

Specifications of Implemented Refactorings

Max Schäfer, Tomáš Kočíský

August 12, 2010

This document collects the pseudo-code specifications of all refactoring implemented in our engine. **Note:** This is work in progress; some specifications are missing, and not all implementations agree completely with the specifications.

1 Pseudocode Conventions

We give our specifications in generic, imperative pseudocode. Parameters and return values are informally typed, with syntax tree nodes having one of the types from Fig. 1. Additionally, we use an ML-like **option** type with constructors **None** and **Some** for functions that may or may not return a value.

Where convenient, we make use of ML-like lists, with list literals of the form $[1; 2; 3]$ and $|xs|$ indicating the length of list xs .

The names of refactorings are written in SMALL CAPS, whereas utility functions appear in **monospace**. A list of utility functions with brief descriptions is given in Fig. 2. An invocation of a refactoring is written with floor-brackets $[LIKE\ THIS]()$ to indicate that any language extensions used in the output program produced by the refactoring should be eliminated before proceeding.

We write $A <: B$ to mean that type A extends or implements type B , and $m <: m'$ to mean that method m overrides method m' .

2 The Refactorings

2.1 Convert Anonymous to Local

This refactoring converts an anonymous class to a local class. Implemented in `TypePromotion/AnonymousClassToLocalClass.jrag`; see Algorithm 1.

2.2 Convert Anonymous to Nested

This refactoring converts an anonymous class to a member class. Implemented in `TypePromotion/AnonymousClassToMemberClass.jrag`; see Algorithm 2.

Note: the implementation additionally handles the case where A occurs in a field initialiser.

Algorithm 1 CONVERT ANONYMOUS TO LOCAL($A : AnonymousClass, n : Name$) : *LocalClass*

Require: Java

Ensure: Java \cup locked names

- 1: $c \leftarrow$ class instance expression containing A
 - 2: $d \leftarrow [\text{EXTRACT TEMP}](c, \text{unCapitalise}(n))$ – not possible to do!!!
 - 3: $b \leftarrow$ enclosing body declaration of d
 - 4: $\text{lockNames}(b, n)$
 - 5: convert A to class named n , remove it from c
 - 6: $\text{INSERT TYPE}(b, A)$
 - 7: lock type access of c to A
 - 8: $\text{INLINE TEMP}(d)$ – but without checks (TODO?)
 - 9: **return** A
-

Algorithm 2 CONVERT ANONYMOUS TO NESTED($A : AnonymousClass, n : Name$) : *MemberType*

Require: Java

Ensure: Java

- 1: $L \leftarrow \text{CONVERT ANONYMOUS TO LOCAL}(A, n)$
 - 2: **return** $\text{CONVERT LOCAL TO MEMBER CLASS}(L)$
-

2.3 Convert Local to Member Class

This refactoring converts a local class to a member class. Implemented in `TypePromotion/LocalClassToMemberClass.jrag`; see Algorithms 3, 4, 5.

2.4 Extract Class

This refactoring extracts some fields of a class into a newly created member class. Implemented in `ExtractClass/ExtractClass.jrag`; see Algorithm 6.

Initializer evaluation is order independent for example if we can invert their order without breaking name and dataflow dependencies.

This is only a bare-bones specification. The implementation additionally allows to encapsulate the extracted fields, and to move the wrapper class W to the toplevel.

2.5 Extract Constant

This refactoring extracts a constant expression into a field. Implemented in `ExtractTemp/ExtractConstant.jrag`; see Algorithm 9.

An expression is extractible if its type is not `void`, it is not a reference to a type or package, and it is not the keyword `super`; furthermore, it cannot be on the right-hand side of a dot.

Algorithm 3 CONVERT LOCAL TO MEMBER CLASS($L : LocalClass$) : $MemberType$

Require: Java

Ensure: Java \cup locked names, fresh variables

```

1:  $A \leftarrow$  enclosing type of  $L$ 
2: closeOverTypeVariables( $L$ )
3: closeOverLocalVariables( $L$ )
4: if  $L$  is in static context then
5:   make  $L$  static
6: end if
7: lockNames(name( $L$ ))
8: lock all names in  $L$ 
9: remove  $L$  from its declaring method
10: INSERT TYPE( $A, L$ )

```

Algorithm 4 **closeOverTypeVariables**($L : LocalClass$)

```

1:  $m \leftarrow$  empty map
2:  $U \leftarrow$  accesses to  $L$ 
3: for all accesses  $V$  to type variables  $T$  of the enclosing body declaration do
4:   if  $m(T)$  undefined then
5:     create new type variable  $T'$  with same bounds as  $T$ 
6:     add  $T'$  as type parameter to  $L$ 
7:      $m(T) \leftarrow T'$ 
8:     for all  $u \in U$  do
9:       add locked access to  $T$  as type argument to  $u$ 
10:    end for
11:   end if
12:   lock  $V$  onto  $m(T)$ 
13: end for

```

Algorithm 5 `closeOverLocalVariables($L : LocalClass$)`

```
1:  $m \leftarrow$  empty map
2: for all accesses  $v$  to local variables  $x$  of enclosing body declaration do
3:   if  $m(x)$  undefined then
4:     create private final field  $f$  of same type as  $x$ 
5:     add  $f$  to  $L$ 
6:      $m(v) \leftarrow f$ 
7:   for all constructors  $c$  of  $L$  do
8:     create new parameter  $p$  of same type and name as  $x$ 
9:     insert  $p$  as first parameter of  $c$ 
10:    if  $c$  is chaining constructor then
11:      add access to  $p$  as parameter to chaining invocation
12:    else
13:      insert assignment from  $p$  to  $f$  as first statement in  $c$ 
14:    end if
15:  end for
16:  for all instantiations  $i$  of  $L$  do
17:    insert access to  $x$  as first argument to  $i$ 
18:  end for
19: end if
20: lock  $v$  onto  $m(x)$ 
21: end for
```

The *effective type* of an expression e is the same as the type of e , except when the type of e is an anonymous class, in which case the effective type is its superclass, or when the type of e is a captured type variable, in which case the effective type is its upper bound.

2.6 Extract Method

See ECOOP 2009 publication. (TODO)

2.7 Extract Temp

This refactoring extracts an expression into a local variable. Implemented in `ExtractTemp/ExtractTemp.jrag`; see Algorithms 10, 11, 12, 13.

2.7.1 Insert Local Variable

The refactoring inserts a local variable before a given statement. Implemented in `ExtractTemp/IntroduceUnusedLocal.jrag`.

Algorithm 6 EXTRACT CLASS(C : *Class*, fs : **list** *Field*, n : *Name*, fn : *Name*)

Require: Java

Ensure: Java \cup locked names, locked dataflow, first-class array init

```

1:  $v \leftarrow$  maximum visibility of any of the  $fs$ 
2:  $W \leftarrow$  new static class of name  $n$  with visibility  $v$ 
3: INSERT TYPE( $C$ ,  $W$ )
4:  $w \leftarrow$  new field of type  $W$  and name  $fn$ , initialised to a new instance of  $W$ 
5: INSERT FIELD( $C$ ,  $w$ )
6: for all  $f \in fs$  do
7:   assert  $f$  is not static
8:   for all uses  $v$  of  $f$  do
9:     qualify  $v$  with a locked access to  $w$ 
10:  end for
11:  if  $f$  has initialiser then
12:    split field declaration and initializer, leaving initializer in initializer
    block after
13:  end if
14:  remove  $f$ 
15:  INSERT FIELD( $W$ ,  $f$ )
16: end for
17:  $inits \leftarrow$  {initializers of  $fs$ }
18: MOVE INITS TOGETHER( $inits$ ,  $w$ ) or ID()
19: merge consecutive  $inits$  to common initializer blocks
20: PASSINITS TO CONSTRUCTOR( $inits$ ,  $w$ ) or ID()

```

Algorithm 7 MOVE INITS TOGETHER($inits$: **list** *InitializerBlock*, $after$: *Field*)

Require: Java

Ensure: Java

```

1: for all  $init \in inits$  do
2:   lock names and dataflow in  $init$ 
3:   remove  $init$  and insert it after already moved initializers (possibly  $after$ )
4:   unlock names and dataflow in  $init$ 
5: end for

```

Algorithm 8 PASSINITS TO CONSTRUCTOR(*inits* : list InitializerBlock, *w* : Field)

```

1: assert all inits are in one initializer block
2: index  $\leftarrow$  position of w
3: vars  $\leftarrow$  []
4: for all init in inits do
5:   v  $\leftarrow$  [EXTRACT TEMP](left side of init, fresh name, index)
6:   vars  $\leftarrow$  v : vars
7:   index  $\leftarrow$  index + 1
8: end for
9: for all var in vars do
10:  INLINE TEMP(var)
11: end for
12: in W create a constructor for initializing all fields
13: change the constructor call for w to initialize the fields and remove inits

```

Algorithm 9 EXTRACT CONSTANT(*e* : Expr, *n* : Name)

Require: Java

Ensure: Java \cup locked names, locked dataflow

```

1: assert e is extractible
2: A  $\leftarrow$  enclosing type of e
3: t  $\leftarrow$  effective type of e
4: f  $\leftarrow$  new private (public if A is an interface) static final field of type t and name n
5: INSERT FIELD(A, f)
6: lock names, flow, and synchronisation of e
7: set initialiser of f to e
8: replace e with locked access to f

```

Algorithm 10 EXTRACT TEMP(*e* : Expr, *n* : Name) : LocalVar

Require: Java

Ensure: Java \cup locked names, locked dataflow

```

1: t  $\leftarrow$  effective type of e
2: v  $\leftarrow$  new local variable of type t and name n
3: s  $\leftarrow$  enclosing statement of e
4: INSERT LOCAL VARIABLE(s, v)
5: EXTRACT ASSIGNMENT(v, e)
6: MERGE DECLARATION(v)
7: return v

```

Algorithm 11 INSERT LOCAL VARIABLE($s : Stmt, v : LocalVar$)

Require: Java

Ensure: Java \cup locked names

- 1: $b \leftarrow$ enclosing block of s
 - 2: **assert** variable v can be introduced into block b
 - 3: `lockNames(b, n)`
 - 4: insert v before s
-

2.7.2 Extract Assignment

This refactoring extracts an expression into an assignment to a local variable. Implemented in `ExtractTemp/ExtractAssignment.jrag`.

Algorithm 12 EXTRACT ASSIGNMENT($v : LocalVar, e : Expr$) : *Assignment*

Require: Java

Ensure: Java \cup locked dependencies

- 1: **assert** e is extractible
 - 2: $a \leftarrow$ new assignment from e to v
 - 3: **if** e is in expression statement **then**
 - 4: replace e with a
 - 5: **else**
 - 6: $s \leftarrow$ enclosing statement of e
 - 7: lock all names in e
 - 8: insert a before s
 - 9: replace e with locked access to v
 - 10: **end if**
 - 11: **return** a
-

2.7.3 Merge Variable Declaration

This refactoring merges a variable declaration with the assignment immediately following it, if that assignment is an assignment to the same variable. Implemented in `ExtractTemp/MergeVarDecl.jrag`.

2.8 Inline Constant

This refactoring inlines a constant field into all its uses. Implemented in `InlineTemp/InlineConstant.jrag`; see Algorithms 14, 15, 16.

2.9 Inline Method

See ECOOP 2009 publication. (TODO)

Algorithm 13 MERGE VARIABLE DECLARATION($v : LocalVar$)

Require: Java \ multi-declarations – TODO no checks for this in the implementation and no test for this, refactorings are not stand alone enough imo

Ensure: Java

- 1: **if** v has initialiser **then**
 - 2: **return**
 - 3: **end if**
 - 4: $s \leftarrow$ statement following v
 - 5: **if** s is assignment to v **then**
 - 6: make RHS of s the initialiser of v
 - 7: remove s
 - 8: **end if**
-

Algorithm 14 INLINE CONSTANT($f : Field$)

Require: Java \ implicit assignment conversion

Ensure: Java

- 1: **for all** uses u of f **do**
 - 2: INLINE CONSTANT(u)
 - 3: **end for**
 - 4: REMOVE FIELD(f)
-

Algorithm 15 INLINE CONSTANT($u : FieldAccess$)

Require: Java

Ensure: Java \cup locked dependencies

- 1: $f \leftarrow$ field accessed by u
 - 2: **assert** f is **final** and **static**, and has an initialiser
 - 3: $e \leftarrow$ locked copy of the initialiser of f
 - 4: **assert** if u is qualified, then its qualifier is a pure expression
 - 5: replace u with e , discarding its qualifier if any
-

Algorithm 16 REMOVE FIELD($f : Field$)

Require: Java

Ensure: Java

- 1: **if** f is not used and if it has an initialiser, it is pure **then**
 - 2: remove f
 - 3: **end if**
-

2.10 Inline Temp

This refactoring inlines a local variable into all its uses. Implemented in `InlineTemp/InlineTemp.jrag`; see Algorithms 17, 18, 19, 20

Algorithm 17 `INLINE TEMP($d : LocalVar$)`

Require: Java

Ensure: Java

- 1: $a \leftarrow \lfloor \text{SPLIT DECLARATION} \rfloor(d)$
 - 2: $\lfloor \text{INLINE ASSIGNMENT} \rfloor(a)$
 - 3: $\lfloor \text{REMOVE DECL} \rfloor(v)$
-

Algorithm 18 `SPLIT DECLARATION($d : LocalVar$) : option Assignment`

Require: Java \ compound declarations

Ensure: Java \cup locked names, first-class array init

- 1: **if** d has initialiser **then**
 - 2: $x \leftarrow$ variable declared in d
 - 3: $a \leftarrow$ new assignment from initialiser of d to x
 - 4: insert a as statement after d
 - 5: remove initialiser of d
 - 6: **return** Some a
 - 7: **else**
 - 8: **return** None
 - 9: **end if**
-

2.11 Introduce Factory

This refactoring introduces a static factory method as a replacement for a given constructor, and updates all uses of the constructor to use this method instead. Implemented in `IntroduceFactory/IntroduceFactory.jrag`; see Algorithm 21

We use `createFactoryMethod` (implemented in `util/ConstructorExt.jrag`) to create the factory method corresponding to constructor cd and insert it into the host type of cd . The factory method has the same signature as cd , but it has its own copies of all type variables of the host type used in cd .

2.12 Introduce Indirection

This refactoring creates a static method m' in type B that delegates to a method m in type A . Implemented in `IntroduceIndirection/IntroduceIndirection.jrag`; see Algorithm 22.

Algorithm 19 INLINE ASSIGNMENT($a : \text{Assignment}$)

Require: Java \setminus implicit assignment conversion**Ensure:** Java \cup locked dependencies

```
1:  $x \leftarrow$  LHS of  $a$ 
2: assert  $x$  refers to local variable
3:  $U \leftarrow$  all  $u$  such that  $a$  is a reaching definition of  $u$ 
4: for all  $u \in U$  do
5:   assert  $a$  is the only reaching definition of  $u$ 
6:   assert  $u$  is not an lvalue
7:   assert  $u, a$  are in same body declaration
8:   replace  $u$  with a locked copy of the RHS of  $a$ 
9: end for
10: if  $U \neq \emptyset$  then
11:   remove  $a$ 
12: end if
```

Algorithm 20 REMOVE DECL($d : \text{LocalVar}$)

Require: Java \setminus compound declarations**Ensure:** Java

```
1: if  $d$  is not used and has no initialiser then
2:   remove  $d$ 
3: end if
```

Algorithm 21 INTRODUCE FACTORY($cd : \text{ConstructorDecl}$)

Require: Java**Ensure:** Java \cup locked names

```
1:  $f \leftarrow$  static factory method for  $cd$ 
2: for all uses  $u$  of  $cd$  and its parameterised copies do
3:   if  $u$  is a class instance expression without anonymous class and it is not
     in  $f$  then
4:     replace  $u$  with a call to  $f$ 
5:   end if
6: end for
```

Algorithm 22 INTRODUCE INDIRECTION($m : \text{Method}, B : \text{ClassOrInterface}$)

Require: Java

Ensure: Java \cup locked names, **return void**

- 1: **assert** B is non-library
 - 2: $fn \leftarrow$ fresh method name
 - 3: $m' \leftarrow$ copy of m with locked names and empty body
 - 4: set name of m' to fn
 - 5: $xs \leftarrow$ locked accesses to parameters of m'
 - 6: set body of m' to **return** $m(xs)$;
 - 7: INSERT METHOD(**hostType**(m), m')
 - 8: MAKE METHOD STATIC(m')
 - 9: MOVE STATIC METHOD(m', B)
-

2.13 Introduce Parameter

This refactoring turns an expression into a parameter of the surrounding method. Implemented in `ChangeMethodSignature/IntroduceParameter.jrag`; see Algorithm 23.

Algorithm 23 INTRODUCE PARAMETER($e : \text{Expr}, n : \text{Name}$)

Require: Java

Ensure: Java \cup locked names

- 1: **assert** n is a valid name
 - 2: **assert** e is extractible and constant
 - 3: **assert** e appears within a method m
 - 4: **assert** m is not overridden by and does not override any other methods
 - 5: **assert** m has no parameter or local variable n
 - 6: lockMethodCalls(**name**(m))
 - 7: $t \leftarrow$ effective type of e
 - 8: $p \leftarrow$ new parameter of type t and name n
 - 9: insert p as the first parameter of m
 - 10: replace e with locked access to p
 - 11: **for all** calls c to m **do**
 - 12: insert a locked copy of e as first argument of c
 - 13: **end for**
-

2.14 Introduce Parameter Object

This refactoring wraps a set P of parameters of a method m into a single parameter n of type w , where w is a newly created wrapper class containing fields corresponding to all the parameters in P . Implemented in `IntroduceParameterObject/IntroduceParameterObject.jrag`; see Algorithm 24.

Algorithm 24 INTRODUCE PARAMETER OBJECT(m : *Method*, P : *set Parameter*, w : *set Name*, n : *set Name*)

Require: Java \ variable arity parameters

Ensure: Java \cup locked names

```

1: assert  $m$  has a body
2: assert the parameters in  $P$  are in contiguous positions  $i, \dots, i+k$ 
3:  $W \leftarrow$  new class containing fields for all the  $P$  and a standard constructor to initialise them
4: INSERT TYPE(hostType( $m$ ),  $W$ )
5: lockMethodCalls(name( $m$ ))
6: for all relatives  $r$  of  $m$  do
7:   assert  $r$  has no parameter or local variable with name  $n$ 
8:    $[p_1; \dots; p_n] \leftarrow$  parameters of  $r$ 
9:    $p \leftarrow$  new parameter of type  $W$  and name  $n$ 
10:  replace parameters  $p_i, \dots, p_{i+k}$  with  $p$ 
11:  for all  $j \in \{i, \dots, i+k\}$  do
12:     $v_j \leftarrow$  new variable of same name, type, and finality as  $p_j$ 
13:    insert assignment from  $p.f_j$  to  $v_j$  at beginning of  $m$ 
14:  end for
15:  for all calls  $c$  to  $r$  do
16:     $[a_1; \dots; a_n] \leftarrow$  arguments of  $c$ 
17:    replace arguments  $a_i, \dots, a_{i+k}$  with new  $W(a_i, \dots, a_{i+k})$ 
18:  end for
19: end for

```

Note that we need to perform the transformation for all relatives of m , *i.e.* for all methods r such that there exists a method m' with $m <:^* m'$ and $r <:^* m'$. We also lock all calls to methods of the same as m in the whole program; this ensures that if overloading resolution changes due to the transformation, the name binding framework will insert appropriate casts to rectify the situation.

Note: the implementation actually: eliminates variable arity parameter for this method and adjusts all calls; does not require p_i to be contiguous and adds new argument at the beginning. (This can be unsound for parameters with side effects!!!)

2.15 Move Inner To Toplevel

This refactoring converts a member type to a toplevel type. Implemented in `TypePromotion/MoveMemberTypeToToplevel.jrag`; see Algorithms 25, 26, 27.

Algorithm 25 MOVE MEMBER TYPE TO TOPLEVEL($M : MemberType$)

Require: Java

Ensure: Java \cup locked names

- 1: **if** M is not static **then**
 - 2: `[MAKE TYPE STATIC](M)`
 - 3: **end if**
 - 4: $p \leftarrow \text{hostPkg}(M)$
 - 5: lock all names in M
 - 6: remove M from its host type
 - 7: `INSERT TYPE(p, M)`
-

Algorithm 26 INSERT TYPE($p : Package, T : ClassOrInterface$)

Require: Java

Ensure: Java \cup locked names

- 1: **assert** no type or subpackage of same name as T in p
 - 2: `lockNames(name(T))`
 - 3: remove modifiers `static`, `private`, `protected` from T
 - 4: insert T into p
-

2.16 Move Instance Method

This refactoring moves a method into a variable, which is either a parameter of that method or an accessible field. Implemented in `Move/MoveMethod.jrag`.

Algorithm 27 MAKE TYPE STATIC($M : \text{MemberType}$)

Require: Java

Ensure: Java \cup **with**, locked names

```
1:  $[A_n; \dots; A_1] \leftarrow$  enclosing types of  $M$ 
2: for all  $i \in \{1, \dots, n\}$  do
3:    $f \leftarrow$  new field of type  $A_i$  with name this $\$i$ 
4:   INSERT FIELD( $M, f$ )
5:   for all constructors  $c$  of  $M$  do
6:      $p \leftarrow$  parameter of type  $A_i$  with name this $\$i$ 
7:     assert no parameter or variable this $\$i$  in  $c$ 
8:     insert  $p$  as first parameter of  $c$ 
9:     if  $c$  is chaining then
10:      add this $\$i$  as first argument of chaining call
11:     else
12:       $a \leftarrow$  new assignment of  $p$  to  $f$ 
13:      insert  $a$  after super call
14:     end if
15:   end for
16: end for
17: for all constructors  $c$  of  $M$  do
18:   for all non-chaining invocations  $u$  of  $c$  do
19:      $es \leftarrow$  enclosing instances of  $u$ 
20:     assert  $|es| = n$ 
21:     insert  $es$  as initial arguments to  $u$ 
22:     discard qualifier of  $u$ , if any
23:   end for
24: end for
25: if  $M$  not in inner class then
26:   put modifier static on  $M$ 
27: end if
28: for all non-static callables  $m$  of  $M$  do
29:   if  $m$  has a body then
30:     surround body of  $m$  by
31:     with(this $\$n$ , ..., this $\$1$ , this) {...}
32:   end if
33: end for
```

Algorithm 28 MOVE METHOD($m : \text{InstanceMethod}, v : \text{Variable}$)

Require: Java

Ensure: Java \cup locked names, **return void**, fresh variables, demand **final**

```
1: assert  $v$  is either a parameter of  $m$  or a field
2:  $T \leftarrow$  type of  $v$ 
3: assert  $T$  is a non-library class
4: assert  $m$  has a body and is not from library
5:  $m' \leftarrow$  copy of  $m$  with synchronized removed and all names locked
6:  $xs \leftarrow$  list of locked accesses to parameters of  $m$ 
7: if  $v$  is a parameter then
8:    $i \leftarrow$  position of  $v$  in parameter list of  $m$ 
9:   remove  $i$ th parameter from  $m'$ 
10:  remove  $i$ th element of  $xs$ 
11: else
12:    $i \leftarrow 0$ 
13: end if
14:  $v' \leftarrow$  final local variable declaration with same name and type as  $v$ , initialised to this
15: insert  $v'$  as first statement into  $m'$ 
16: lock all uses of  $v$  inside  $m'$  to  $v'$ 
17:  $qs \leftarrow []$ 
18: for all enclosing classes  $C$  of  $m$  do
19:    $p_C \leftarrow$  demand final parameter with fresh name, of type  $C$ 
20:   make  $p_C$  the  $i$ th parameter of  $m'$ 
21:    $e \leftarrow$  access to  $C.\text{this}$ 
22:   insert  $e$  as  $i$ th element into  $xs$ 
23:    $qs \leftarrow [p_C] :: qs$ 
24: end for
25: wrap body of  $m'$  into with( $qs$ ) {...}
26: set body of  $m$  to return  $\llbracket v \rrbracket. \llbracket m \rrbracket(xs)$ ;
27: INSERT METHOD( $T, m'$ )
28: eliminate with statement in  $m'$ 
29: INLINE TEMP( $v'$ )
30: for all  $p_C$  do
31:   REMOVE PARAMETER( $p_C$ ) or ID()
32: end for
```

2.17 Move Members

In order to move Field, static methods, and member types, we simply lock all references to them, as well as all names contained in them, and (for fields) the flow dependencies of their initialiser, and then move them inside the AST.

2.18 Promote Temp to Field

This refactoring turns a local variable into a field. Implemented in `PromoteTempToField/PromoteTempToField`.

Algorithm 29 PROMOTE TEMP TO FIELD($d : LocalVar$)

Require: Java

Ensure: Java \cup locked dependencies

- 1: $\lfloor \text{SPLIT DECLARATION} \rfloor(d)$
 - 2: $d' \leftarrow$ new **private** field of same type and name as d
 - 3: make d' **static** if d is in static context
 - 4: $\lfloor \text{INSERT FIELD} \rfloor(\text{hostType}(d), d')$
 - 5: **for all** uses u of d **do**
 - 6: lock u onto d'
 - 7: lock reaching definitions of u
 - 8: **end for**
 - 9: REMOVE DECL(d)
-

Algorithm 30 INSERT FIELD($T : ClassOrInterface, d : Field$)

Require: Java

Ensure: Java \cup locked names

- 1: **assert** T has no local field with same name as d
 - 2: **assert** d has no initialiser
 - 3: **assert** if T is inner and d is static, then d is a constant
 - 4: lockNames(name(d))
 - 5: insert field d into T
-

2.19 Pull Up

This refactoring pulls up a method m from its host class B to the super class A . Implemented in `PullUp/PullUpMethod.jrag`.

TODO: explain translation of type variables; this is basically a right-inverse of the type variable substitution that happens when inheriting a method

Note that INSERT METHOD ensures that the inserted method is not called from anywhere.

Algorithm 31 PULL UP METHOD($m : \text{Method}$)

Require: Java**Ensure:** Java \cup locked names

- 1: **assert** the host type of m B is a non-library class
 - 2: **assert** the superclass A of B is also non-library
 - 3: $m' \leftarrow$ copy of m with locked names
 - 4: translate type variables in m' from B to A
 - 5: **Insert Method**(A, m')
 - 6: remove m from B
-

2.20 Push Down

This refactoring pushes a method down to all subclasses of its defining class. Implemented in `PushDown/PushDownMethod.jrag`.

Types that inherit a method m include the host type of m .

Algorithm 32 TRIVIALY OVERRIDE($B : \text{Type}, m : \text{VirtualMethod}$) : *option MethodCall*

Require: Java \setminus implicit method modifiers**Ensure:** Java + locked names, **return void**

- 1: **assert** m is not **final**
 - 2: **if** m not a member method of B **then**
 - 3: **return** None
 - 4: **end if**
 - 5: $m' \leftarrow$ copy of m with locked names
 - 6: **if** m is **abstract** **then**
 - 7: insert method m' into B
 - 8: **return** None
 - 9: **else**
 - 10: $xs \leftarrow$ list of locked accesses to parameters of m'
 - 11: $c \leftarrow \text{super}.m(xs)$
 - 12: set body of m' to **return** c ;
 - 13: insert method m' into B
 - 14: **return** Some c
 - 15: **end if**
-

2.21 Rename

This family of refactorings is used for renaming named program entities. Implemented in `Renaming/`.

Refactoring `RENAME FIELD` changes the name of a field f to n . It ensures that n is indeed a valid name and that the host type of f contains no other

Algorithm 33 REMOVE METHOD($m : Method$)

Require: Java**Ensure:** Java

```
1: assert ( $\text{uses}(m) \cup \text{calls}(m) \setminus \text{below}(m) = \emptyset$ )
2:  $o \leftarrow \{m' \mid m <: m'\}$ 
3: if  $o \neq \emptyset \wedge \forall m' \in o. m'$  is abstract then
4:   for all types  $B$  that inherit  $m$  do
5:     MAKE TYPE ABSTRACT( $B$ )
6:   end for
7: end if
8: remove  $m$ 
```

Algorithm 34 MAKE METHOD ABSTRACT($m : Method$)

Require: Java**Ensure:** Java

```
1: assert  $m$  is not native, static, private, nor final
2: assert there are no static calls to  $m$  (e.g., super-call)
3: for all types  $B$  that inherit  $m$  do
4:   MAKE TYPE ABSTRACT( $B$ )
5: end for
6: make  $m$  abstract
```

Algorithm 35 MAKE TYPE ABSTRACT($T : Type$)

Require: Java**Ensure:** Java

```
1: if  $T$  is interface then
2:   return
3: end if
4: assert  $T$  is class and never instantiated
5: make  $T$  abstract
```

Algorithm 36 PUSH DOWN VIRTUAL METHOD($m : VirtualMethod$)

Require: Java**Ensure:** Java \cup locked names

```
1: for all types  $B <: \text{hostType}(m)$  do
2:    $c \leftarrow \lfloor \text{TRIVIALLY OVERRIDE} \rfloor(B, m)$ 
3:   if  $c \neq \text{None}$  then
4:     INLINE METHOD( $c$ )
5:   end if
6: end for
7: REMOVE METHOD( $m$ )
8:   or MAKE METHOD ABSTRACT( $m$ )
9:   or ID()
```

Algorithm 37 RENAME FIELD($f : \text{Field}, n : \text{Name}$)

Require: Java**Ensure:** Java \cup locked names

- 1: **assert** n is a valid name
 - 2: **assert** host type of f contains no other field of name n
 - 3: `lockNames($\{n, \text{name}(f)\}$)`
 - 4: set name of f to n
-

field called n . It then globally locks all accesses to variables, types, or packages named either n or `name(f)`, and changes the name of f to n .

Algorithm 38 RENAME LOCAL($v : \text{Local}, n : \text{Name}$)

Require: Java**Ensure:** Java \cup locked names

- 1: **assert** n is a valid name
 - 2: **assert** scope of v does not intersect scope of any other *Local* named n
 - 3: `lockNames(block(v), $\{n, \text{name}(f)\}$)`
 - 4: set name of v to n
-

Refactoring RENAME LOCAL changes the name of a local variable or parameter v to n . It ensures that n is indeed a valid name and that the renaming v to n will not violate the rule that scopes of local variables of the same name cannot be nested. It then again locks all accesses to variables, types, or packages named either n or `name(v)`, but only within the enclosing block of v , and changes the name of v to n .

Refactoring RENAME METHOD changes the name of a method m to n . It ensures that n is a valid name, then locks all calls to methods of name `name(m)` or n , and their overriding dependencies. Now it changes the names of all methods m' related to m (*i.e.*, such that m and m' both transitively override the same method), checking that the resulting program will be well-formed: in particular, there cannot be another local method with the same signature, and any methods that the renamed m' would override or hide must, in fact, be overridable or hidable by m' , and vice versa for methods that would override or hide m' . If there is a static import that only imports m' (and not also another static member of the surrounding class), then remove that import. We could, of course, try to adjust it, but changing imports is a tricky business.

Refactoring RENAME TYPE changes the name of a type T to n . It is fairly straightforward, except for the well-formedness checks and the treatment of import declarations.

Algorithm 39 RENAME METHOD($m : \text{Method}, n : \text{Name}$)

Require: Java

Ensure: Java \cup locked names, locked overriding

```
1: assert  $n$  is a valid name
2: lockMethodNames( $\{\text{name}(m), n\}$ )
3: lockOverriding( $\{\text{name}(m), n\}$ )
4: for all  $m'$  such that  $\exists m''.m <^* m'' \wedge m' <^* m''$  do
5:   assert  $m'$  is not native
6:    $s \leftarrow$  signature of  $m'$  after renaming
7:   assert host type of  $m'$  contains no local method of signature  $s$ 
8:   assert  $m'$  can override or hide any ancestor method of signature  $s$ 
9:   assert  $m'$  can be overridden or hidden by any descendant method of
      signature  $s$ 
10:  set name of  $m'$  to  $n$ 
11:  remove any static import of  $m'$  if it would become vacuous
12: end for
```

Algorithm 40 RENAME TYPE($T : \text{Type}, n : \text{Name}$)

Require: Java

Ensure: Java \cup locked names

```
1: assert  $n$  is a valid name
2: assert no native method is nested in  $T$ 
3: assert there is no nesting or enclosing type of name  $n$ 
4: assert if  $T$  is a toplevel type, there is no other toplevel type  $n$  in the
      enclosing package, and it has no subpackage of name  $n$ 
5: assert if  $T$  is a type parameter, there is no type parameter of name  $n$  in
      the parameter list where it occurs
6: lockNames( $\{\text{name}(T), n\}$ )
7: set name of  $T$  to  $n$ 
8: set names of constructors of  $T$  to  $n$ 
9: if  $T$  is public, change the name of its compilation unit to match
10: remove any single type import declaration of  $T$  that would clash with a
      visible type or with another import declaration
11: remove any static import of  $T$  if it would become vacuous
```

2.22 Self-Encapsulate Field

This refactoring makes a field private, rerouting all accesses to it through getter and setter methods. Implemented in `SelfEncapsulateField/SelfEncapsulateField.jrag`.

Algorithm 41 SELF-ENCAPSULATE FIELD($f : \text{Field}$)

Require: Java \ abbreviated assignments

Ensure: Java \cup locked names

```

1: create getter method  $g$  for  $f$ 
2: if  $f$  is not final, create setter method  $s$  for it
3: for all all uses  $u$  of  $f$  and its substituted copies do
4:   if  $u \notin \text{below}(g) \cup \text{below}(s)$  then
5:     if  $u$  is an rvalue then
6:       replace  $u$  with locked access to  $g$ 
7:     else
8:       if  $f$  is not final then
9:          $q \leftarrow$  qualifier of  $u$ , if any
10:         $r \leftarrow$  RHS of assignment for which  $u$  is LHS
11:        replace  $u$  with locked access to  $s$  on argument  $r$ , qualified with  $q$ 
           if applicable
12:      end if
13:    end if
14:  end if
15: end for

```

By “abbreviated assignment” we mean $x+=y$ and friends, as well as increment and decrement expressions. The language restriction tries to expand these into normal assignments, but may fail if the data flow is too complicated. If it succeeds, every lvalue will appear on the left hand side of a (simple) assignment.

Note that even when f is final there may still be assignments to f from within constructors; we cannot encapsulate these assignments, so we skip them.

3 Node Types

See Fig. 1. We also use the non-node type *Name* to represent names.

4 Utility Functions

See Fig. 2.

List of Algorithms

1	CONVERT ANONYMOUS TO LOCAL($A : \text{AnonymousClass}, n : \text{Name}$) :	
	<i>LocalClass</i>	2

Node Type	Description
<i>ClassOrInterface</i>	either a class or an interface; is a <i>Type</i>
<i>Field</i>	field declaration
<i>LocalVar</i>	local variable declaration
<i>MemberType</i>	type declared inside another type; is a <i>Type</i>
<i>Method</i>	method declaration
<i>MethodCall</i>	method call
<i>Package</i>	package
<i>Type</i>	type declaration
<i>VirtualMethod</i>	non- private instance method; is a <i>Method</i>

Figure 1: Node Types

Name	Description
below (n)	returns the set of all nodes below n in the syntax tree
calls (m)	returns all calls that may dynamically resolve to method m ; can be a conservative over-approximation
hostPkg (e)	returns the package of the compilation unit containing e
hostType (e)	returns the closest enclosing type declaration around e
lockMethodCalls (n)	locks all calls to methods named n anywhere in the program
lockNames (n)	locks all names anywhere in the program that refer to a declaration with name n
name (e)	returns the name of program entity e
uses (m)	returns all calls that statically bind to method m

Figure 2: Utility Functions

2	CONVERT ANONYMOUS TO NESTED($A : \text{AnonymousClass}, n : \text{Name}$) : MemberType	2
3	CONVERT LOCAL TO MEMBER CLASS($L : \text{LocalClass}$) : MemberType	3
4	closeOverTypeVariables($L : \text{LocalClass}$)	3
5	closeOverLocalVariables($L : \text{LocalClass}$)	4
6	EXTRACT CLASS($C : \text{Class}, fs : \text{list Field}, n : \text{Name}, fn : \text{Name}$)	5
7	MOVE INITS TOGETHER($inits : \text{list InitializerBlock}, after : \text{BodyDeclaration}$)	5
8	PASSINITS TO CONSTRUCTOR($inits : \text{list InitializerBlock}, w : \text{Field}$)	6
9	EXTRACT CONSTANT($e : \text{Expr}, n : \text{Name}$)	6
10	EXTRACT TEMP($e : \text{Expr}, n : \text{Name}$) : $\text{VariableDeclaration}$	6
11	INSERT LOCAL VARIABLE($s : \text{Stmt}, v : \text{LocalVar}$)	7
12	EXTRACT ASSIGNMENT($v : \text{LocalVar}, e : \text{Expr}$) : Assignment	7
13	MERGE VARIABLE DECLARATION($v : \text{LocalVar}$)	8
14	INLINE CONSTANT($f : \text{Field}$)	8
15	INLINE CONSTANT($u : \text{FieldAccess}$)	8
16	REMOVE FIELD($f : \text{Field}$)	8
17	INLINE TEMP($d : \text{LocalVar}$)	9
18	SPLIT DECLARATION($d : \text{LocalVar}$) : option Assignment	9
19	INLINE ASSIGNMENT($a : \text{Assignment}$)	10
20	REMOVE DECL($d : \text{LocalVar}$)	10
21	INTRODUCE FACTORY($cd : \text{ConstructorDecl}$)	10
22	INTRODUCE INDIRECTION($m : \text{Method}, B : \text{ClassOrInterface}$)	11
23	INTRODUCE PARAMETER($e : \text{Expr}, n : \text{Name}$)	11
24	INTRODUCE PARAMETER OBJECT($m : \text{Method}, P : \text{set Parameter}, w : \text{set Name}, n : \text{set Name}$)	12
25	MOVE MEMBER TYPE TO TOPLEVEL($M : \text{MemberType}$)	13
26	INSERT TYPE($p : \text{Package}, T : \text{ClassOrInterface}$)	13
27	MAKE TYPE STATIC($M : \text{MemberType}$)	14
28	MOVE METHOD($m : \text{InstanceMethod}, v : \text{Variable}$)	15
29	PROMOTE TEMP TO FIELD($d : \text{LocalVar}$)	16
30	INSERT FIELD($T : \text{ClassOrInterface}, d : \text{Field}$)	16
31	PULL UP METHOD($m : \text{Method}$)	17
32	TRIVIALY OVERRIDE($B : \text{Type}, m : \text{VirtualMethod}$) : option MethodCall	17
33	REMOVE METHOD($m : \text{Method}$)	18
34	MAKE METHOD ABSTRACT($m : \text{Method}$)	18
35	MAKE TYPE ABSTRACT($T : \text{Type}$)	18
36	PUSH DOWN VIRTUAL METHOD($m : \text{VirtualMethod}$)	18
37	RENAME FIELD($f : \text{Field}, n : \text{Name}$)	19
38	RENAME LOCAL($v : \text{Local}, n : \text{Name}$)	19
39	RENAME METHOD($m : \text{Method}, n : \text{Name}$)	20
40	RENAME TYPE($T : \text{Type}, n : \text{Name}$)	20
41	SELF-ENCAPSULATE FIELD($f : \text{Field}$)	21