and introduce the basic ideas of our design. That design is then put to the test in Section 3, where we discuss a number of example refactorings, including *Extract Method*. We proceed to discuss the implementation in Section 4. As already indicated above, there is a great deal of related work, and we explore that in Section 5. Finally, we evaluate our results in Section 6, and we map out directions for future research.

2. JUNGL

Our scripting language for refactoring is called *JunGL* — short for *Jungle Graph Language*. JunGL is primarily a functional language in the tradition of ML. Like ML, it has important features such as pattern matching and higher-order functions, while allowing the use of updatable references. The advantages of this type of programming language in compiler-like tools is well-known [1]. To illustrate the style of definition, here is the 'map' function that applies a function f to all elements of a list l:

```
let rec Map(f,l) =
  match l with
  | [] → []
  | x :: xs → f(x) :: Map(f,xs) ;;
```

That is, *Map* is recursively defined: in the body, we examine whether l is empty or whether l consists of an element x followed by the remaining list xs. In the latter case, we apply f to x and recurse on xs.

Program graph.

The main data structure manipulated by functions is a graph that represents the program we wish to transform. Each node and edge in the graph can have a *kind*, indicated by a string. So for example, edges that indicate the control flow from one statement to another are labelled "cfsucc", for *Control Flow Successor*.

Lazy edges.

Initially the graph will just consist of a forest of ASTs for all the compilation units, where edges indicate children and parents. Additional computed information is then added via "lazy" edge definitions, which will only be evaluated when their value is required. To illustrate, we describe how some control flow edges may be added to the

graph. The following function in JunGL computes the control flow edges emanating from a conditional statement:

```
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] ;;
```

Here *DefaultCFSucc* is another user-defined function, which computes the control flow successors of ordinary statements such as assignments. Note how in the above definition, we can cope with missing (null) branches of the conditional in order to handle incomplete programs. We can now declare that *IfStmtCFSucc* is the way to add control flow edges to conditional statements:

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

The operation *AddLazyEdge* is a primitive of JunGL. Its effect is to add new edges labelled "cfsucc" from every node n labelled "IfStmt" to each node in *IfStmtCFSucc*(n). The edges will however only be constructed when we try to access the successor edges of n.

As we shall see later in the discussion of *Extract Method*, this mechanism of lazy edge construction is very convenient when introducing new graph nodes, as it often relieves us of the burden to laboriously construct all the auxiliary information on new nodes. Once again, all computed information on the AST is handled in this way, so for example edges that link a variable use to the declaration of that variable are also represented as lazy edges. In that instance, the function we pass to *AddLazyEdge* stipulates how one finds the declaration corresponding to a variable use. We shall further elaborate on this example when discussing the *Rename Variable* refactoring.

Streams. Demand-driven evaluation is very important in scripting refactorings: without it, scripts would quickly become prohibitively complex (because we have to remember to construct all relevant edges when creating new graph nodes), and also inefficient. Besides lazy edges, JunGL also provides *streams*, which are lazily evaluated lists. The main use of these is via predicates, which we discuss next. The use of streams often allows us to specify a search problem in a nice, compositional way: generate a stream of successes, and pick the first one — no further elements will be computed.

Predicates. Typically we manipulate the object program graph by running a number of queries to find out specific information, and then we make some destructive updates to the graph. A functional language is not very suitable for querying a graph structure; logic languages (in the Datalog tradition) are much better suited to that task. We therefore add the notion of *predicates* to our language, effectively making JunGL a hybrid functional-logic language.

When integrating a functional and a logic language, the key question is how we use predicates in functions, and vice versa. In JunGL, one can use predicates in functions via a stream comprehension.

```
{ ?x | P(?x) }
```

will return a stream of all x that satisfy the predicate P. Note that logical variables such as x are prefixed by a query mark to distinguish them from normal variable names. One can use expressions as arguments in predicates — but obviously all logical variables in such an expression must be bound.

Interestingly, we have not found it necessary to import the full power of a logic language such as Prolog, and in particular there is no use of unification in the implementation. Datalog (essentially Prolog minus data structures) provides just the right balance of expressive power with an efficient implementation. JunGL is somewhat akin to early attempts at integrating logic features into functional languages, such as LogLisp [32], but to our knowledge it is the first attempt at integrating a variant of Datalog rather than full logic programming in a functional language.

Path queries. The most common way of constructing predicates is via *path queries*: regular expressions that identify paths in the program graph. For example, here is a predicate that describes a path from a variable occurrence *var* to its declaration as a method parameter:

```
  [var]
  parent+
  [?m:Kind("MethodDecl")]
  child
  [?dec:Kind("ParamDecl")]
&
  ?dec.name == var.name
```

The path components between square brackets are conditions on nodes, whereas *parent* and *child* match edge labels. The above predicate thus says that we can reach a method declaration by following one or more *parent* edges from the *var* node. Furthermore, that method declaration (named *?m*) has a *child* that is a parameter declaration (named *?dec*). Finally, the last conjunct checks that the parameter's name coincides with the name of the variable node we started out with. The key is that the variable occurrence *var* is known, so the predicate above can be used to find instantiations of the logical variables *?m* and *?dec* so that the above predicate is true.

Predicates can be named just like functions, by using the keyword **predicate** in a **let** binding:

```
let predicate ParamDecl(?m,?dec) = ... ;;
```

3. REFACTORING EXAMPLES

We illustrate the design of JunGL by implementing two of the most frequently used refactorings: Rename Variable and Extract Method. JunGL allows us to manipulate any program graphs and add all semantics information required for the transformations by defining lazy edges.

Our object language will be a modest but non-trivial subset of the $C^{\#}$ language [33]. Notably, we ignore two major features of the object-oriented paradigm, inheritance and visibility, since they do not present any particular difficulty in the mechanisation of our example refactorings. We also leave aside arrays, exceptions, generics, structs, enums, delegates, open-ended namespaces and adapt the full specifications of $C^{\#}$ to our subset. Thus, the JunGL code below for computing additional information about the program is merely an illustration: we do not claim to express here the complete variable binding rules of $C^{\#}$, but we do wish to convey a taste of how one can fully accomplish it.

3.1 Rename Variable

The automation of *Rename Variable* is far beyond a simple search-and-replace mechanism, because it requires variable binding information and the ability to detect potential conflicting declarations of variables with a similar name. For that reason, we first briefly discuss how variable bindings are manipulated in JunGL.

Variable binding. In JunGL, we define a function that looks for the first declaration match of a given name starting from a specific point in the program, by means of carefully ordered disjunctions of path queries:

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

Here the path queries follow three different kinds of edges: *child* and *parent* are the only built-in edges of the graph; *pred* is defined as the left sibling of a node if non null and as its parent otherwise. The function *FindFirst* is simply a convenient primitive that yields the first element of a stream. Note that the order of the three alternatives is significant: the stream will first have results for locals, then for parameters, and then for fields.

The lazy edge for simple name lookup can now be defined as follows:

```
let SimpleNameLookup(ref) =
  FindFirstDeclarationMatch(ref,ref.name) ;;
AddLazyEdge("SimpleName","lookup",
            SimpleNameLookup) ;;
```

As a first illustration of the use of the *lookup* edges, assume we want the *Rename Variable* refactoring to be applicable either on a variable reference or on a declaration. The skeleton JunGL code would then be as follows:

```
let RenameVariable(var,newName) =
  let dec = FindFirst({ ?dec |
              [var]lookup[?dec:VariableDecl()]
            | [?dec:Equals(var)] }) in
  if !Is(dec,"VariableDecl") then
    Error("Please select a variable");
  if dec.name != newName then
  begin
    // ... Further checks...
    // ... Transformation...
  end ;;
```

We look up the declaration of *var*, or assume the node *var* to be the declaration itself if it has no outgoing *lookup* edge. Then, if the desired new name actually differs from the previous one, we may perform further checks and eventually perform the transformation.

Conflicting declarations. To understand more precisely the intricacies of renaming, let us consider the following code:

```
class A
{
  int i;
  public int getI()
  {
    int j = 0;
    return i;
  }
}
```

One may want to rename the local variable j to i, although the instance member i is used in the same context. In Eclipse or Visual Studio, post-transformation checks are performed to ensure that variable bindings have not changed, and in particular no inadvertant variable capture occurred. In the example, the transformation would be rejected a posteriori — in Visual Studio, after the tool has offered a view of how the transformation applied.

IntelliJ IDEA 5.0 however seems to overcome the problem of conflicting declarations beforehand and would normally cope with variable hiding. Yet, the above refactoring results, without any warning, in code where the occurrence of j has simply been changed to i. The code still compiles but i in the second statement of the method is no longer bound to the instance member, but to the freshly renamed local variable [1]. This situation is certainly the worst in a refactoring process since your code remains compilable, but now has a different meaning. Using JunGL, we correctly remove the ambiguity by changing i in the return statement to $this.i$ in order to refer to the instance member, even in the presence of a new local variable i.

Let us detail the JunGL code we omitted before in the *RenameVariable* function, that deals with potential conflicts. We first check that there is no former declaration of a variable that would inevitably conflict with our renamed variable declaration. By inevitable conflict, we mean that variable hiding is not even allowed. Here is the part of the function to check for conflicts:

```
let conflictFormerDec =
  FindFirstDeclarationMatch(dec,newName) in
if conflictFormerDec != null then
begin
  if !IsHidingAllowed(dec,conflictFormerDec) then
    Error(newName + " already exists")
end;
```

If we ignore type references and other members than fields, the function *IsHidingAllowed* can simply be defined as:

```
let IsHidingAllowed(x,y) =
  !IsField(x) &&IsField(y) ;;
```

We then need to check all other declarations which may conflict with our renamed variable. At this point, it is convenient to call another user-defined function *Scope* that returns the scope of a declaration. *Scope* is easy to define with JunGL, even in the presence of derived classes, although we have chosen to ignore these here for expository reasons. For the scoping rules of our $C^{\#}$ subset, we also need a predicate $InCurrentOrLocalParentScope(?x, ?s)$ that holds whenever the element x is in s or a local parent scope of s if s is a nested block of a method. The code then reads:

[1] JetBrains informed us this bug is now fixed in IntelliJ IDEA 5.1.

```
let scope = Scope(dec) in
let predicate InCurrentOrLocalParentScope(?x,?s) =
      [?s] child* [?x]
    | [?s] parent*[?b:Block()]
      (local ?z: child [?z :! Block()])*
      [?x] in
let conflictDecs = { ?otherDec |
      InCurrentOrLocalParentScope(?otherDec,scope)
   & VariableDecl(?otherDec)
   & ?otherDec.name == newName } in
foreach otherDec in conflictDecs do
begin
   if !IsHidingAllowed(otherDec,dec) then
      Error(newName +" already exists")
end;
```

Transforming. Now that we have checked that there are no conflicting declarations that would require us to reject the refactoring, we can proceed with the transformation. If there exists a field declaration that would be hidden by the newly renamed variable, we need to remove the ambiguity of all its references by introducing an explicit *this* target or a reference to its declaring type in the case of a static field. We define the function *RemoveAmbiguity* as follows:

```
let RemoveAmbiguity(ref,dec) =
   if ref.kind == "SimpleName" then
   begin
      let access = CreateNode("MemberAccess") in
      if IsStaticField (dec) then
         access.target ← CreateReference(DeclaringType(dec))
      else
         access.target ← CreateNode("ThisRef");
      access.name ← ref.name;
      Replace(ref, access)
   end ;;
```

CreateNode and *Replace* are primitive operations in JunGL. Then, in the *RenameVariable* function, we add:

```
if conflictFormerDec != null then
begin
   let needDisambiguity = { ?ref |
         InCurrentOrLocalParentScope(?ref,scope)
      & [?ref]lookup[conflictFormerDec] } in
   foreach ref in needDisambiguity do
      RemoveAmbiguity(ref,conflictFormerDec)
end;
```

Note how we use the *lookup* edges in reverse here, to find all references to a declaration of interest.

Next, we find all references *needRename* of the variable to be renamed. If it happens that this variable is hidden at some points of the program due to nested declarations with the same name, we also remove the ambiguity in addition to the proper renaming process. Last, we update the name of the variable in the declaration itself:

```
let needRename = { ?ref | [?ref]lookup[dec] } in
let ambiguityScopes = { Scope(?dec) |
      In(?dec, conflictDecs ) } in
let needDisambiguityRename = { ?ref |
      In(?ref ,needRename)
   & In(?scope,ambiguityScopes)
   & InCurrentOrLocalParentScope(?ref,?scope) } in
foreach ref in needRename do
   ref.name ← newName;
foreach ref in needDisambiguityRename do
   RemoveAmbiguity(ref,dec);
dec.name ← newName;
```

We have exposed almost all the JunGL code required for the mechanisation of *Rename Variable* with careful checks and minimal rejection. Of course, one could argue that the way we deal with variable hiding is undesirable because the resulting code might be sometimes less readable. In our view this objection comes more under coding style and best practices, and such concerns could also be checked with JunGL.

At this point some readers may be concerned about the efficiency of our variable binding mechanism. It would be obviously much more efficient to compute bindings in a single pass, like in classical compiler construction. Nevertheless, it is very convenient to declaratively specify the binding rules like we did, by translating the specifications of the language to concise predicates. Our prototype implementation is quite workable as it stands, and yet vast improvements are possible, for instance by selectively caching the results of the lazy edge evaluation, or by specifying additional edges for storing binding information in intermediate nodes such as blocks.

3.2 Extract Method

Let us now return to the *Extract Method* refactoring mentioned in the introduction. Here the user selects a region of code. Below we shall identify that region by two nodes (statements) in the graph, namely the start of the region to be extracted, and the end of the region to be extracted. There are four major phases in the implementation of this refactoring, and we shall consider each in turn: checking validity of the selection, determining what parameters must be passed, where declarations should be moved, and finally doing the transformation itself.

Checking validity. The refactoring will first need to check that it is a valid selection: for instance, one can only extract a block of code into a method if it is single-entry single-exit. These are the usual conditions: the start node dominates the end node, the end node post-dominates the start node, and the set of cycles containing the start node is equal to the set of cycles containing the end node. These conditions are easily expressed in terms of path patterns. For example, here is the definition of *Dominates*:

```
let Dominates(entryNode,startNode,endNode) =
   IsEmpty({ () | [entryNode]
                  ( local ?z: cfsucc [?z] &
                              ?z != startNode )*
                  [endNode] }) ;;
```

It takes three parameters: the entry node of the method that contains the block, the start node of the block, and the end node of the block. By definition, the start node dominates the end node if all paths from the entry node to the end node pass through the start node. To express that in our language, first note that the predicate

```
[entryNode]
(local ?z: cfsucc [?z] & ?z != startNode)*
[endNode]
```

signifies a path all of whose elements are *not* equal to the start node. We therefore require that no such path exists, by testing that the above set is empty.

Other checks, simpler ones, are required. The control flow graph lacks indeed some scoping information, and therefore, we also need to check that the selection doesn't straddle different scopes.

Parameters. When we have verified that the selection is indeed amenable to method extraction, our next task is to determine what the parameters of the method should be, and what results must be returned. We shall consider three different sets of parameters, namely those passed by value, those passed by reference, and the output parameters whose only function is to return a result.

A variable x in *variables*, the set of local variables or parameters that may be used or defined in the selection, will become a value parameter if the following conditions are satisfied:

- x is *live* upon entry in the extracted block, that is it may be used in the selection, and it is not redefined before it is used. The condition that x may be used is obvious; if x is always redefined before such a use, there is no need to pass it as a parameter, as its value can be computed locally in the extracted method.
- It is *not* the case that x may both be redefined in the selection, and it may be used before it is redefined after the selection. If x is *live* at the end of the selection, but not redefined in the selection, it is fine to pass it by value.

We can thus compute the set of value parameters as follows:

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

Each of the predicates used here has an elegant definition in JunGL. To illustrate, consider *MayUseBeforeDefAfterSelection(?x)*. This predicate holds if there is a path from the end node to a use of x with no intervening definition of x. A node u uses x if it has a user-defined lazy edge labelled *use* to x. Similarly, an intervening node z does not define x if it has no lazy edge labelled *def* to x.

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

Note that this definition can also deal with the possibility for a 'use' outside the method where the extraction occurs, namely when x is a non-value parameter. Indeed, *use* lazy edges are defined from the exit node of the method to all non-value parameters.

We now consider when a variable x should become an output parameter of the extracted method. Here the specification consists of three conjuncts:

- First, there exists a potential use without prior definition of the variable x after the selected statements: without such a potential use, there is no point in returning x as a result of the method.
- Second, there should be no use of x before a definition of x in the selection itself. If there was such a use, it would not be sufficient to pass x merely as an output parameter: its initial value is important too.
- Third, x must actually be defined in the selection. If it were not, then the result of the refactoring would not be compilable, because $C^{\#}$ requires all output variables to be definitely assigned.

In summary, we can define the set of output parameters as follows:

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

Again, the definitions of these predicates are all straightforward in JunGL, and we do not spell out the details.

At this point, we have precisely defined what should be the value and output parameters of the extracted method. It remains to define the reference parameters. At first glance, one might say that any variable in the selected block that is not a value or output parameter is a reference parameter. Such a criterion would however be much too crude. Some variables will just be local to the selection, and such variables do not need to be passed as parameters at all. They will become local variables of the extracted method body. A more accurate definition of the set of reference parameters is therefore as follows:

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

That is, x may be used before it is redefined in the selection or it is only potentially defined in the selection, x may be used before it is redefined after the selection, and it is not a value or output parameter.

It is interesting to work out the effect of these definitions on an example such as Figure 1. Clearly b is classified as a value parameter. But what about i? As explained in the introduction, the bug in Visual Studio was that i became an output parameter (and being the only such parameter, in fact the method result). In our definition, that is prevented by the final conjunct in the definition of *out* because we have

```
!MustDefInSelection(i)
```

Note that we also don't get i as a value parameter because there is a definition before its use in the selection. Finally, it does not become a reference parameter because it is defined before being used after the selection. We conclude that according to our definition, i does not become a parameter at all.

Placing declarations. Having decided on the parameters of the extracted method, we now turn to placing declarations for its local variables. In doing so, we consider three cases: declarations that must be moved out of the selection, declarations that must be moved into the selection, and finally those that need to be duplicated. We discuss each of these in turn.

A declaration needs to be moved out of the selected block if it is declared there, and if it is used or defined outside the selection:

```
let needDecMoveOut =
  { ?x |
    DecInSelection(?x) &
    MayUseOrDefOutOfSelection(?x)
  }
```

Conversely, if a declaration doesn't occur in the selected block, it is defined or used in that block, and it is not a parameter, then the declaration should be moved into the extracted method's body:

```
let needDecMoveIn =
  { ?x |
    In(?x, variables ) &
    !DecInSelection(?x) &
    !In(?x,value) &
    !In(?x,out) &
    !In(?x, ref )
  }
```

Finally, there are the declarations that must be duplicated. This can happen because the use of a variable in the selection is in fact independent of the use of the variable outside the selection: effectively, we can split the variable into two independent ones. The declarations in question are defined by

```
let needDecDuplication =
  { ?x |
    In(?x,needDecMoveIn) &
    MayUseOrDefOutOfSelection(?x)
  }
```

To wit, the declaration of x needs to be moved into the existing declaration (as we have just defined it), but there are also uses and/or definitions outside the selection.

Again, let us return to Figure 1 and see what happens to the variable i. Because it is not a parameter of any kind, but it occurs in the selection and it is not declared in the selection, i will be a member of *needDecMoveIn*. However, note that because it also occurs after the selection, it will in fact be classified as a declaration that needs duplication: the two uses of i, inside and outside the selection, have been correctly separated.

Transforming. Armed with all the necessary information, we can now actually perform the required transformation of creating a new method. This is, in fact, the least interesting part of the code: all that needs to be done is to reconstruct the relevant portions of the graph.

As a small example fragment, consider the operation of inserting a new statement before an existing one:

```
let InsertStatementBefore(n,s) =
  if IsInList (n) then
    InsertBefore(n,s)
  else
    let block = CreateNode("BlockStmt") in
    Replace(n,block);
    block.statements ← [s;n] ;;
```

First we check whether n is itself in fact part of a sequence in the AST. If so, we simply add s as the left-hand sibling of n. If not, however, we first need to create a new block statement, which replaces n in the AST; both s and n become descendants of this new block statement. Note that it is *not* necessary to define control flow edges (*cfsucc*) on the new block statement, because we defined these to be lazy, so they will be automatically constructed when necessary. Like *CreateNode* and *Replace*, the functions *IsInList* and *InsertBefore* are also primitive operations in JunGL.

We return once again to the example of Figure 1. Figure 2 shows the result of applying this refactoring in our own tool. Note that at present, we do not detect that the selected block did not contain any instance references, so as yet we do not make it static — it would however be easy to add that improvement.

The complete specification for *Extract Method* is only 135 lines in JunGL, so about two A4 pages. We find this very encouraging, and a strong indication that even complex refactorings can be concisely defined.

In the exposition above we have assumed that the original program compiles without errors. Of course in practice it is very common to apply refactorings to programs that cannot be compiled for subtle reasons such as the definite assignment rule of $C^{\#}$ (which states that every local must be initialised before it is used). In such cases, the refactoring should preserve the compilation errors in the result of the transformation. We have found it easy to amend the JunGL code to transform such slightly faulty input programs.

Of course we are not the first to attempt a precise account of extract method. Indeed, Martin Fowler's book [11] contains quite detailed recipes. However, the big advantage of having a precise, formal description such as the parameter definitions above is that they provide a sound basis for rigorous reasoning. For example, an important property is that no variable will be classified as two different kinds of parameter. It is easy to check that this requirement is indeed satisfied, using the above definitions. Another desirable property, which is again quite easy to check, is that no variable use will become orphaned, with no declaration to match it.

3.3 Further Examples

The above example refactorings were chosen for expository reasons, and space restrictions prevent us from including further examples. Yet, an exciting experiment that we have done is to code the *Untangling* refactoring we proposed earlier in [9]. It is like *Extract Method*, but instead of selecting a contiguous region of code, the programmer selects a single expression; the tool then extracts the backward slice (statements that may have contributed to the value of that expression) [39]. Slicing can be expressed quite elegantly in JunGL. More generally, one can define the Program Dependence Graph [15] via path queries. That in turn allows the correct mechanisation of many different transformations that require reordering of statements.

There are other refactorings that alter the type structure of a program, by extracting an interface from a class [36], introducing generic types [5, 38], or supporting class library migration [3]. These refactorings involve an analysis of the class hierarchy and usually require solving type constraints. The next step in the validation of our language design is to code these advanced refactorings. Although additional data structures can be handled in JunGL, it is likely that efficient constraint solving will only be achieved by providing a solver as a built-in feature of JunGL.

4. IMPLEMENTATION

JunGL is implemented on the .NET platform. The system consists of three components: a graph data structure, an interpreter for the scripts that manipulate this data structure, and a structure editor (for the object language, not the scripts) that facilitates interactive development of scripts.

```
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;
  NewMethod(b);
  i = 1;
  Console.WriteLine(i);
}

private void NewMethod(bool b)
{
  int i;
  if (b)
  {
    i = 0;
    Console.WriteLine(i);
  }
}
```

Figure 2: Correct refactoring of Figure 1.

Currently, JunGL does not fully deal with user-interface issues. User inputs are passed to the refactoring scripts as parameters. Yet, for a rich interactive experience, refactoring tools commonly guide users through 'wizards' and we plan to soon support these via calls to external code.

Jungle graph. JunGL manipulates a graph through basic operations defined in a small interface. We provide a default implementation of this interface in $C^{\#}$. Furthermore, we provide a facility for defining the *type graph* that stipulates certain syntactic requirements on the links that are allowed (say, the fact that an if-statement can only have three children, the first an expression and the others statements). These constraints are enforced through run-time checks. Given the default graph implementation and the type graph definition, a parser is required to build the AST as a particular jungle graph. To work with a different object language, one simply provides another type graph definition, along with the new parser. Furthermore, the architecture makes it easy to leverage an existing strong-typed AST implementation. All one needs to do is to make the AST classes implement our jungle graph interface.

As we explained in Section 2, once the AST has been represented as the initial jungle graph, this can be further decorated by connecting nodes via lazy edges. For now we have found that a naive implementation yields adequate efficiency, but obviously in a production system one would need to make careful use of caching. We are confident that the same techniques that have proved effective in the demand-driven evaluation of rewritable reference attribute grammars [7] will also apply here.

Interpreter. The JunGL interpreter follows the usual pattern of an interpreter for a functional language in a functional language. Indeed, JunGL is implemented in $F^{\#}$ [35], a variant of ML that runs on top of the .NET framework. Because $F^{\#}$ is fully integrated in .NET it allows us to work across languages. In particular, we can use the $C^{\#}$ implementation of the jungle graph in our $F^{\#}$ programs and vice versa. For now, JunGL does not support user-defined types and, like most other scripting languages, it is only dynamically typed.

There exists a simple translation from path queries to Datalog, and we use the same scheme as in *e.g.* [24]. The most interesting part of the interpreter, therefore, is its treatment of Datalog queries. Here we employ a technique originally due to Mycroft and Jones, who were the first to model the operational semantics of logic programs in terms of streams [16]. As said earlier, however, it is key to our design that we do not require the full power of Prolog; just Datalog queries, augmented with expression evaluation, suffices. The $C\#$ 2.0 coroutines feature makes the implementation of the pipeline pattern of streams, and therefore of Datalog queries, very neat. We support all operations of the relational algebra, plus transitive closure and filtering with expression evaluation.

We are hopeful that in future, certain analyses that are conveniently expressed via Datalog queries can be given a more efficient implementation, in particular via BDDs. A good starting point for such work would be the generation of efficient analyses from relational queries in [23].

Structure editor. In addition to the interpreter itself, we have implemented a structure editor to visualize and edit, in limited ways, the graph that is manipulated by JunGL. The editor uses some pretty-printing annotations of the type graph definition to render the AST. By selecting blocks or nodes, we can visualize the connections to other nodes in the graph, that we have added by defining lazy edges in JunGL. We have found this tool indispensable in the interactive development of new refactoring scripts.

5. RELATED WORK

Rigorous refactoring. We are by no means the first to realise the need for a formal, precise approach to refactoring. In their PhD theses, Opdyke then Roberts insisted on the importance of preconditions and postconditions for refactoring transformations [30,31]. More recently, several works that focus on refactorings that change the type structure of a program precisely define how to automate these refactorings [3, 5, 36, 38]. While there is no attempt to define a scripting language for expressing them, we share their aim of pinning down the relevant transformations that were previously only loosely defined.

Optimisation specifications. In the introduction, we already indicated that the design of JunGL heavily borrows from the literature on declarative specifications of compiler optimisations. In particular, our use of path queries can be traced back to the design of Gospel by Whitfield and Soffa [40]. Gospel has a similar feature, but the dataflow facts are hard-coded in the implementation, whereas in JunGL they are user-definable. The idea to achieve that flexibility via a form of logic programming augmented with path expressions originated in our own work [6, 20]. A separate branch of research, instigated by Lacey, is the formal verification of compiler optimisations that are specified in this style [19]. Indeed, Lerner *et al.* have demonstrated how to automate such proofs [21, 22].

A completely different approach to scripting compiler optimisations was proposed by Olmos and Visser in [29]. There, the optimisations are rewrites of the syntax tree. Dataflow information is obtained by dynamically introducing new rewrite rules. As a consequence, the specification of dataflow facts is much less declarative than in JunGL, but for the description of the transformation itself (the change to the object program) the situation is reversed. We are currently investigating the use of Stratego's rewriting primitives to streamline that part of our specifications.

Graph transformations for refactoring. The idea of declarative specifications of refactorings via graph transformations was first put forward by Tom Mens in [26]. The refactorings considered there are renaming and numerous variants of moving class members. Their specification is purely declarative, as a graph rewrite system. A big advantage of using graph rewrite systems is that it becomes possible, for example, to detect conflicting refactorings [27].

The main difference with our work is that none of the refactorings require dataflow analysis. It would be interesting to see whether Mens's techniques scale up to the full-blown refactorings of [36].

An earlier attempt to use graph rewrite systems for specifying program transformations was the Optimix system by Aßmann [2]. Optimix does not have a mechanism for directly expressing properties of program paths, however.

Path queries. The idea of path queries in the context of program transformations is due to ourselves [6, 28, 34]. For the particular version used here, we drew inspiration from the syntax in [24], which followed on from our own design in the above citations. A similar style of queries is of course very common in the literature on semi-structured databases *e.g.* [4].

6. CONCLUSION

In this paper, we identified the need for a scripting language to author new refactorings. New refactorings are proposed all the time, and yet even very basic examples like *Rename Variable* or *Extract Method* are incorrectly implemented in leading development environments.

We also exposed the requirements for such a scripting language: easy manipulation of graph structures, demand-driven evaluation under the programmer's control, a query language for concise expression of program analysis problems (in particular through the use of *path queries*). We proposed a concrete design for such a language, named JunGL. One missing feature that we intend to add in the near future is quotation for object programs [37].

The design of JunGL was validated through a number of non-trivial refactoring scripts on a substantial subset of the $C^{\#}$ language. In particular we demonstrated how some bugs in IntelliJ IDEA and Visual Studio are easily discussed (and avoided) by expressing the refactoring in JunGL.

We then presented the implementation of JunGL on the .NET platform. A notable feature is the use of streams to achieve the desired integration of logic elements into the predominantly functional nature of JunGL. There is much scope for further work here, however, for instance by implementing certain program analysis queries via BDDs [23, 25], and others (that manipulate very large portions of the code base) via database queries [14].

Acknowledgements. We would like to thank Microsoft Research, and in particular Dr. Fabien Petitcolas, for their generous support of this project. Bill Gates provided helpful feedback at the planning stage, by emphasising the importance of a *scriptable* refactoring engine. Members of the Programming Tools Group at Oxford, in particular Damien Sereni, provided helpful advice and inspiration throughout.

7. REFERENCES

[1] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[2] Uwe Aßmann. OPTIMIX — a tool for rewriting and optimizing programs. In H. Ehrig, G. Engels, H. J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation. Volume 2: Applications, Languages and Tools*, pages 307–318. World Scientific, 1998.

[3] Ittai Balaban, Frank Tip, and Robert Fuhrer. Refactoring support for class library migration. In *Proceedings of the 20th ACM conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 265–279, 2005.

[4] Peter Buneman, Mary Fernandez, and Dan Suciu. UnQL: A query language and algebra for semistructured data based on structural recursion. *VLDB Journal*, 9(1):76–110, 2000.

[5] Alan Donovan, Adam Kiezun, Matthew S. Tschantz, and Michael D. Ernst. Converting java programs to use generic libraries. In *Proceedings of the 19th ACM conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 15–34, 2004.

[6] Stephen J. Drape, Oege de Moor, and Ganesh Sittampalam. Transforming the .NET intermediate language using path logic programming. In *Principles and Practice of Declarative Programming*, pages 133–144, 2002.

[7] Torbjörn Ekman and Görel Hedin. Rewritable reference attributed grammars. In Martin Odersky, editor, *European Conference on Object-Oriented Programming*, pages 144–169, 2004.

[8] Dawson R. Engler. Interface compilation: Steps toward compiling program interfaces as languages. *IEEE Transactions on Software Engineering*, 25(3):387–400, 1999.

[9] Ran Ettinger and Mathieu Verbaere. Untangling: a slice extraction refactoring. In Gail C. Murphy and

Karl J. Lieberherr, editors, *Aspect-Oriented Software Development*, pages 93–101, 2004.

[10] Ran Ettinger and Mathieu Verbaere. Refactoring bugs in Eclipse, IntelliJ IDEA and Visual Studio. `http://progtools.comlab.ox.ac.uk/projects/refactoring/bugreports`, 2005.

[11] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[12] Martin Fowler. Refactoring home page. `http://www.refactoring.com`, 2005.

[13] Sam Z. Guyer and Calvin Lin. An annotation language for optimizing software libraries. In *Second conference on Domain-Specific Languages*, pages 39–52. USENIX, 1999.

[14] Elnar Hajiyev, Mathieu Verbaere, Oege de Moor, and Kris de Volder. Codequest with datalog. In *OOPSLA Companion*, 2005.

[15] Susan Horwitz and Thomas Reps. The use of program dependence graphs in software engineering. In *International Conference on Software Engineering*, pages 392–411, 1992.

[16] Neil D. Jones and Alan Mycroft. Stepwise development of operational and denotational semantics for prolog. In *Symposium on Logic Programming*, pages 281–288, 1984.

[17] Joshua Kerievsky. *Refactoring to Patterns*. Addison Wesley, 2005.

[18] Marion Klein, Jens Knoop, Dirk Koschützki, and Bernhard Steffen. DFA & OPT-METAFrame: a toolkit for program analysis and optimization. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 418–421. Springer, 1996.

[19] David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *Proceedings of the 29th ACM symposium on Principles of Programming Languages*, pages 283–294, 2002.

[20] David Lacey and Oege de Moor. Imperative program transformation by rewriting. In R. Wilhelm, editor, *Proceedings of the 10th International Conference on Compiler Construction*, volume 2027 of *Lecture Notes in Computer Science*, pages 52–68. Springer Verlag, 2001.

[21] Sorin Lerner, Todd Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. In *Programming Language Design and Implementation*, pages 220–231, 2003.

[22] Sorin Lerner, Todd Millstein, Erika Rice, and Craig Chambers. Automated soundness proofs for dataflow and analyses via local rules. In *Proceedings of the 32nd ACM symposium on Principles of Programming Languages*, pages 364–377, 2005.

[23] Ondrej Lhotak and Laurie Hendren. Jedd: A BDD-based relational extension of java. In *Programming Language Design and Implementation*, pages 158–169, 2004.

[24] Yanhong Annie Liu and Scott D. Stoller. Querying complex graphs. In P. Van Hentenryck, editor, *Proceedings of the 8th International Symposium on Practical Aspects of Declarative Languages*, pages 16–30, 2006.

[25] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: a program query language. In *Proceedings of the 20th ACM conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 365–383, 2005.

[26] Tom Mens, Serge Demeyer, and Dirk Janssens. Formalising behaviour preserving program transformations. In *Graph Transformation*, volume 2505 of *Lecture Notes in Computer Science*, pages 286–301, 2002.

[27] Tom Mens, Gabriele Taentzer, and Olga Runge. Detecting structural refactoring conflicts using critical pair analysis. *Electronic Notes in Theoretical Computer Science*, 127(3):113–128, 2005.

[28] Oege de Moor, David Lacey, and Eric Van Wyk. Universal regular path queries. *Higher-order and symbolic computation*, 16(1-2):15–35, 2003.

[29] Karina Olmos and Eelco Visser. Strategies for source-to-source constant propagation. In B. Gramlich and S. Lucas, editors, *Workshop on Reduction Strategies in Rewriting and Programming*, volume 70 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, May 2002.

[30] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[31] Donald F. Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[32] J. Alan Robinson and Ernest E. Sibert. LOGLISP: Motivation, design and implementation. In K. L. Clark and S. A. Tänlund, editors, *Logic Programming*, pages 299–313. Academic Press, 1982.

[33] Peter Sestoft and Henrik I. Hansen. *$C^{\#}$ Precisely*. MIT Press, 2004.

[34] Ganesh Sittampalam, Oege de Moor, and Ken Friis Larsen. Incremental execution of transformation specifications. In *Proceedings of the 31st ACM symposium on Principles of Programming Languages*, pages 26–38, 2004.

[35] Don Syme. *$F^{\#}$*home page. `http://research.microsoft.com/projects/ilx/fsharp.aspx`, 2005.

[36] Frank Tip, Adam Kiezun, and Dirk Bäumer. Refactoring for generalization using type constraints. In *Proceedings of the 18th ACM conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 13–26, 2003.

[37] Eelco Visser. Meta-programming with concrete object syntax. In *Generative programming and component engineering*, pages 299–315, 2002.

[38] Daniel von Dincklage and Amer Diwan. Converting java classes to use generics. In *Proceedings of the 19th ACM conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 1–14, 2004.

[39] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10:352–357, 1984.

[40] Deborah Whitfield and Mary Lou Soffa. An approach for exploring code-improving transformations. *ACM Transactions on Programming Languages and Systems*, 19(6):1053–1084, 1997.