Using PROGRES for graph-based program refactoring

Erhard Weinell

RWTH Aachen University of Technology
Department of Computer Science 3 (Software Engineering)
Ahornstraße 55, D-52074 Aachen, Germany
Weinell@cs.rwth-aachen.de
http://se.rwth-aachen.de

1 Introduction

Applying the graph rewriting language PROGRES [1] to model program refactoring is a a natural choice, as it already been successfully applied for re-engineering. In [2], complex legacy telecommunication systems were analyzed and re-designed using the PROGRES toolchain. Similarly, we combine PROGRES' high-level language features such as visual graph transformation rules, functional expressions, and means for rule composition to yield an executable prototypical refactoring tool. The source code generated from this specification is combined with a framework for visual prototyping, which allows direct inspection of the basic graph structure. Furthermore, the prototype extends the framework's functionality to provide GXL import/export functionality according to the predefined schema.

This paper is structured as follows: The created specification is introduced in Section 2, followed by a short presentation of the generated refactoring tool in Section 3. Section 4 discusses experiences made during solving the tool contest task in response to the case studies' authors. In the appendix, Section A provides informations how to retrieve and evaluate the refactoring tool and the PROGRES specification. For the reader's convenience, Section B holds the entire PROGRES specification relevant to the case study solution.

2 Specification

The developed PROGRES specification is roughly composed of four parts, i.e. sections of the specification: First, the program graph meta-model is reproduced in form of a PROGRES graph schema. Second, additional helper functionality is added, e.g. commonly used path expressions. Third, refactoring rules are modeled using several transformation rules, with an "outer rule" to control their respective execution. Fourth, rules to allow users to build up or modify program graphs are provided for testing purposes. In the following, observations that were made during the development of the respective parts are discussed.

Meta-model: The meta-model of the given case study could be mapped to PROGRES in an almost direct way. One disadvantaged we encountered is the fact that PROGRES requires edge type identifiers to be unique. Therefore, numerous associations of the meta-model were renamed, e.g. belongsTo between Variable and Class to vBelongsToClass. However, it could be argued whether massive overloading of identifiers leads to an easily understandable meta-model. Furthermore, PROGRES does not support ordered edge types in general, so we simulated this feature simply by adding an integer attribute to the association's respective *target* class.

Helper functionality mainly comprises path expressions which are re-used in the refactoring rules. Among others, paths expressions are provided to determine the Class where an Expression is defined in, or to determine a MethodBody's references to this.

Encapsulate Field Refactoring. This refactoring is implemented by a total of five rules, i.e. to declare getters and setters, replace accesses and updates of the encapsulated fields, and a single controller rule. The latter also checks for previously existing getters and setters, creating new ones if none can be found. In the generated prototype, the controller rule is made available to the user

Move Method Refactoring. To implement this refactoring, seventeen transformation rules were required. This refactoring is inherently complex as it requires the selection of an appropriate increment to move the selected method body along, i.e. either a Variable or a Parameter. As this selection is not part of the given interface, and non-deterministic choice can hardly be considered a good solution, transformation processing has to allow user interaction. Basically, graph transformation rules modeled in PROGRES can access native code to add such functionality, Java in this case. However, requiring the developer to add UI code manually, and to embed this code into the transformation rules can hardly be considered as model-based development. We therefore followed a different approach by adding special marker nodes to the host graph in order to suggest possible continuations of the refactoring. From the seventeen rules, two are therefore required to find appropriate points of application and to add corresponding marker elements. Among the rest, there is again one controller rule, three rules to construct delegate methods, five to handle this references in the moved methods (including parameter addition and invocation-rewrite), two for call-rewriting in the non-delegation case, and some minor clean-up rules.

Pull-up Method Refactoring. This comparatively simple refactoring case is composed of eight rules. Again, a controller rule is added for checking preconditions and invoking sub-rules. The latter are responsible for moving the pulled-up method body, removing other bodies of the same operation, adding stub methods and abstract declarations of the pulled-up method. This refactoring case does not require any user interaction.

Constructive rules allow user-interactive manipulation of program graphs. Whilst PROGRES ensures schema-conformant results of all rule applications, users can nevertheless violate constraints which are not encoded in the schema, e.g. disjointness of a Variable's belongsTo edges.

Example rule. Figure 1 depicts a moderately complex transformation rule, whilst making use of some more "advanced" PROGRES functionality. Its intention is to rewrite calls to a method being moved along one of its parameters, given that the call has no explicitly stated receiver. A previously invoked transformation rule will already have rewritten the call's receiver for being passed as actual parameter, if required by the moved method's body.

The transformation rule's header defines two formal parameters, expecting the method declaration, and the parameter being moved along. The rule is applied for *all matches* in pseudo-parallel mode, as indicated by the star.

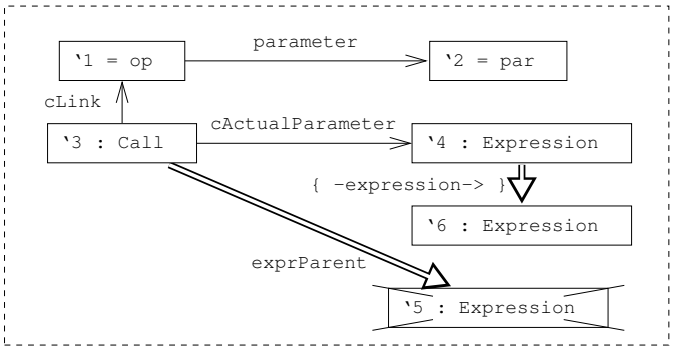
As defined by the rule's left-hand side (LHS), values of the formal parameters are bound to nodes `'1` and `'2`, respectively. Furthermore, a `Call` (`'3`) to the moved method is retrieved from the host graph, as well as the actual parameter (`'4`) with the same ordering index as the passed parameter, stated by the attribute condition below. The chain of `expression` edges emerging from the actual parameter is traversed *as long as possible*, until the end of this chain is reached in node `'6`. Considering the case where an actual parameter comprises only a single node, the `folding` clause allows non-isomorphic binding of rule nodes. Finally, the LHS ensures that no "expression parent" exists for the given call by referring to the `exprParent` path expression and a negated target node `'5`. This prohibits the existence of a receiver for this call (e.g. calls and accesses), though it does not deny the existence of other expression elements such as blocks.

If any occurrence of the rule's LHS is found, the matched sub-graph is transformed according to the right-hand side (RHS), and the textual transformation descriptions. According to the RHS, the `cActualParameter` edge between `'3` and `'4` is removed, whereas an `expression` edge is added between `6'` and `3'`. Therefore, the call is now the final element of the expression chain belonging to the previous actual parameter. Furthermore, the beginning of this expression chain is *embedded* into the unknown parts of the host graph by *redirecting* incoming edges from the call node `3'`. Finally, the `order` attribute is transfered from `3'` to `4'`, to keep ordered relations to the call node intact.

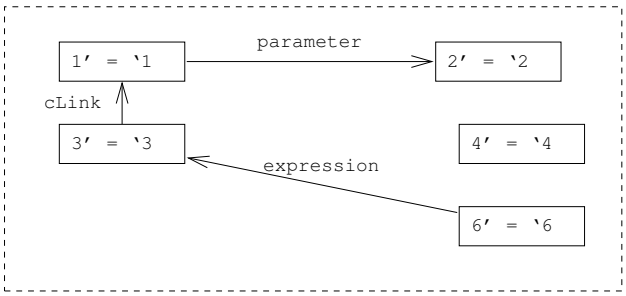
3 Prototype

Figure 2 show the prototypical editing tool. The main part of the view window is taken up by the graph viewer ①. On the left, attribute values of the currently selected node is depicted ②. Transformation rules can either be invoked by selection from the menubar, or directly from the toolbar ③. In case of the EncapsulateField rule, the UI framework queries parameters of the transformation rule, either using the toolbar as well, or through a detached window ④. Here, the

```
transformation - mm_rewriteCall_alongParam_noReceiver
 ( op : Operation ; par : Parameter) * =
```




```
::=
```




```
 folding  '4, '6 ;
 condition '2.order = '4.order;
 embedding
   redirect <-expression-, <-cActualParameter- from '3 to 4';
 transfer 4'.order := '3.order;
end;
```

Fig. 1. PROGRES transformation rule for rewriting method calls

user can input parameter values in a convenient way, e.g. using checkboxes for boolean values. To pass nodes as parameter values, the user simply selects nodes from the graph view. If a transformation rule requires multiple node-valued parameters, selected nodes are assigned in the order they were selected to those parameters they match. In case of ambiguities, the user may also enter node identifiers directly. For better navigation, ⑤ shows an overview window on the entire host graph.

Most of the prototype's functionality is generated from the underlying PROGRES specification, i.e. all transformation rules for refactoring. The behavior

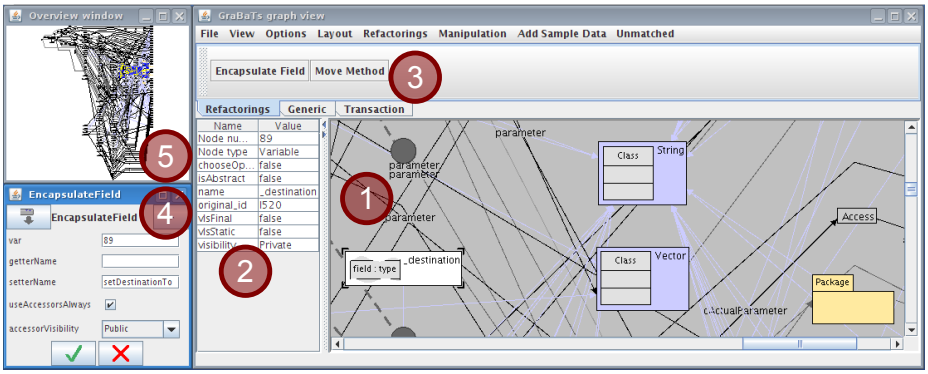


Fig. 2. Prototypical refactoring tool

of transformation selection, as discussed above, is also configured automatically from the information provided by the PROGRES specification. Some additional effort was made to provide a nicer display of graph elements, e.g. colors and icons as shown in the graph view, whilst hiding elements of lesser interest to the user. Furthermore, transformation rules are grouped into menus. This fine-tuning of the user interface can be achieved through configuration only. The only programming effort required is for importing and exporting GXL documents according to the provided schema, as the PROGRES-internal graph schema slightly diverges due to naming requirements.

4 Discussion

Language features. PROGRES, as a language for programmed graph rewriting, showed to be very much feasible to model the case studie's task. Whilst basic graph transformation rules allow visual and declarative modeling, these rules can be *combined* in an *imperative manner* including parameterization. This way, PROGRES allows e.g. to conditionally branch transformation execution, iterate over all matches, and the like. All transformations are conducted in a transaction-like atomic way - either successfully executing all transformations in a sequence, or none. For the refactoring tool contest, we can conclude that imperative control structures allow re-use and composition of elementary transformation rules.

Besides simple graph patterns comprising pattern nodes and edges, PROGRES provides a set of valuable additional language constructs. Firstly, *path expressions* allow convenient reasoning on connectivity structures, including sequences, alternatives, and conjunctions. Path expressions also allow to reason on the reached nodes regarding their type and attribute values. Paths of unknown length can be expressed by transitive closure of edge traversals and loop expressions, where the latter only retrieve the set of nodes from which no continuation exists. As shown in Figure 1, path expressions can be directly added to a rule's LHS (between `4 and `6), or refer to externally defined expressions by an iden-

tifier (between `'3` and `'5`). The latter choice is a valuable element for re-using functionality. Externally defined path expressions, which may also be attributed by scalar and node values, can be defined in a textual or visual way.

Another valuable language construct is *embedding* of nodes into an unknown context. This not only allows to conveniently redirect edges from one node to another, but also to copy edges, revert their direction, or to remove them. Although this behavior can be achieved by set-valued nodes as well, one set-valued node would be required *for each* edge type considered in an embedding directive, making an LHS considerably hard to read.

As for *genericity*, PROGRES provides a two-level type system for nodes, which allows to reason on *types of nodes*, and to specify transformation rules in a generic way. However, this is not provided for edges, as inheritance on edges comes with some nasty semantical effects [3]. The two level type system, nevertheless, has already been applied to build syntax-directed editors in the IPSEN project [4]. A set of language-independent editing rules, e.g. to add an increment to the abstract syntax tree (AST), is complemented by language-specific node types, which define the allowed AST children by means of meta-attributes. Such an approach could be followed to build generic refactoring tools as well, although this would require a sufficiently powerful generic meta model.

Development environment. PROGRES provides a sophisticated, though by modern standards difficult getting used to, development environment. Developers are aided by numerous checks considering static semantics, e.g. whether a edges in a graph pattern and their adjacent nodes match the corresponding type definition. More sophisticated checks support proper use of non-determinism, e.g. whether an expressions always delivers exactly, at most, or at least one result.

However, PROGRES does not perform checks considering rule overlapping or the like. As control flow is specified explicitly be the developer, there would be very few occasions where such an analysis could reveal programming errors, though. That said, PROGRES cannot in any way guarantee correctness of a specification w.r.t. its intended semantics, such as behavior preservation of transformation rules. It is up to the developer to specify refactoring rules to check decidable preconditions, and undecidable ones to an appropriately simplified extend.

Generated refactoring tool. The current status of the generated tool is clearly prototypical, as almost no customization of this tool have been carried out so far. Therefore, refactoring rules can be accessed by the standard UI including menus and toolbars. One could also move refactoring rules directly to the context menu of a node, simply by redefining existing transformation rules as *methods* of the respective node types.

Customized user interaction is probably the most pressing requirement for the current tool, e.g. to interactively query the user's choice for a specific element. Currently this is achieved by marker elements which are added to the host graph, although this surely isn't the most understandable approach. Furthermore, the marker approach could be extended to guide users in finding "bad smells", if

supplied with queries that determine nasty language constructs, e.g. write access to fields across class borders.

References

1. Schürr, A., Winter, A.J., Zündorf, A.: The PROGRES approach: Language and environment. In Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools. Vol. 2. 1^{st} edn. World Scientific, Singapore (1999) 487–550
2. Marburger, A.: Reverse Engineering of Complex Legacy Telecommunication Systems. PhD thesis, RWTH Aachen University (2005)
3. Schürr, A.: Operationales Spezifizieren mit programmierten Graphersetzungssystemen. Dissertation, RWTH Aachen University (1991)
4. Schürr, A.: Specification of logical documents and tools. In Nagl, M., ed.: Building Tightly Integrated Software Development Environments: The IPSEN Approach. Vol. 1170 of Lect. Notes in Comp. Sci. Springer-Verlag (1996) 297–323

A Getting & using the prototype

This section gives basic information on how to get and run the prototypical editor. It is intended as reference for the reviewers, but is not part of the paper contribution. In case of having trouble with the prototype, please contact the authors.

A.1 Getting

- Our prototype is available for download at (case sensitive): http://se.rwth-aachen.de/files/progres/GraBaTs08_Refactoring_PROGRES.tgz
  The filesize is approximately 14 megabytes.

A.2 Starting the prototype

- The downloaded archive should be extracted somewhere.
- Make sure a Java SE 5 or 6 runtime environment is installed, and the `JAVA_HOME` environment variable is set.
- The prototype is started using `Refactoring.sh` (Linux) or `Refactoring.bat` (Win32).
- The prototype was tested on recent Windows and Linux versions, we cannot guarantee its functionality on other platforms.

A.3 Running the example

- The sample code can be loaded via the File menu (Backup / Restore GXL). Loading 100k of GXL code might take some time, although we are still optimizing the solution. Alternatively, trivial demo graphs can be created via the Add Sample Data Menu.
- The generated diagram can be layouted using e.g. Layout - Sugiyama.
- Refactoring rules are applied using the Refactorings toolbar or menu.
- The prototype should be closed via File - Exit.

A.4 Inspecting the specification

- PROGRES can be freely downloaded from our website:
  http://se.rwth-aachen.de/research/progres
  However, getting it running is non-trivial due to quite out-dated library dependencies. We recommend the virtual machine image release - have a look at the Release page.
- After you launched PROGRES, single-left click on the last specification document, and type “m” (short for import)
- Enter any name, confirm with TAB
- Select the GraBaTs.backup file from the distribution to import the specification
- Right-click to open the appended specification
- Source-code generation isn’t fully possible with the current VM release - we will fix this soon. Still you may inspect the specification and try the interpreter

B Full Specification

This section comprises the full PROGRES specification used to create the refactoring tool. It is also provided in plain text format in the archive (see above) as file named GraBaTs.backup.

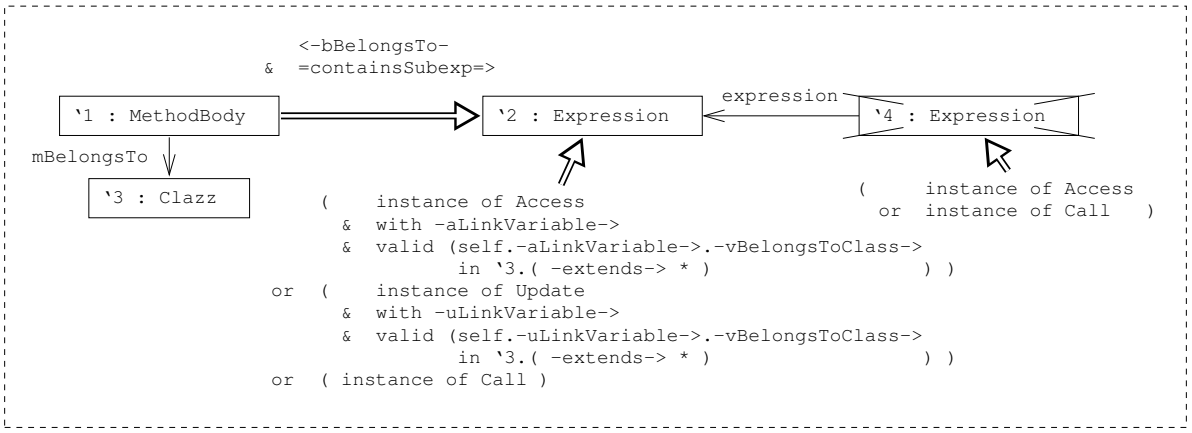
```
spec GraBats08_CS1

 declares

 section metamodel

  declares

  node class PGElement
   intrinsic
   original_id : string := "";
   derived
   chooseOption = not empty ( self.<-marker_of- );
   methods
   transformation selectMarker
   =

    use m : Marker := elem ( self.<-marker_of- )
    do
    m.select
    end
   end           ;
  end;

  node class Named is a PGElement
   intrinsic
   name : string;
  end;

  node class StructuralFeature is a Named
   intrinsic
   visibility : type in Visibility := Public;
   isAbstract := false;
  end;

  node class Classifier is a StructuralFeature end;

  node class Ordered is a PGElement
   intrinsic
   order : integer := 1;
```

```
end;

edge type extends : Classifier [0:n] -> Classifier [0:n];

edge type import : Classifier [0:n] -> Classifier [0:n];

node type Clazz : Classifier
 intrinsic
 cIsFinal : boolean := false;
end;

node type Interface : Classifier end;

edge type implements : Clazz [0:n] -> Interface [0:n];

node type Package : Named end;

edge type pBelongsTo : Package [0:n] -> Package [0:1];

edge type cBelongsTo : Classifier [0:n] -> Package [1:1];

node type Variable : StructuralFeature
 intrinsic
 vIsStatic : boolean := false;
 vIsFinal : boolean := false;
end;

edge type vType : Variable [0:n] -> Classifier [0:1];

edge type vBelongsToClass : Variable [0:n] -> Clazz [0:1];

edge type vBelongsToBody :
 Variable [0:n] -> MethodBody [0:1];

edge type vBelongsToExpr :
 Variable [0:n] -> Expression [0:1];

node type MethodBody : PGElement end;

edge type mBelongsTo : MethodBody [0:n] -> Clazz [1:1];

edge type binding : Operation [1:1] -> MethodBody [0:n];

node type Operation : StructuralFeature
 intrinsic
```

```
 oIsStatic : boolean := false;
 oIsFinal : boolean := false;
end;

edge type oBelongsTo : Operation [0:n] -> Interface [0:1];

edge type oType : Operation [0:n] -> Classifier [0:1];

edge type parameter : Operation [1:1] -> Parameter [0:n];

node type Parameter : Ordered end;

edge type pType : Parameter [0:n] -> Classifier [1:1];

node type Literal : PGElement
 intrinsic
 val : string;
end;

edge type lType : Literal [0:n] -> Clazz [1:1];

node class Expression is a Ordered end;

edge type expression : Expression [0:1] -> Expression [0:n];

node type Access : Expression
 intrinsic
 aThis : boolean := false;
end;

edge type aLinkLiteral : Access [1:1] -> Literal [0:1];

edge type aLinkParameter : Access [0:n] -> Parameter [0:1];

edge type aLinkVariable : Access [0:n] -> Variable [0:1];

edge type aLinkClass : Access [0:n] -> Clazz [0:1];

node type Update : Expression
 intrinsic
 uThis : boolean := false;
end;

edge type uLinkParameter : Update [0:n] -> Parameter [0:1];
```

```
 edge type uLinkVariable : Update [0:n] -> Variable [0:1];

 node type Call : Expression
  intrinsic
  cThis : boolean := false;
  cSuper : boolean := false;
 end;

 edge type cActualParameter : Call [0:1] -> Expression [0:n];

 edge type cLink : Call [0:n] -> Operation [0:1];

 node type Instantiation : Expression end;

 node type Operator : Expression
  intrinsic
  oName : string;
 end;

 node type Return : Expression end;

 node type Block : Expression end;

 edge type bBelongsTo : Block [0:1] -> MethodBody [0:1];

end;

section mmadditions

 declares

 path ofClass : Expression [0:n] -> Clazz [0:1] =
    <-expression-
  & [ ( instance of Block
    & -bBelongsTo->
    & -mBelongsTo->  )
   | ( <-cActualParameter-
    & =ofClass=>   ) ]
 end;

 path ofOperation : Expression [0:n] -> Operation [0:n] =
    <-expression-  : [0:1]
  & [ ( instance of Block
    & -bBelongsTo->
    & <-binding-  )
```

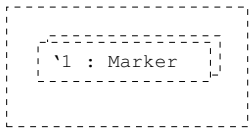
```
  | ( <-cActualParameter-
   & =ofOperation=>  ) ]
end;

path containsSubexp : Expression [0:1] -> Expression [0:n] =
 (  -expression->
 or ( instance of Call
   & -cActualParameter-> ) ) *
end;

path containsExp : Clazz [1:1] -> Expression [0:n] =
  <-mBelongsTo-
 & <-bBelongsTo-
 & =containsSubexp=>
end;

path explicitThisRef : MethodBody [0:1] -> Expression [0:n]
 =
  <-bBelongsTo-
 & =containsSubexp=>
 & (  ( instance of Update
    & valid (self.uThis) )
  or ( instance of Access
    & valid (self.aThis) )
  or ( instance of Call
    & valid (self.cThis) ) )
end;

path implicitThisRef : MethodBody [0:1] -> Expression [0:n]
 =
 '1 => '2 in
```



```
 end;

 path exprParent : Expression [0:n] -> Expression [0:n] =
   <-expression-
  & (  instance of Call
   or instance of Access )
 end;

end;

section utility

 declares

 node class ENUM
  meta
  ordinal : integer;
 end;

 node class Visibility is a ENUM end;

 function '<=' : ( left, right : type in Visibility) ->
  boolean =
  left.ordinal <= right.ordinal
 end;

 node type Public : Visibility
  redef meta
  ordinal := 1 ;
 end;

 node type Protected : Visibility
  redef meta
  ordinal := 2 ;
 end;

 node type Default : Visibility
  redef meta
  ordinal := 3 ;
 end;

 node type Private : Visibility
  redef meta
  ordinal := 4 ;
 end;
```

```
node class Accessor is a ENUM end;

node type Getter : Accessor
 redef meta
 ordinal := 1 ;
end;

node type Setter : Accessor
 redef meta
 ordinal := 2 ;
end;

function - accessorName :
 ( v : Variable ; variant : type in Accessor ;
 predefName : string [0:1])     -> string =
 [ not empty ( predefName ) :: predefName : [1:1]
 | [ variant = Getter :: "get"
 | "set" ]     & toFirstUpper ( v.name ) ]
end;

function - toFirstUpper : ( n : string) -> string =
 use first := substr ( n, 1, 1 ) ::
 [ first = "a" :: "A"
 | first = "b" :: "B"
 | first = "c" :: "C"
 | first = "d" :: "D"
 | first = "e" :: "E"
 | first = "f" :: "F"
 | first = "g" :: "G"
 | first = "h" :: "H"
 | first = "i" :: "I"
 | first = "j" :: "J"
 | first = "k" :: "K"
 | first = "l" :: "L"
 | first = "m" :: "M"
 | first = "n" :: "N"
 | first = "o" :: "O"
 | first = "p" :: "P"
 | first = "q" :: "Q"
 | first = "r" :: "R"
 | first = "s" :: "S"
 | first = "t" :: "T"
 | first = "u" :: "U"
 | first = "v" :: "V"
```

```
  | first = "w" :: "W"
  | first = "x" :: "X"
  | first = "y" :: "Y"
  | first = "z" :: "Z"
  | first ]   & substr ( n, 2, length ( n ) )
  end
 end;

 function - min : ( i : integer [0:n]) -> integer =
  min2 ( 0, all i )
 end;

 function - min2 : ( i, j : integer) -> integer =
  [ i > j :: j
  | i ]
 end;

 function - max : ( i : integer [0:n]) -> integer =
  max2 ( 0, all i )
 end;

 function - max2 : ( i, j : integer) -> integer =
  [ i < j :: j
  | i ]
 end;

end;

section markers

 declares

 node class Marker
  intrinsic
  description : string;
  methods
  transformation select;
 end;

 edge type marker_of : Marker [0:1] -> PGElement [1:1];

 transformation + mrk_ClearChooses =
```



::=

end;

end;

section refactorings

declares

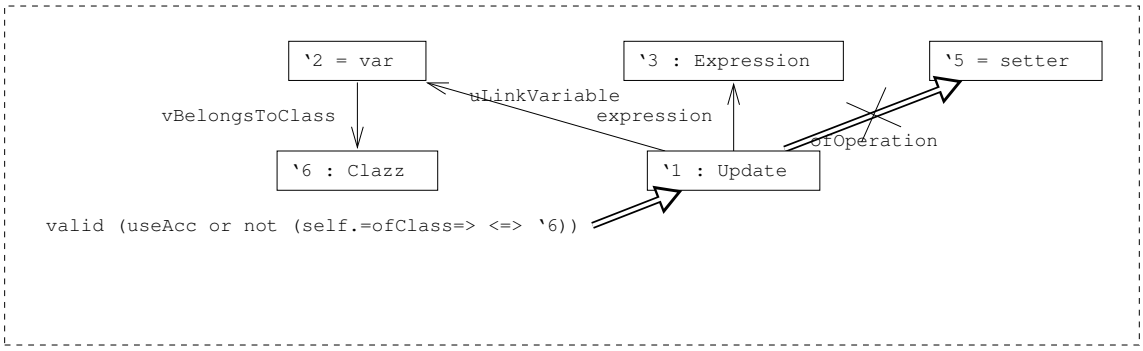
section s_EncapsulateField

declares

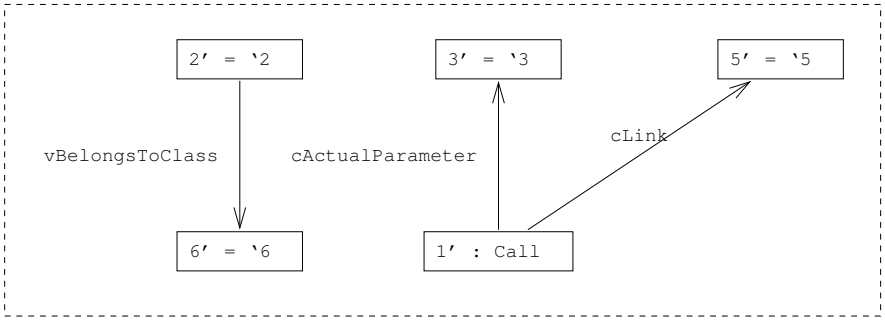
transformation EncapsulateField
(var : Variable ; getterName : string [0:1] ;
setterName : string [0:1] ; useAccessorsAlways : boolean ;
accessorVisibility : type in Visibility)
=

skip
& not empty (var.-vBelongsToClass->)
&
use
go : Operation [0:1]
:=
elem
(var.-vBelongsToClass->.<-mBelongsTo-.<-binding-.valid
(self.name = accessorName (var, Getter, getterName)))
(*search for method of adequate name *) ;
so : Operation [0:1]
:=
elem
(var.-vBelongsToClass->.<-mBelongsTo-.<-binding-.valid
(self.name = accessorName (var, Setter, setterName)))
(*search for method of adequate name *) ;
newVis : type in Visibility

```
  := elem ( ef_NewVisibility ( accessorVisibility, var ) )
 do
 skip
 & for all op := elem ( go or so ) ::
  (op.oIsStatic = var.vIsStatic)
  and not (op.isAbstract)
  and (op.visibility.ordinal <= newVis.ordinal)
  end
  (*Check preconditions on existing operations *)
 &
 choose
  when empty ( go )
  then
  ef_DeclareGetter
   ( var, accessorName ( var, Getter, getterName ), newVis,
   out go )
 else
  skip
 end
 (*Declare getter unless present *)
 &
 choose
  when empty ( so )
  then
  ef_DeclareSetter
   ( var, accessorName ( var, Setter, setterName ), newVis,
   out so )
 else
  skip
 end
 (*Declare setter unless present *)
 & ef_ReplaceUpdate ( var, so : [1:1], useAccessorsAlways )
 & ef_ReplaceAccess ( var, go : [1:1], useAccessorsAlways )
 & var.visibility := Private (*hide variable *)
 end
end;

transformation - ef_ReplaceUpdate
 ( var : Variable ; setter : Operation ; useAcc : boolean)
 * =
```

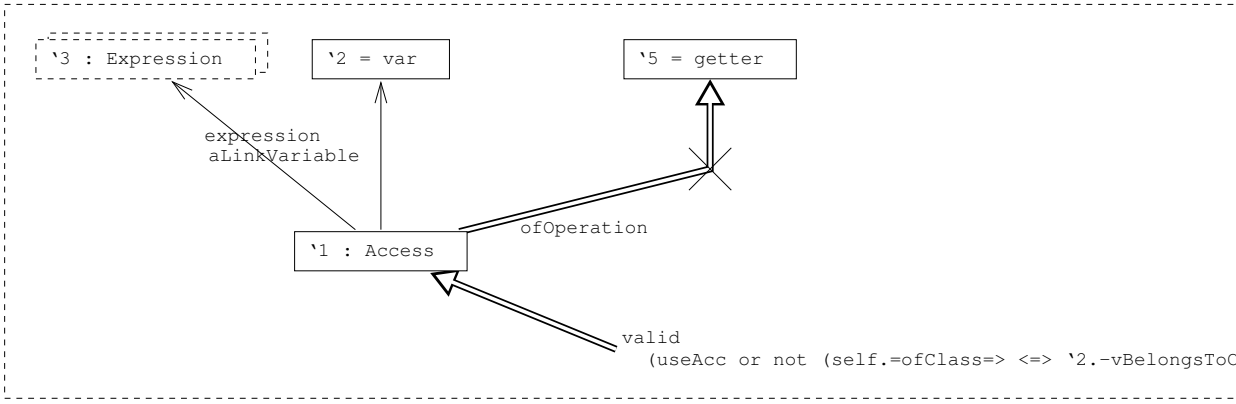


::=

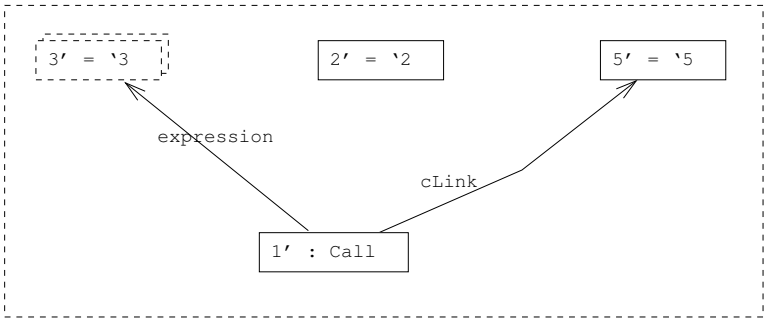


embedding
redirect <-expression-, <-cActualParameter- from ‘1 to 1’;
transfer 1’.order := ‘1.order;
end;

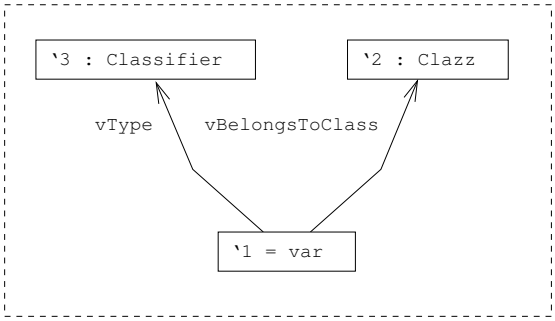
transformation - ef_ReplaceAccess
(var : Variable ; getter : Operation ; useAcc : boolean)
* =



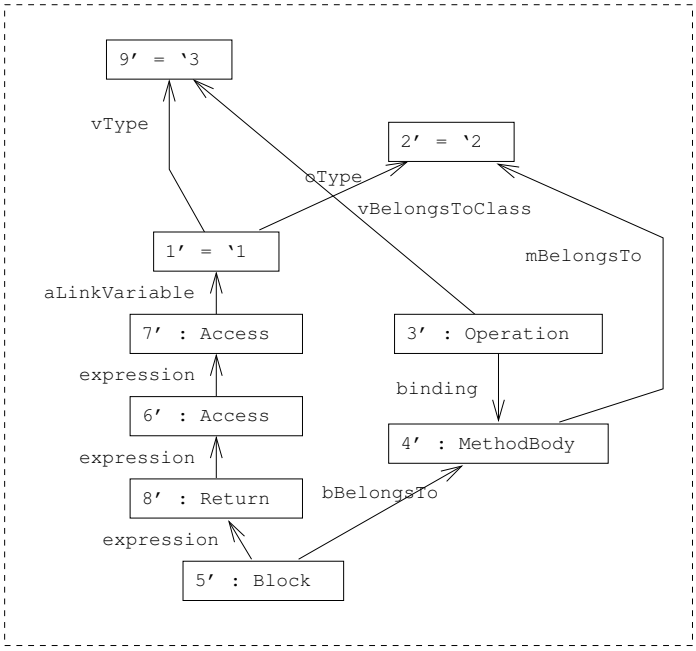
::=



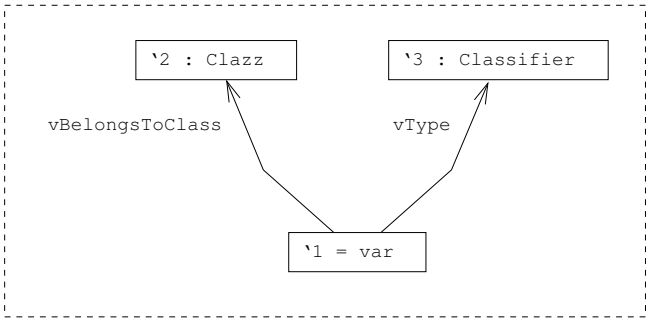

```
 embedding
 redirect <-expression-, <-cActualParameter- from '1 to 1';
 transfer 1'.order := '1.order;
end;

transformation - ef_DeclareGetter
 ( var : Variable ; gname : string ;
 newVis : type in Visibility ; out op : Operation) =
```



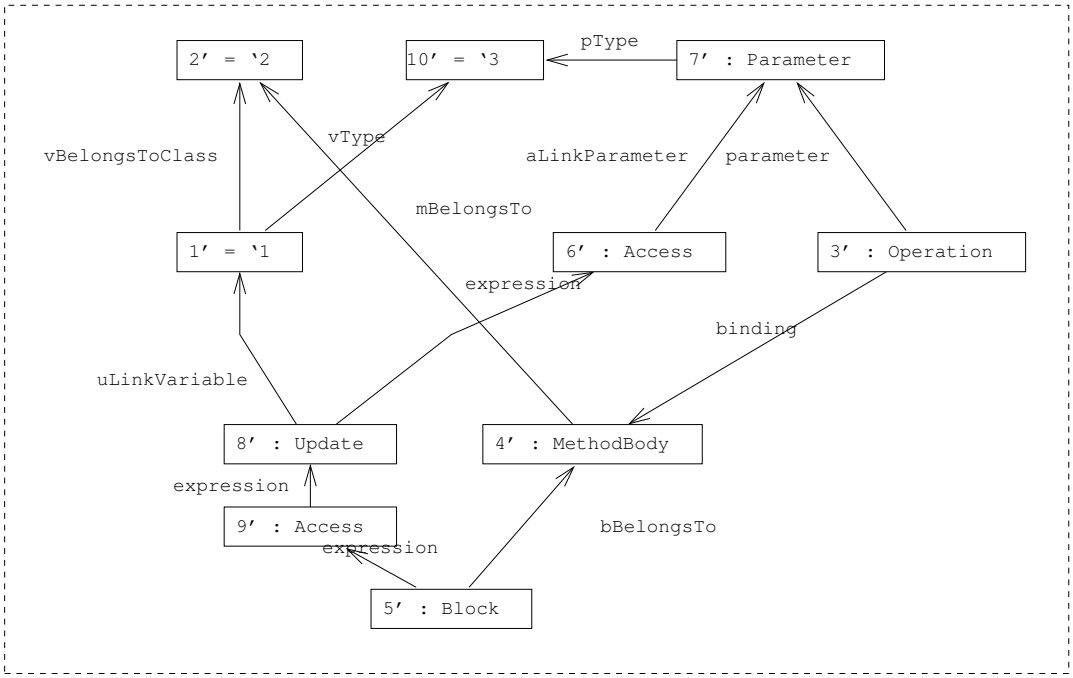
::=




```
 folding  '2, '3 ;
 transfer 3'.name := gname;
   3'.oIsStatic := '1.vIsStatic;
   6'.aThis := true;
   3'.visibility := newVis;
 return op := 3';
end;

transformation - ef_DeclareSetter
 ( var : Variable ; sname : string ;
 newVis : type in Visibility ; out op : Operation) =
```



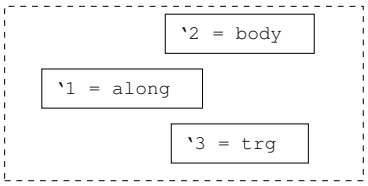

```
 ::=
```



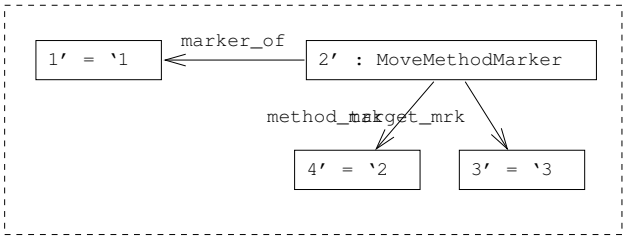

```
  folding  '2, '3 ;
  transfer 3'.name := sname;
    3'.oIsStatic := '1.vIsStatic;
    3'.visibility := newVis;
    9'.aThis := true;
  return op := 3';
 end;

 function - ef_NewVisibility :
  ( selVis : type in Visibility ; var : Variable) ->
  type in Visibility [0:1] =
  [ var.visibility = Public :: Public
  | var.visibility = Private :: selVis
  | (var.visibility in (Protected or Default))
  and (selVis in (Public or var.visibility)) ::
  selVis          ]
 end;

end;

section s_MoveMethod

 declares
```

```
transformation MoveMethod
 ( body : MethodBody ; trg : Clazz ; use_delegation : boolean)
 =

 not (body.<-binding-.oIsStatic)
 & not mm_conflictingMoval ( body, trg )
       (*will not override method *)
 & not mm_hasSuperCall ( body )
      (*no call to super constructor *)
 & not (body.-mBelongsTo-> <=> trg)
  (*do not move to own class *)
 & not (body.<-binding-.name = body.-mBelongsTo->.name)
  (*not a constructor *)
 & empty ( instance of Marker )
 (*no selection open *)
 &
 for all candidate : PGElement
   := elem ( body.=mm_incrementCandidates ( trg )=> )
 do
 mm_createSelection ( body, trg, use_delegation, candidate )
 end
 &
 use choices : Marker [0:n] := (instance of Marker)
 do
 choose
  when empty ( choices )
  then
  fail
  (*No choice induces transaction failure *)
 else
  when (card ( choices ) = 1)
  then
  use c : Marker [1:1] := elem ( choices )
  do
   c.select (*Unique choice is selected automatically *)
  end
 else
  skip
  (*Requires User interaction to select choice. *)
 end
 end
end;

transformation mm_createSelection
 ( body : MethodBody ; trg : Clazz ; use_delegation : boolean
```

```
 ; along : PGElement)
 =
```



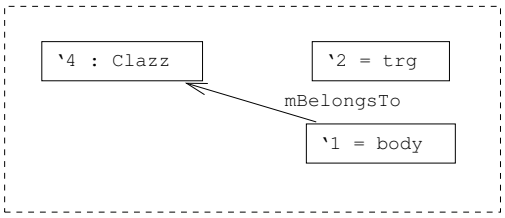

```
 ::=
```



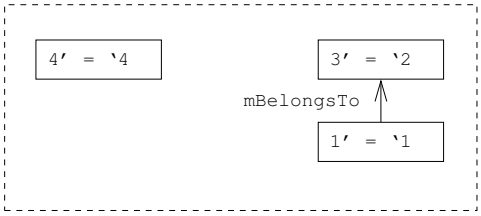

```
 transfer 2'.useDelegation_mrk := use_delegation;
end;

node type MoveMethodMarker : Marker
 intrinsic
 method_mrk : MethodBody;
 useDelegation_mrk : boolean;
 target_mrk : Clazz;
 methods redef
 redef transformation select =

  use src : Clazz := self.method_mrk.-mBelongsTo->;
   op : Operation := self.method_mrk.<-binding-;
   along := self.marker_of
  do
  choose
   when self.useDelegation_mrk
    or not mm_isLocalDecl ( self.method_mrk )
   then
   use deleg_op : Operation;
    call : Call
   do
    mm_addDelegate_base
    ( self.method_mrk, src, self.marker_of, out deleg_op,
     out call )
    & choose
```

```
   mm_addDelegate_return ( deleg_op )
  else
   skip
  end
  & mm_addDelegate_parameters
   ( op, deleg_op, call, self.marker_of )
 end
else
 skip
end
(*When using delegates, add them immediately to the source class. *)
&
choose
 when
 exist e := self.method_mrk.(  ( =explicitThisRef=>
          & -expression->  )
         or =implicitThisRef=>  )
  ::
  not (e.( instance of Access
    & -aLinkVariable-> ) <=> along)
  and not (e.( instance of Update
     & -uLinkVariable-> ) <=> along)
 end
 (*This access or update, except to the variable being moved along *)
 or exist c := self.method_mrk.<-binding-.<-cLink- ::
   not empty ( c.( =exprParent=> +
       & instance of Call ) )
  end
  (*Complex call: having a call as receiver *)
 then
 use thisParam : Parameter
 do
  mm_rewriteOp_sourceParamDecl ( op, src, out thisParam )
  & mm_rewriteOp_sourceParamAppl_explicit ( thisParam )
  & mm_rewriteOp_sourceParamAppl_implicit ( thisParam )
  & mm_rewriteOp_sourceParam_simplify
   ( thisParam, self.marker_of )
  & mm_rewriteCall_sourceParam_implicitReceiver ( thisParam )
  & mm_rewriteCall_sourceParam_explRcv ( thisParam )
 end
else
 skip
end
&
choose
```

```
    when self.useDelegation_mrk
     or not mm_isLocalDecl ( self.method_mrk )
    then
    skip
   else
    choose
    when (self.marker_of is instance of Variable)
    then
     mm_rewriteCall_variable ( op, self.marker_of : Variable )
    else
    when (self.marker_of is instance of Parameter)
    then
     mm_rewriteCall_alongParam_removeRcv
     ( op, self.marker_of : Parameter )
     & mm_rewriteCall_alongParam_noReceiver
      ( op, self.marker_of : Parameter )
    end
   end
   (*If NOT using delegates, rewrite calls acoording to move-along
   element after treatmet of 'this'references. *)
   & mm_moveBody ( self.method_mrk, self.target_mrk )
       (*Right, we still have to move a method. *)
   & choose
    mm_rewriteOp_param ( op, self.marker_of )
    else
    skip
    end
    (*Remove parameters we moved along - just in case. *)
   & mm_fixVisibility ( self.method_mrk )
   & mrk_ClearChooses
   end
  end;
end;

transformation - mm_moveBody
  ( body : MethodBody ; trg : Clazz) =
```

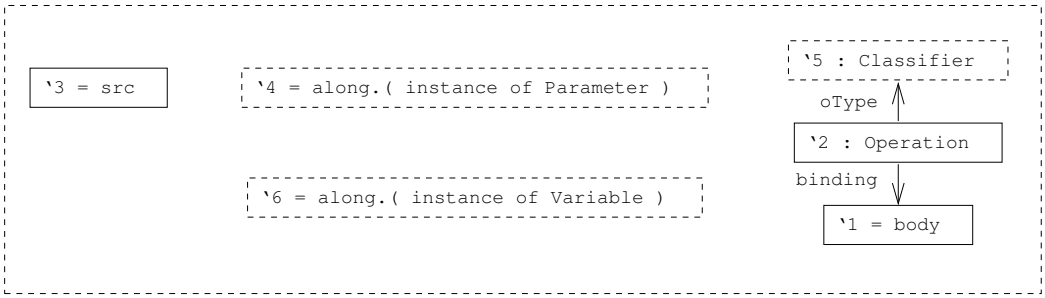



```
  ::=
```

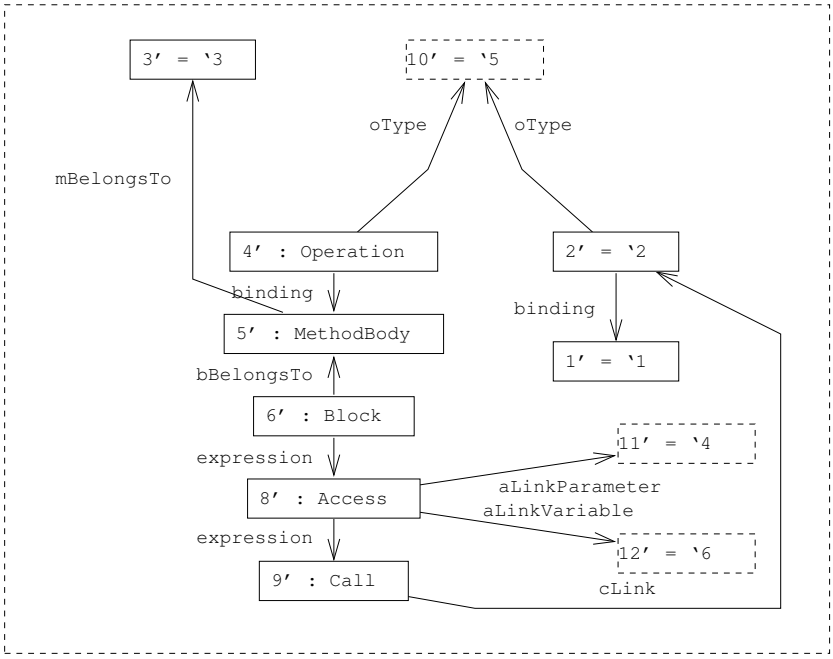


end;

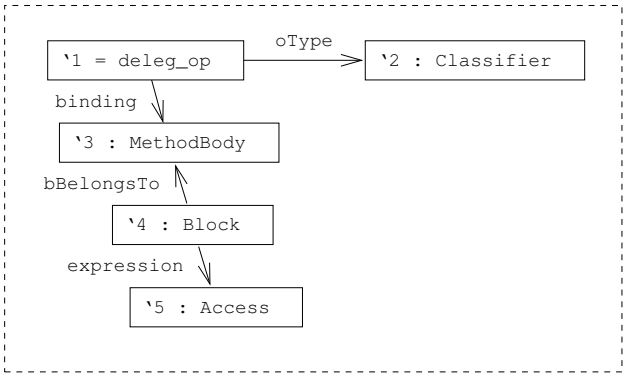
transformation - mm_addDelegate_base
(body : MethodBody ; src : Clazz ; along : PGElement ;
out op : Operation ; out call : Call) =



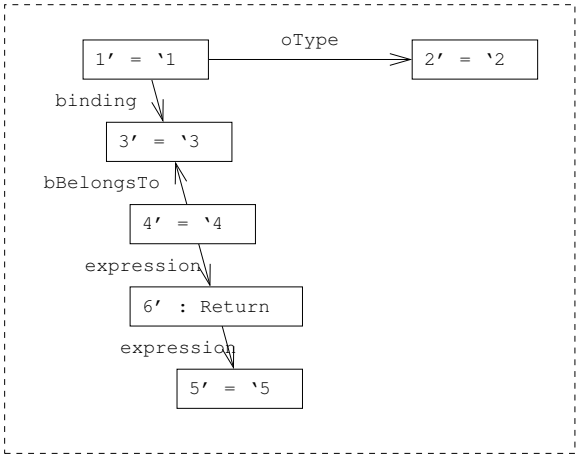
::=



```
 embedding redirect <-cLink- from `2 to 4';
 transfer 4'.name := `2.name;
   4'.oIsFinal := `2.oIsFinal;
   4'.visibility := `2.visibility;
   2'.visibility := Public;
 return op := 4';
   call := 9';
end;
(*Replaces the method to be moved by a delegation method,
though not considering parameters. Furthermore, any call to the
moving method is redirected to the newly created delegate method.
*)

transformation - mm_addDelegate_return
 ( deleg_op : Operation) =
```

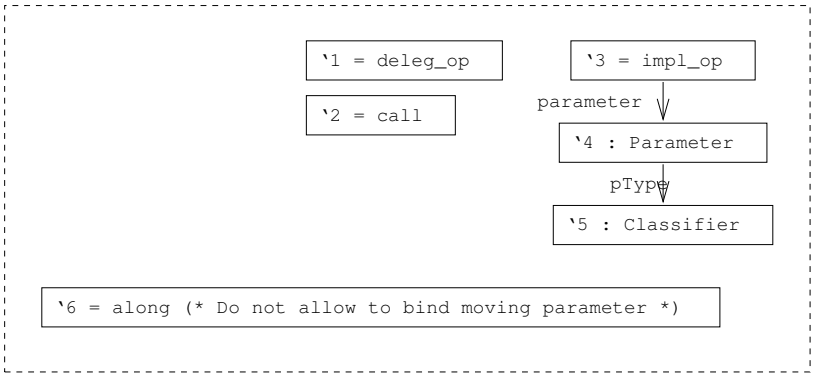


::=

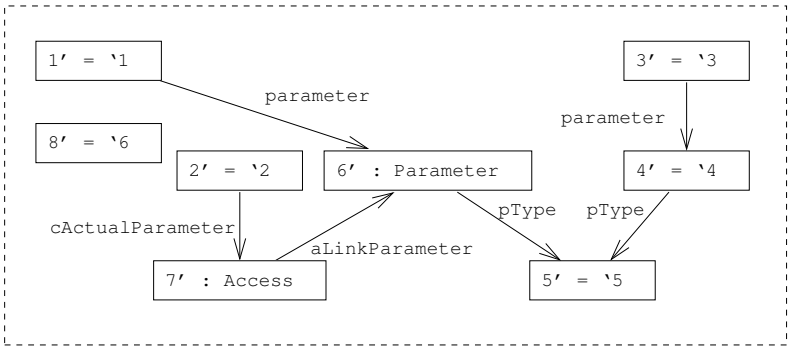


end;
(*If the method to be moved has a return value, then
a return statement is required. *)

transformation - mm_addDelegate_parameters
(impl_op, deleg_op : Operation ; call : Call ;
along : PGElement) * =

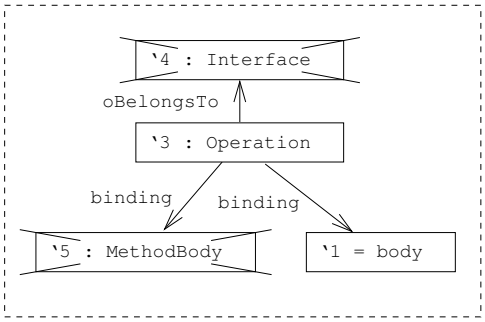


::=

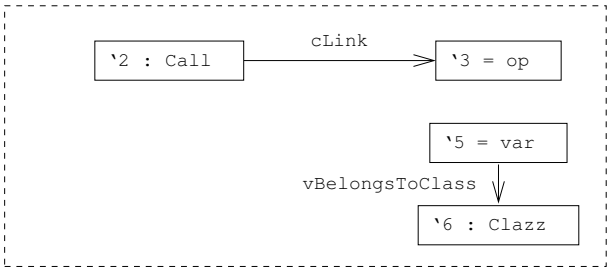


transfer 6'.order := '4.order;
7'.order := '4.order;
end;
(*For each parameter of the moving method, with exception of the
parameter the method is being moved along, an access and
actual parameter is added to the delegation method.
*)

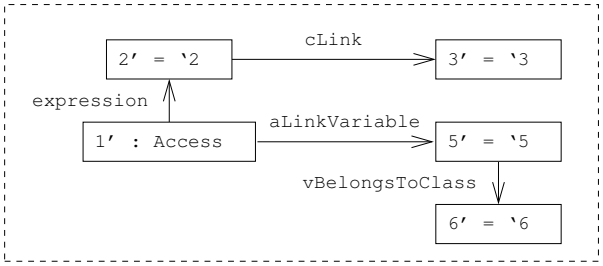
query - mm_isLocalDecl(body : MethodBody) =




```
end;
(*An operation is called 'locally declared'iff
it has a unique binding, and is not declared in some
interface.
*)

transformation - mm_rewriteCall_variable
  ( op : Operation ; var : Variable) * =
```

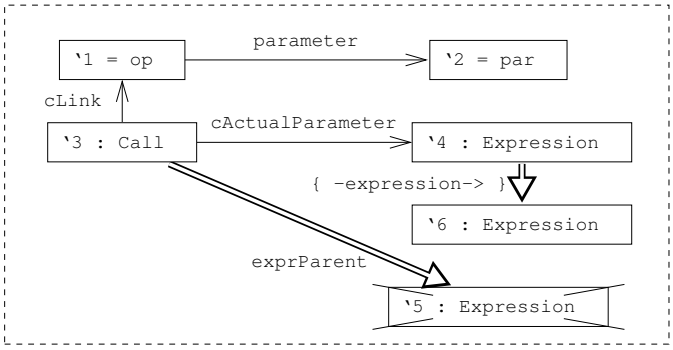



```
  ::=
```

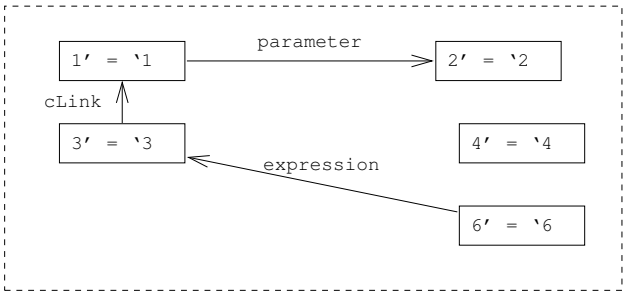



```
  embedding
  redirect <-expression-, <-cActualParameter- from '2 to 1';
  transfer 1'.order := '2.order;
    2'.order := 1;
end;
```

```
(*Rewrites calls to the moved method by redirecting calls
through the selected variable. If the respective call does
not belong to the class of var, then the var's visibility
is changed to Public, otherwise it is kept.
*)

transformation - mm_rewriteCall_alongParam_noReceiver
  ( op : Operation ; par : Parameter) * =
```

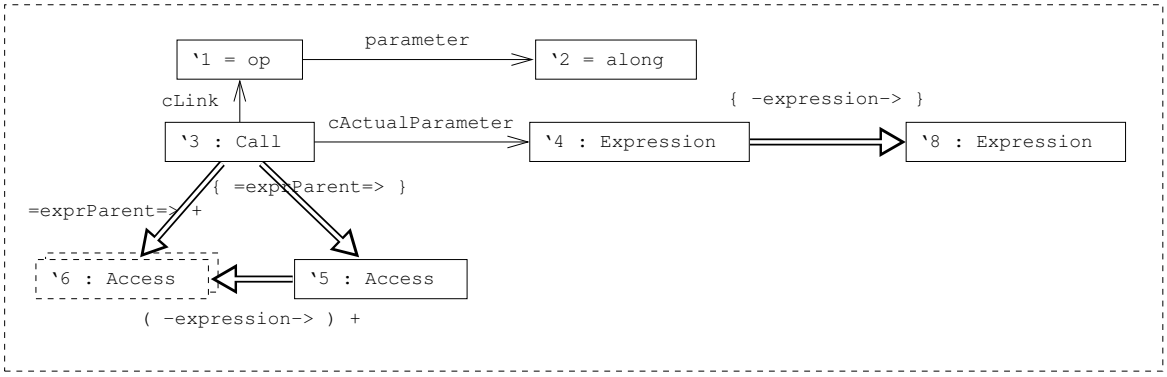



```
  ::=
```

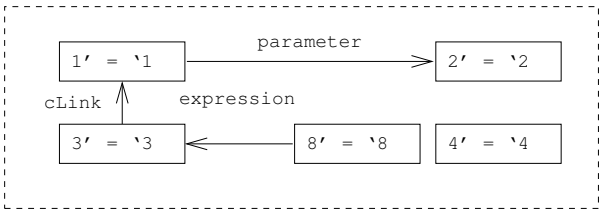



```
  folding  '4, '6 ;
  condition '2.order = '4.order;
  embedding
  redirect <-expression-, <-cActualParameter- from '3 to 4';
  transfer 4'.order := '3.order;
end;
(*Rewrites accesses to the former method container and
 subsequent calls to the moved method by invoking the call
 on the actual parameter of the respective formal parameter
 where the method is being moved along.
 This variant treats cases where the call's container is
 simply retained (e.g. this-calls or remainders of thisParam
 productions).
```

```
*)
(*Known Bugs: Expects '6 to be unique, which isnt't the case e.g.
 for binary operators - would require a proper AST structure.
*)

transformation - mm_rewriteCall_alongParam_removeRcv
  ( op : Operation ; along : Parameter) * =
```

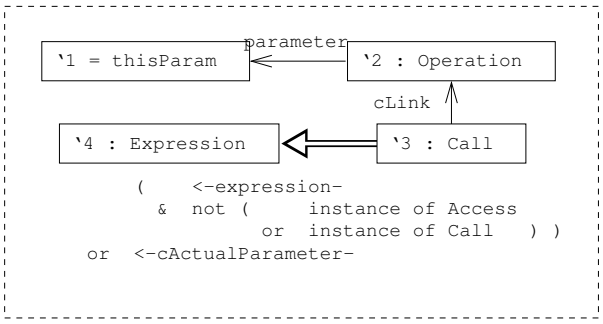



```
  ::=
```

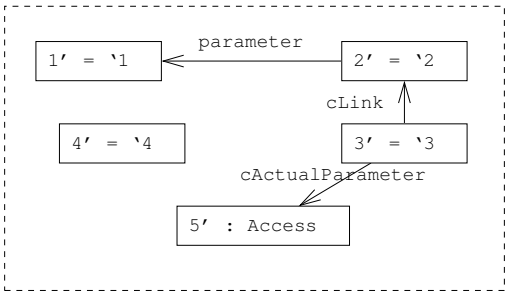



```
 folding  '4, '8 ;
 condition '2.order = '4.order;
 embedding
 redirect <-expression-, <-cActualParameter- from '5 to 4';
 transfer 4'.order := '5.order;
end;
(*Rewrites accesses to the former method container
and subsequent calls to the moved method by invoking the
call on the actual parameter of the respective formal parameter
where the method is being moved along.
This variant treats the case where the method is invoked from an
expression of accesses, which are DELETED here ('5, '6). Note that any
calls on the expression chain must be preserved, which implies
using them as 'this'parameter value.
*)
```

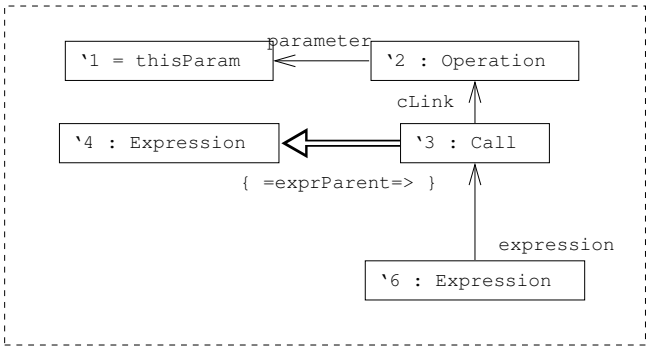
transformation - mm_rewriteCall_sourceParam_implicitReceiver
(thisParam : Parameter) * =



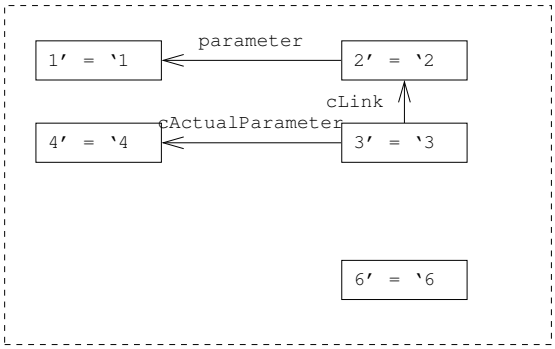
::=



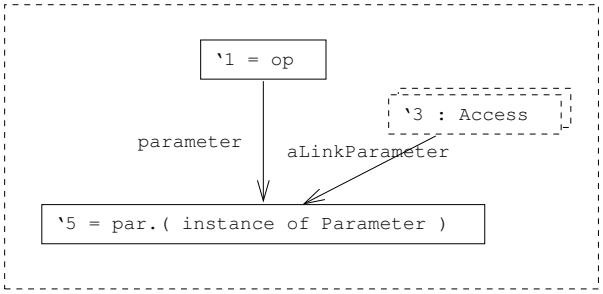

```
 transfer 5'.order := '1.order;
   5'.aThis := true;
end;
(*Rewrites calls to the moved method in a 'this'context, i.e.
with no explicit receiver reference. In this case, an explicit
'this'access is inserted.
*)

transformation - mm_rewriteCall_sourceParam_explRcv
 ( thisParam : Parameter) * =
```



::=

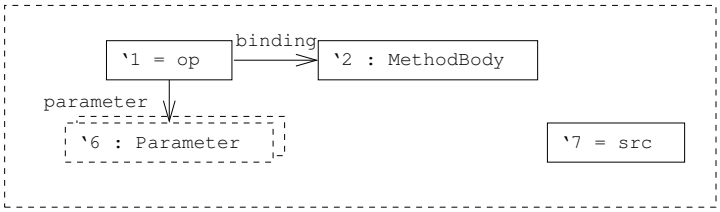



```
 folding  '4, '6 ;
 condition not '3.cThis;
 embedding
 redirect <-expression-, <-cActualParameter- from '4 to 3';
 transfer 3'.order := '4.order;
   4'.order := '1.order;
end;
(*Rewrites calls to the moved method, where the invocation receiver
is determined by an expression. It does so by searching the first
call / access in a sequence of such, and uses this as the actual
parameter. The previous container of this call / access is used
as the container of the rewritten method call.
*)

transformation - mm_rewriteOp_param
 ( op : Operation ; par : PGElement) =
```

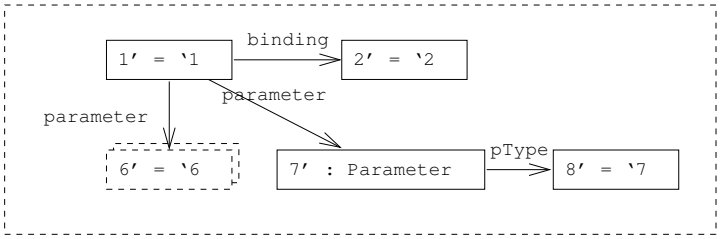


::=

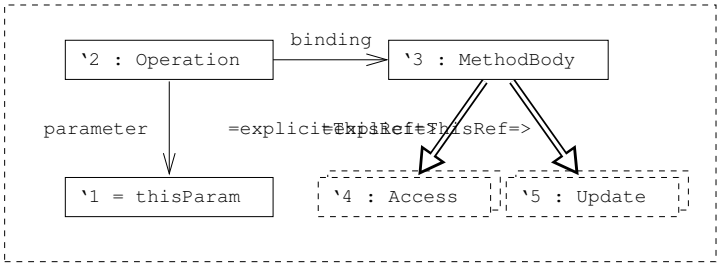

```
 transfer 3'.aThis := true;
end;
(*Rewrites accesses to the formal parameter this method was moved along
 to 'this'references, and removes the old parameter.
*)

transformation - mm_rewriteOp_sourceParamDecl
 ( op : Operation ; src : Clazz ; out thisParam : Parameter)
 =
```



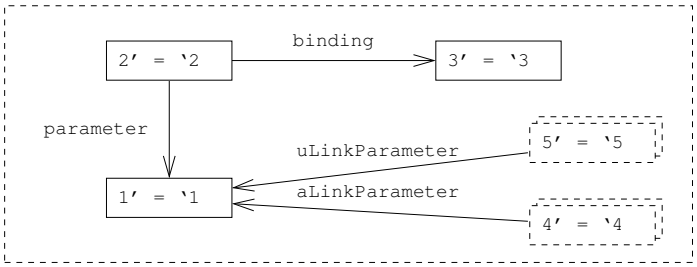
::=



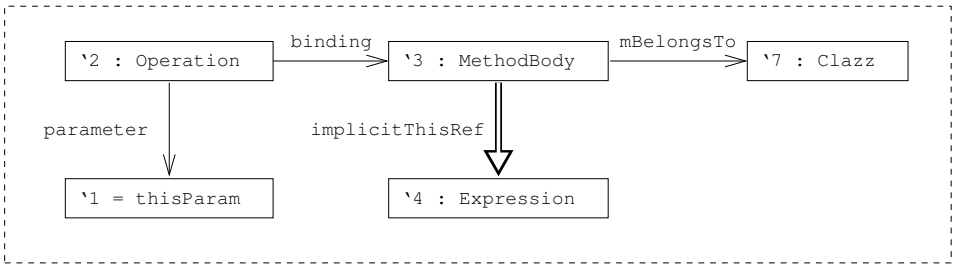
```
 transfer 7'.order := min ( `6.order ) - 1;
 return thisParam := 7';
end;
(*If the method being moved has 'this'references in its implementation,
 then a parameter for the source class is added.
*)

transformation - mm_rewriteOp_sourceParamAppl_explicit
 ( thisParam : Parameter) * =
```



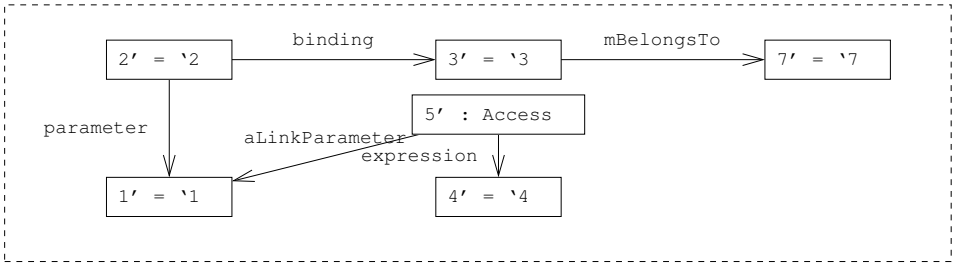

```
 ::=
```



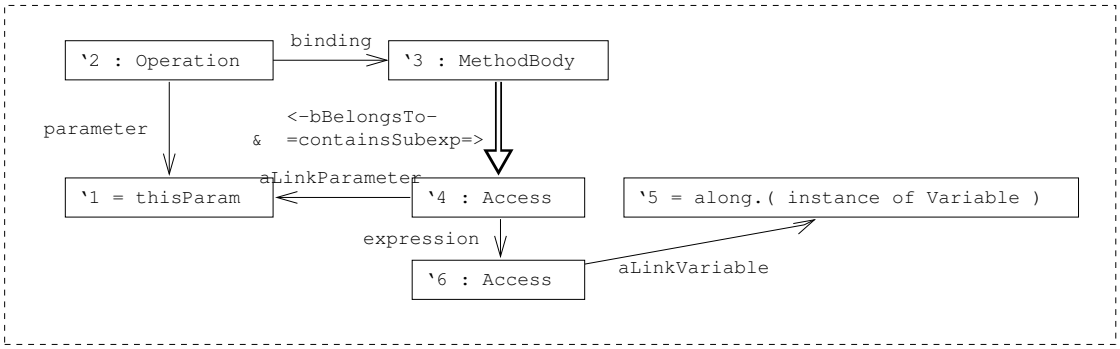

```
 transfer 4'.aThis := false;
   5'.uThis := false;
end;
(*Replaces any this references of accesses and updates in
the moved method's body by a reference to the newly introduced parameter.
*)

transformation - mm_rewriteOp_sourceParamAppl_implicit
 ( thisParam : Parameter) * =
```



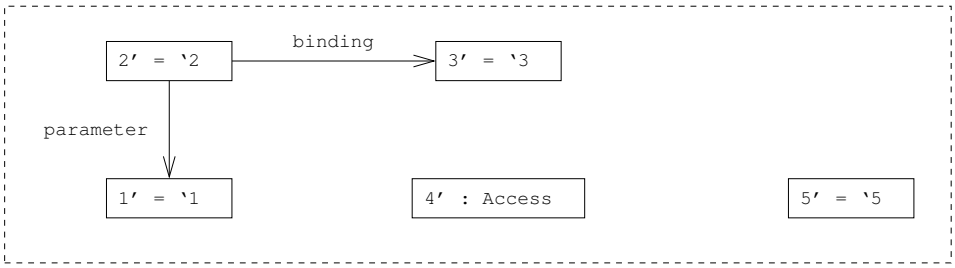
::=



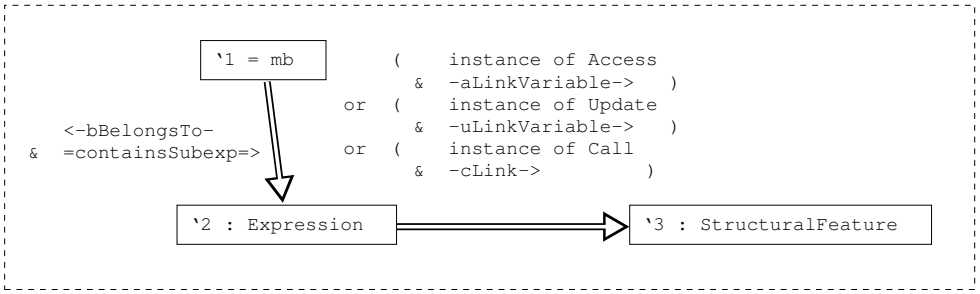

```
 embedding
 redirect <-expression-, <-cActualParameter- from '4 to 5';
 transfer 5'.order := '4.order;
   4'.order := 1;
end;

transformation - mm_rewriteOp_sourceParam_simplify
 ( thisParam : Parameter ; along : PGElement) * =
```



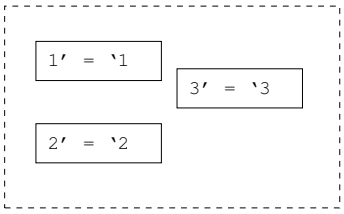
::=



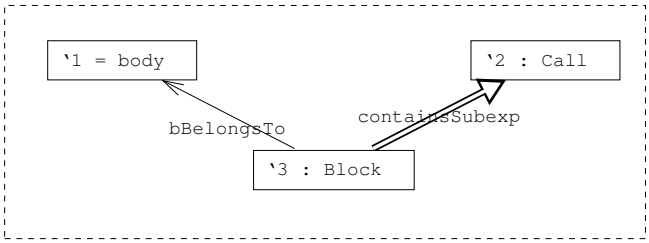

```
 embedding
 redirect <-expression-, <-cActualParameter- from '4 to 4';
   redirect -expression-> from '6 to 4';
 transfer 4'.order := '4.order;
   4'.aThis := true;
end;

transformation - mm_fixVisibility ( mb : MethodBody) * =
```



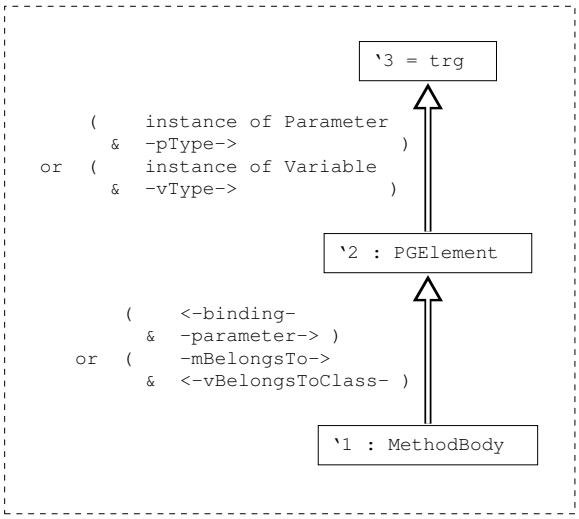

```
 ::=
```



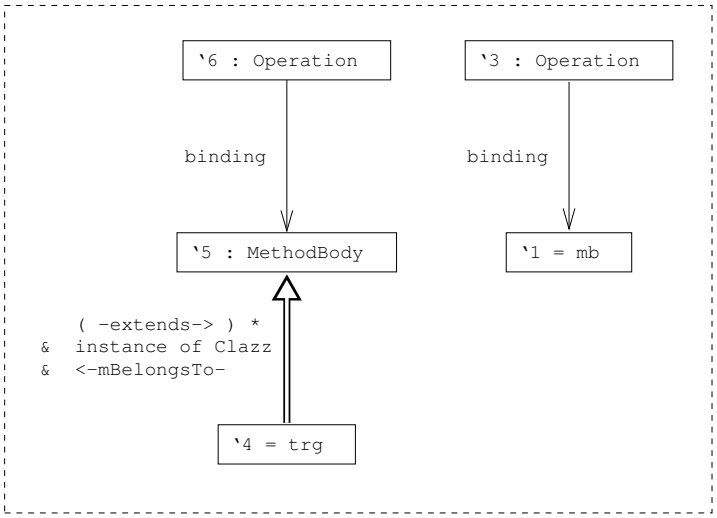

```
 transfer 3'.visibility :=
    use vis : type in Visibility [1:1]
     := mm_requiresVisibility ( '2, '3 ) ::
    [ vis.ordinal > '3.visibility.ordinal :: vis
    | '3.visibility ]
    end           ;
end;

query - mm_hasSuperCall( body : MethodBody) =
```




```
 condition ‘2.cSuper;
end;

path - mm_incrementCandidates( trg : Clazz) :
 MethodBody -> PGElement =
 ‘1 => ‘2 in
```



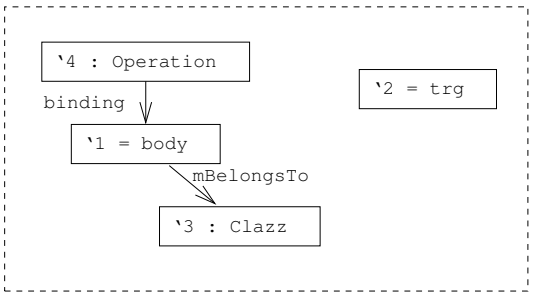

```
end;

query - mm_conflictingMoval( mb : MethodBody ; trg : Clazz)
=
```



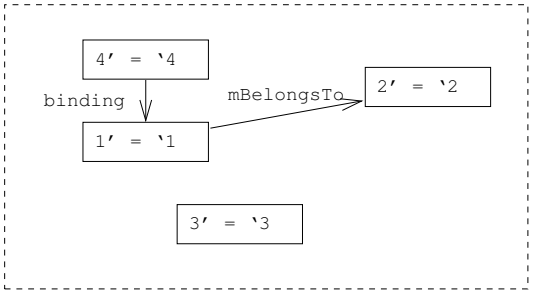

```
  condition '3.name = '6.name;
 end;

 function - mm_requiresVisibility :
  ( exp : Expression ; f : StructuralFeature) ->
  type in Visibility =
  use fromCls : Classifier [0:1] := exp.=ofClass=> ::
  use toCls : Classifier [0:n]
    := [ f.( instance of Variable
      & -vBelongsToClass-> )
     | f.( instance of Operation
      & [ -oBelongsTo->
       | -binding->
       & -mBelongsTo-> ] ) ] ::
   [ fromCls = toCls :: Private
   | fromCls.-cBelongsTo-> <=> toCls.-cBelongsTo-> :: Default
   | toCls implies fromCls.( -extends-> + ) :: Protected
   | Public ]
  end
  end
 end;

 end;

end;

section s_pullupMethod

 declares
```

```
transformation pullUpMethod
 ( trg : Clazz ; body : MethodBody ; use_supertype : boolean
 ; make_abstract : boolean ; keep : Clazz [0:n])
 =

 (trg in body.-mBelongsTo->.( -extends-> + ))
 (*Has superclass, trg is in this hierarchy *)
 &
 not
 (exist trg_op := trg.<-mBelongsTo-.<-binding- ::
  pm_sameSignature ( body.<-binding-, trg_op )
  end           )
 (*Superclass does not have method with same signature *)
 & for all e := body.(  =implicitThisRef=>
      or ( =explicitThisRef=>
        & -expression->  ) ) ::
  e.(  ( (  ( instance of Access
       & -aLinkVariable-> )
      or ( instance of Update
       & -uLinkVariable-> ) )
    & -vBelongsToClass->    )
   or ( instance of Call
    & -cLink->
    & [ -oBelongsTo->
     | -binding->
      & -mBelongsTo-> ] )    )
  in trg.( -extends-> * )
 end
 (*accessed members belong to (a superclass of)trg *)
 &
 not
 (body.<-binding-.name = body.-mBelongsTo->.name)
 (*method name does not equal class name (constructor!)*)
 &
 for all sc := trg.( <-extends- + ).( instance of Clazz ) ::
 for all sc_op := sc.<-mBelongsTo-.<-binding- ::
  pm_sameSignature ( body.<-binding-, sc_op )
  implies (body.<-binding-.-oType-> <=> sc_op.-oType->)
 end
 end
 (*same method signature in sub-types of the target
  class have the same return type *)
 & use op := body.<-binding-;
  src := body.-mBelongsTo->
```

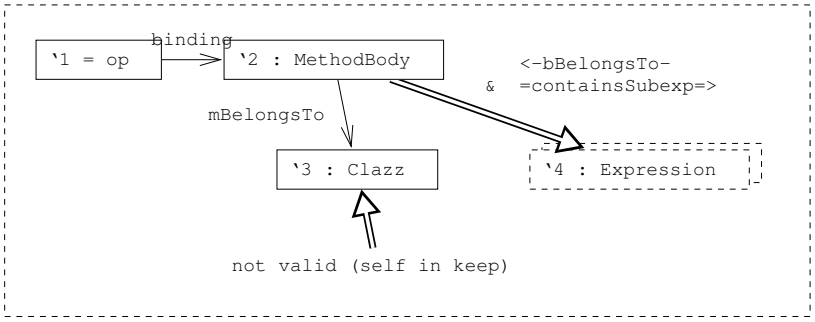
```
 do
  choose
  when (make_abstract)
  then
   use absOp : Operation
   do
   pm_addAbstractDecl ( op, trg, out absOp )
   & pm_addAbstractDecl_param ( op, absOp )
   & pm_addStubEmpty ( op, trg )
   & pm_addStubReturn ( op, trg )
   end
  else
  pm_moveMethod ( body, trg )
  end
  & pm_removeUnguarded ( op, keep or trg )
  & choose
   when (use_supertype)
   then
   use cands : PGElement [0:n]
   do
    pm_superTypeCandidates ( src, trg, out cands )
    & pm_useSuperType ( cands, src, trg )
   end
  else
   skip
  end
 end
end;

transformation - pm_moveMethod
 ( body : MethodBody ; trg : Clazz) =
```



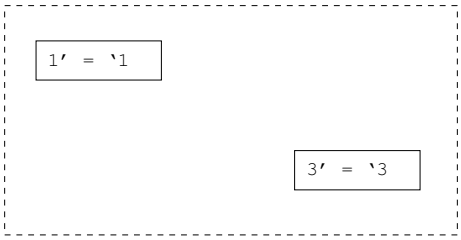

```
 ::=
```



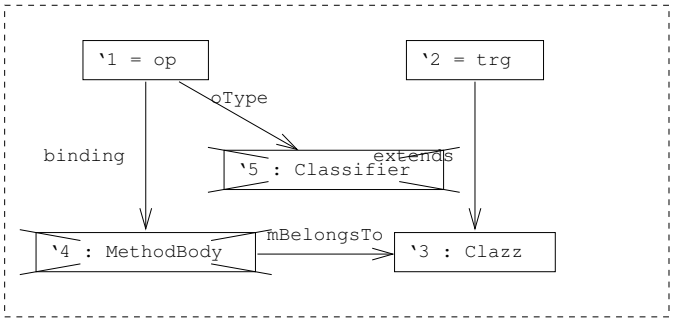

```
 transfer
 4'.visibility :=
  [ `4.visibility.ordinal > Protected.ordinal :: Protected
  | `4.visibility ]          ;
end;

transformation - pm_removeUnguarded
 ( op : Operation ; keep : Clazz [0:n]) * =
```



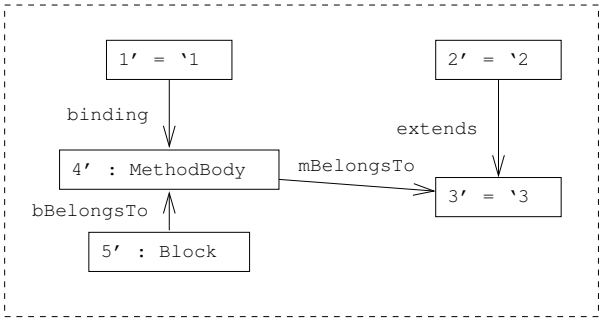

```
 ::=
```




```
end;

transformation - pm_addStubEmpty
 ( op : Operation ; trg : Clazz) * =
```

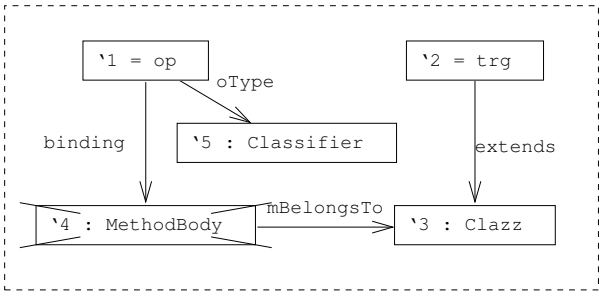


::=

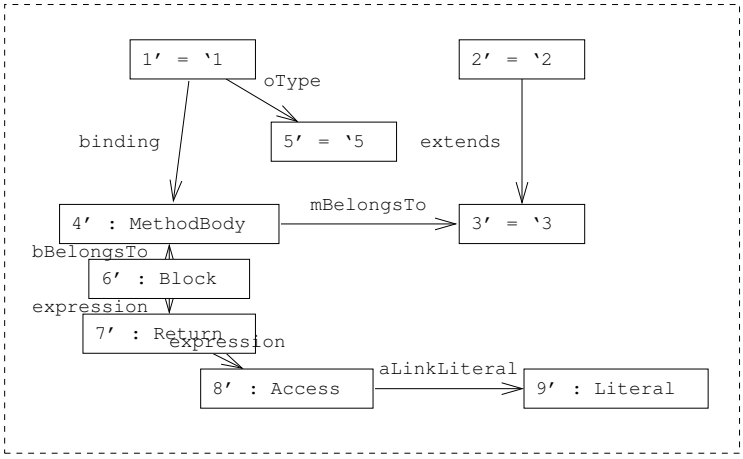


condition not '3.isAbstract;
end;

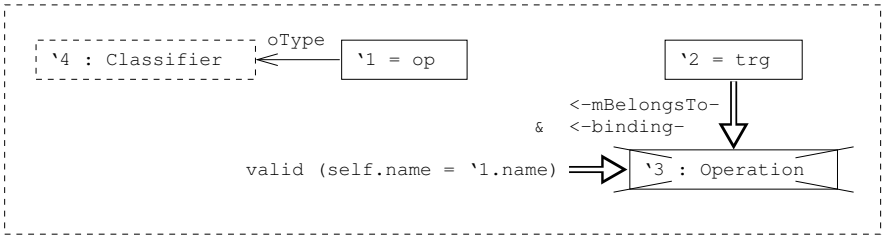
transformation - pm_addStubReturn
(op : Operation ; trg : Clazz) * =



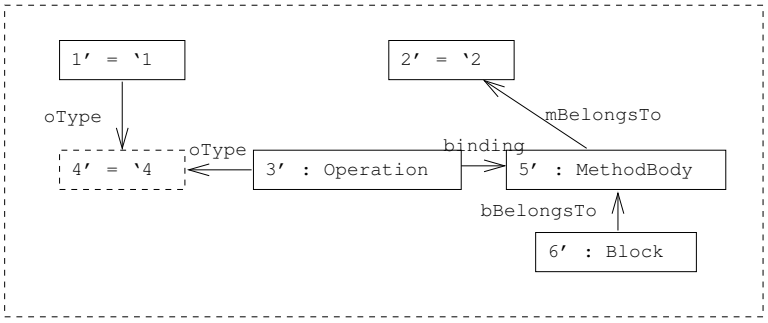
::=




```
 condition not `3.isAbstract;
 transfer 9'.val := "null";
end;

transformation - pm_addAbstractDecl
 ( op : Operation ; trg : Clazz ; out absOp : Operation) =
```

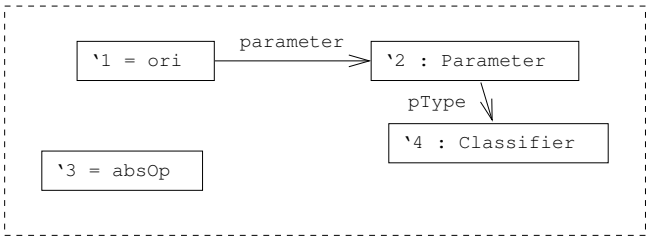



```
 ::=
```

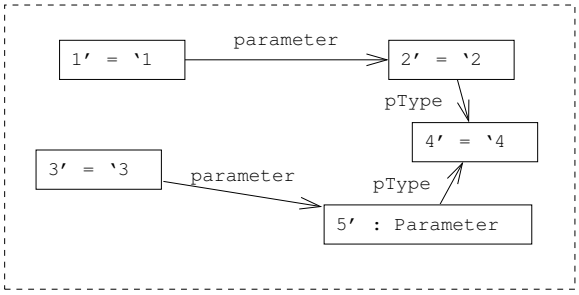



```
 embedding redirect <-cLink- from `1 to 3';
 transfer 3'.name := `1.name;
```

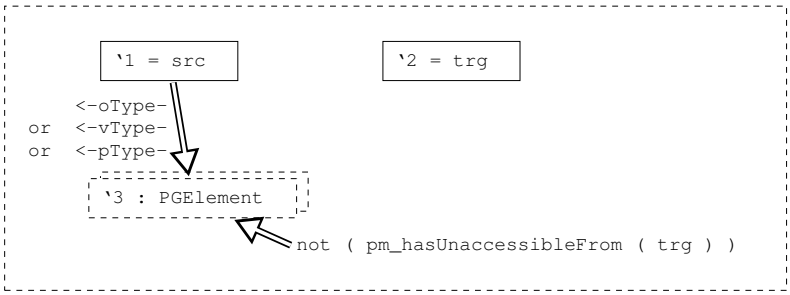
```
   3'.isAbstract := true;
   3'.oIsFinal := '1.oIsFinal;
   3'.oIsStatic := '1.oIsStatic;
   3'.visibility := '1.visibility;
   2'.isAbstract := true;
 return absOp := 3';
end;

transformation - pm_addAbstractDecl_param
 ( ori, absOp : Operation) * =
```



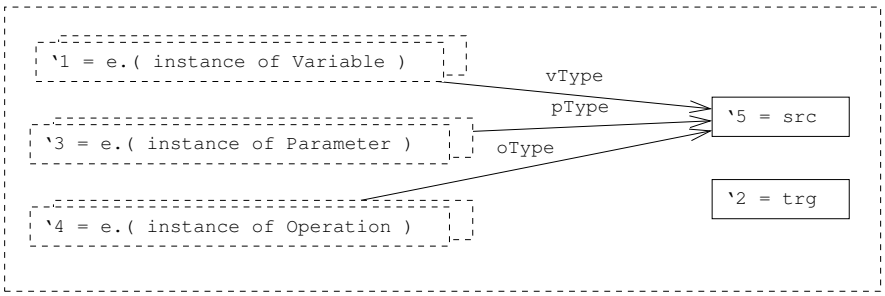

```
 ::=
```



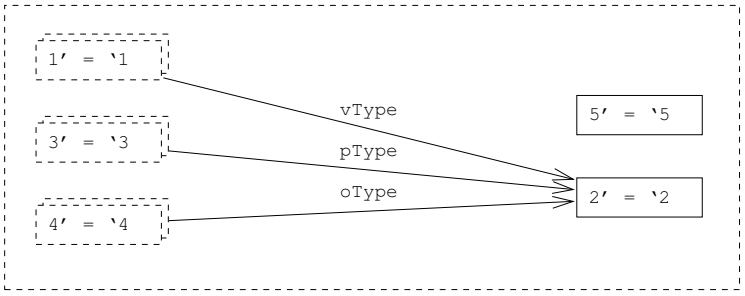

```
 transfer 5'.order := '2.order;
end;

query - pm_superTypeCandidates( src, trg : Clazz ; out e : PGElement [0:n])
=
```



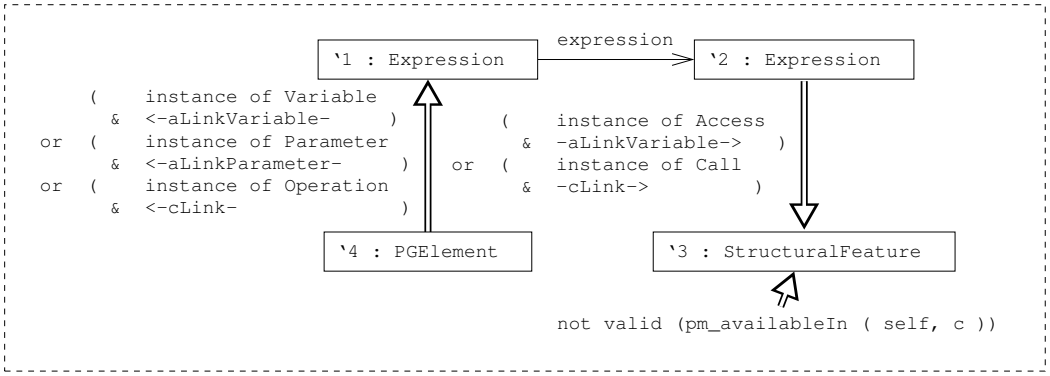
```
 return e := ‘3;
end;

transformation - pm_useSuperType
 ( e : PGElement [0:n] ; src, trg : Clazz) =
```




```
 ::=
```




```
end;

function - pm_sameSignature : ( o1, o2 : Operation) ->
 boolean =
 (o1.name = o2.name)
 and for all trg_op_param := o2.-parameter-> ::
  exist own_param := o1.-parameter-> ::
   (trg_op_param.order = own_param.order)
   and (trg_op_param.-pType-> = own_param.-pType->)
  end
  end
end;

restriction - pm_hasUnaccessibleFrom( c : Clazz) : PGElement
=
 ‘4 in
```




```
  end;

  function - pm_availableIn : ( e : PGElement ; c : Clazz) ->
   boolean =
   [ e is instance of Variable ::
      c in (e : Variable.-vBelongsToClass->.( <-extends- * ))
   | e is instance of Operation ::
      c in (e : Operation.-binding->.-mBelongsTo->.( <-extends- * ))
   | false ]
  end;

 end;

 [...]
(* Constructive methods omitted *)

end.
```