Specifications of Implemented Refactorings

Max Schäfer, Tomáš Kočiský

August 13, 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 $\lfloor$LIKE THIS$\rfloor()$ 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.

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, 11.

2.4 Extract Class

This refactoring extracts some fields of a class into a newly created member class. Implemented in `ExtractClass/ExtractClass.jrag`; see Algorithms 4, 5, 6.

We can pass initializers to a constructor if they do not depend on values of previous initializers.

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 7.

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.

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

Implemented in `ExtractMethod/ExtractMethod.jrag`; see Algorithms 8, 9, 10, 11, 12, 13.

2.7 Extract Temp

This refactoring extracts an expression into a local variable. Implemented in `ExtractTemp/ExtractTemp.jrag`; see Algorithms 14, 15, 16, 17.

2.7.1 Insert Local Variable

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

2.7.2 Extract Assignment

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

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

Require: Java
Ensure: Java ∪ locked names

1: $c \leftarrow$ class instance expression containing A
2: $d \leftarrow \lfloor$EXTRACT TEMP$\rfloor(c, \texttt{unCapitalise}(n))$
3: $b \leftarrow$ enclosing body declaration of d
4: $\texttt{lockNames}(b, n)$
5: convert A to class named n, remove it from c
6: INSERT TYPE(b, A)
7: lock type access of c to A
8: INLINE TEMP(d)
9: **return** A

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

Require: Java
Ensure: Java

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

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

Require: Java
Ensure: Java ∪ locked names, fresh variables

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

Algorithm 4 EXTRACT CLASS(C : *Class*, *fs* : `list` *Field*, n : *Name*, *fn* : *Name*)

Require: Java
Ensure: Java ∪ 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: PASSINITSTOCONSTRUCTOR(*inits*, w) **or** ID()

Algorithm 5 MOVE INITS TOGETHER(*inits* : `list` InitializerBlock, *after* : *Field*)

Require: Java
Ensure: Java

1: **for all** *init* ∈ *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 6 PassInitsToConstructor(*inits* : `list` InitializerBlock, *w* : *Field*)

1: **assert** all *inits* are in one initializer block
2: *index* ← possition of *w*
3: *vars* ← [ ]
4: **for all** *init* in *inits* **do**
5: $v \leftarrow \lfloor$Extract Temp$\rfloor$(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 7 Extract Constant($e : Expr, n : Name$)

Require: Java
Ensure: Java ∪ locked names, locked dataflow

1: **assert** *e* is extractible
2: *A* ← enclosing type of *e*
3: *t* ← effective type of *e*
4: *f* ← 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 8 Extract Method(b: *Block*, i: *nat*, j: *nat*, n: *ident*): *Method*

Require:

Ensure:
1: $b' \leftarrow \lfloor$Extract Block$\rfloor(b, i, j)$
2: $a \leftarrow$ Introduce Anonymous Method(b')
3: Close over Variables(a)
4: Eliminate Reference Parameters(a)
5: **return** $\lfloor$Lift Anonymous Method$\rfloor(n, a)$

Algorithm 9 EXTRACT BLOCK(b: *Block*, i: *nat*, j: *nat*): *Block*

Require: no compound declarations
Ensure: locked names

1: $[s_0; \ldots; s_{n-1}] \leftarrow$ statements in b
2: **assert** $0 \leq i \leq j < n$
3: lock all variable and type names in b
4: **for all** $i \leq k \leq j$ **do**
5: **assert** s_k is not a `case` or `default`
6: **if** s_k declares a variable referenced after s_j **then**
7: SPLIT DECLARATION(s_k)
8: move s_k before s_i
9: **end if**
10: **end for**
11: $b' \leftarrow$ new block with statements $s_i, \ldots, s_j$
12: set statements of b to $s_0, \ldots, s_{i-1}, b', s_{j+1}, \ldots, s_{n-1}$
13: **return** b'

Algorithm 10 INTRODUCE ANONYMOUS METHOD(b: *Block*): *AnonymousMethod*

Require:
Ensure: locked control flow, locked names, `return void`, anonymous methods

1: lock control flow successors in b
2: $[e_1; \ldots; e_n] \leftarrow$ locked accesses to all uncaught checked exceptions thrown in b
3: **if** b can complete normally **then**
4: $c \leftarrow (() :$ `void throws` $e_1, \ldots, e_n \Rightarrow b)()$
5: replace b with c;
6: **else**
7: **if** b is in a method m **then**
8: $T \leftarrow$ locked access to return type of m
9: **else**
10: $T \leftarrow$ `void`
11: **end if**
12: $c \leftarrow (() :$ T `throws` $e_1, \ldots, e_n \Rightarrow b)()$
13: replace b with `return` c;
14: **end if**
15: **return** c

Algorithm 11 CLOSE OVER VARIABLES(a: *AnonymousMethod*)

Require:

Ensure: anonymous methods, `out` and `ref` parameters, locked names

$m \leftarrow$ body declaration enclosing a
$V \leftarrow \emptyset$; $Val \leftarrow \emptyset$; $Out \leftarrow \emptyset$; $Ref \leftarrow \emptyset$
for all variable accesses va in a **do**
  $v \leftarrow$ variable va binds to
  assert if va is a write, then v is not `final`
  if v is a local variable or parameter of m **then**
    $V \leftarrow V \cup \{v\}$
    if va has an incoming data flow edge **then**
      if $v \in Out$ **then**
        $Out \leftarrow Out \setminus \{v\}$
        $Ref \leftarrow Ref \cup \{v\}$
      else if $v \notin Ref$ **then**
        $Val \leftarrow Val \cup \{v\}$
      end if
    end if
    if va has an outgoing data flow edge **then**
      if $v \in Val$ **then**
        $Val \leftarrow Val \setminus \{v\}$
        $Ref \leftarrow Ref \cup \{v\}$
      else if $v \notin Ref$ **then**
        $Out \leftarrow Out \cup \{v\}$
      end if
    end if
  end if
end for
for all $v \in V$ **do**
  if $v \in Val \cup Out \cup Ref$ **then**
    $p \leftarrow$ new parameter with same name and type as v
    make p `ref` if $v \in Ref$, `out` if $v \in Out$
    add p as parameter to a
    add access to v as argument to a
  else
    $v' \leftarrow$ new local variable with same name and type as v
    add v' as local variable to a
  end if
end for
for all type parameters V of m used in a **do**
  add type parameter V' with same name and bounds as V to a
  add type argument V to a
end for

Algorithm 12 ELIMINATE REFERENCE PARAMETERS(a: *AnonymousMethod*): *AnonymousMethod*

Require: no implicit `return`
Ensure: anonymous methods

if a has `ref` or `out` parameters **then**
  assert a has a single `ref` or `out` parameter
  assert return type of a is `void`
  $x \leftarrow$ the `ref` or `out` parameter of a
  $v \leftarrow$ the variable access passed as argument into x
  replace a by v `=` a
  set return type of a to type of x
  replace every `return;` statement with `return` x`;`
  if x is a `ref` parameter or it is live at the entry of a **then**
    make x a value parameter
  else
    make x a local variable
    remove argument v
  end if
end if
return a

Algorithm 13 LIFT ANONYMOUS METHOD(n: *ident*, a: *AnonymousMethod*): *Method*

Require:
Ensure: locked names

assert a does not reference any local variables from surrounding body declaration
assert a has no `ref` or `out` parameters
lock all names in a
$\overline{e} \leftarrow$ argument list of a
$m \leftarrow$ turn a into method named n
make m `static` if a occurs in static context
$T \leftarrow$ innermost type surrounding a
assert T has no member method with same signature as m
assert no subtype of T declares a method that would override or hide m
lock all calls to methods named n
insert m into T
$c \leftarrow$ locked call of m on arguments $\overline{e}$
replace a with c
return m

Algorithm 14 EXTRACT TEMP(e : *Expr*, n : *Name*) : *LocalVar*

Require: Java
Ensure: Java ∪ 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 15 INSERT LOCAL VARIABLE(s : *Stmt*, v : *LocalVar*)

Require: Java
Ensure: Java ∪ 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

Algorithm 16 EXTRACT ASSIGNMENT(v : *LocalVar*, e : *Expr*) : *Assignment*

Require: Java
Ensure: Java ∪ 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 18, 19, 20.

2.9 Inline Method

This refactoring is inverse of Extract Method. Implemented in `InlineMethod/`; see Algorithms 21, 22, 23, 24, 25, 26, 27.

2.10 Inline Temp

This refactoring inlines a local variable into all its uses. Implemented in `InlineTemp/InlineTemp.jrag`; see Algorithms 28, 29, 30, 31.

2.11 Insert Method

This refactoring inserts a method into a type declaration. Implemented in `Move/InsertUnusedMethod.jrag`; see Algorithms 32, 33, 34.

2.12 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 35

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.

2.13 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 36.

Algorithm 17 MERGE VARIABLE DECLARATION(v : *LocalVar*)

Require: Java \ multi-declarations
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 18 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 19 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 20 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**

Algorithm 21 INLINE METHOD(m: *Method*)

Require: Java
Ensure: Java ∪ fresh variables, `with` statement, locked names

1: **for all** *methosAccess* in `polyUses`(m) **do**
2: INLINE METHOD ACCESS(*methodAccess*)
3: **end for**
4: REMOVE METHOD(m) **or** ID()

Algorithm 22 INLINE METHOD ACCESS(ma: *MethodAccess*)

Require: Java
Ensure: Java ∪ fresh variables, `with` statement, locked names

1: $am \leftarrow$ INLINE TO ANONYMOUS METHOD(ma)
2: INTRODUCE OUT PARAMETER(am)
3: OPEN VARIABLES(am)
4: *node* $\leftarrow$ INLINE ANONYMOUS METHOD(am)
5: **if** *node* is a *Block* **then** {in particular, it does not have a label}
6: INLINE BLOCK(*node*)
7: **end if**

Algorithm 23 INLINE TO ANONYMOUS METHOD(am: *MethodAccess*) : *AnonymousMethod*

Require:
Ensure: adds `with` statement, locked names

1: **assert** `target`(ma) is unambiguous
2: *target* $\leftarrow$ `target`(ma)
3: **assert** *target* has a body
4: $am \leftarrow$ copy target as anonymous method, with locked names, unfolded synchronize and arguments from ma
5: **if** ma is right child of *Dot* **then**
6: add `with` statement around the body of am
mapping `this` to qualifier of ma
7: replace qualifier and the access with am
8: **else**
9: replace ma with am
10: **end if**
11: **return** am

Algorithm 24 INTRODUCE OUT PARAMETER(*am*: *AnonymousMethod*)

Require:
Ensure: adds fresh variables, locked names

1: `eliminateVarargs`()
2: *parent* ← `parent`(*am*)
3: **if** *parent* is simple assignment expression **then**
4: **assert** destination of *parent* assignment is a variable
5: *v* ← destination variable of *parent*
6: set return type of *am* to `void`
7: add new fresh paremeter to *am* with `out` modifier, type locked to `type`(*v*)
8: add new argument to *am* locked to `decl`(*v*)
9: change `return` statements to assignment to the parameter and simple `return`
10: replace *parent* with *am*
11: **end if**

Algorithm 25 OPEN VARIABLES(*am*: *AnonymousMethod*)

Require:
Ensure: adds fresh variables, locked names

1: **for all** (*par*, *arg*) in `reverse`(`zip` `params`(*am*) `args`(*am*)) **do**
2: **if** *par* is `in` parameter **then**
3: *newdecl* ← new variable declaration initialized to *arg* with locked names
4: insert *newdecl* at the beginning of block of *am*
5: **else** {*par* is `out` parameter}
6: **assert** *arg* is a variable access
7: lock all uses of *par* to `decl`(*arg*)
8: **end if**
9: remove *par*, *arg* from *am*
10: **end for**

Algorithm 26 INLINE ANONYMOUS METHOD(*am*: *AnonymousMethod*) : *ASTNode*

Require: no implicit `return`, no `return void`
Ensure: no explicit `return`

assert *am* has no parameters
assert one of the following three conditions is `true`
if *am* is the expression in an expression statement **then**
  $l \leftarrow$ fresh label usable for block of *am*
  for all *ret* in `returns`(*am*) **do**
    if *ret* has a result **then**
      if result of *ret* is pure **then**
        {as *am* is in an expression statement the result can be discarted and not evaluated}
      else
        add an evaluation of the result of *ret* before *ret*
      end if
    end if
    replace *ret* with a `break` statement with label *l*
  end for
  replace *am* with its block and remove useless `break`s
else if *am* is an expression closure, i.e. body is only a return statement **then**
  replace *am* with expression from the `return` statement
else if *am* is the expression in a `return` statement **then**
  replace the outer `return` with block of *am*
end if
return the expression or statement we replaced with

Algorithm 27 INLINE BLOCK(*b*: *Block*)

Require:
Ensure: add locked names

assert *b* is a statement in a block without a label
`lockAllNames`(parent block of *b*)
for all *stmt* in *b* **do**
  remove *stmt* from *b* and put it just before *b*
end for
remove *b*

Algorithm 28 INLINE TEMP(d : *LocalVar*)

Require: Java
Ensure: Java

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

Algorithm 29 SPLIT DECLARATION(d : *LocalVar*) : `option` *Assignment*

Require: Java $\setminus$ 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**

Algorithm 30 INLINE ASSIGNMENT(a : *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 31 REMOVE DECL(d : *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 32 INSERT METHOD(m : *Method*, T : *Type*)

Require: Java
Ensure: Java ∪ locked method names

1: `lockMethodNames`(`name`(m))
2: **assert** `canIntroduceMethod`(m, T)
3: **assert not** `isDynamicallyCallable`(m)
4: **assert** $\{$`name`(td)$|$*TypeDecl* $td \in$ `below`(m)$\}$
$\cap \{$`name`(t)$|t$ is enclosing type of $T \vee t = T\} = \emptyset$
5: insert method m into the type T
6: **if** m is `abstract` **then**
7: **for all** *type* in `typesToMakeAbstract`(m) **do**
8: MAKE TYPE ABSTRACT(*type*)
9: **end for**
10: **end if**

Algorithm 33 `canIntroduceMethod`(m : *Method*, T : *Type*)

1: **assert** m is not `static` **or** T is not inner
2: **assert** there is no local method in T with same signature errasure as m
3: **assert** if there are any like-named methods in superclasses, we must be able to override or hide them, and similarly for subclasses

Algorithm 34 `typesToMakeAbstract`(m : *Method*) : `set` Type

1: do DFS from `hostType`(m) through child types
but do not visit a type that declares a method that *overrides* m
(in particular, visit a type in a different package, even if it can't override m)
2: **return** set of all visited types

Algorithm 35 INTRODUCE FACTORY(cd : *ConstructorDecl*)

Require: Java
Ensure: Java ∪ 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**

2.14 Introduce Parameter

This refactoring turns an expression into a parameter of the surrounding method. Implemented in `ChangeMethodSignature/IntroduceParameter.jrag`; see Algorithm 37.

2.15 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/IntroduceParameterObjec` see Algorithm 38.

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!!!)

2.16 Make Method Static

This refactoring makes a method static. Implemented in `MakeMethodStatic/MakeMethodStatic.jrag`; see Algorithms 39, **??**, **??**.

2.17 Move Inner To Toplevel

This refactoring converts a member type to a toplevel type. Implemented in `TypePromotion/MoveMemberTypeToToplevel.jrag`; see Algorithms 40, 41, 42.

2.18 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`; see Algorithm 43.

2.19 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.

We include specification only for Move Static Method; see Algorithm 44. They are all implemented in `Move/MoveMembers.jrag`.

Algorithm 36 INTRODUCE INDIRECTION(m : *Method*, B : *ClassOrInterface*)

Require: Java
Ensure: Java ∪ locked names, `return void`

assert B is non-library
$fn \leftarrow$ fresh method name
$m' \leftarrow$ copy of m with locked names and empty body
set name of m' to fn
$xs \leftarrow$ locked accesses to parameters of m'
set body of m' to `return` m`(`xs`);`
INSERT METHOD(`hostType`(m), m')
MAKE METHOD STATIC(m')
MOVE STATIC METHOD(m', B)

Algorithm 37 INTRODUCE PARAMETER(e : *Expr*, n : *Name*)

Require: Java
Ensure: Java ∪ locked names

assert n is a valid name
assert e is extractible and constant
assert e appears within a method m
assert m is not overridden by and does not override any other methods
assert m has no parameter or local variable n
`lockMethodCalls`(`name`(m))
$t \leftarrow$ effective type of e
$p \leftarrow$ new parameter of type t and name n
insert p as the first parameter of m
replace e with locked access to p
for all calls c to m **do**
    insert a locked copy of e as first argument of c
end for

Algorithm 38 INTRODUCE PARAMETER OBJECT(m : *Method*, P : `set` *Parameter*, w : `set` Name, n : `set` Name)

Require: Java \ variable arity parameters
Ensure: Java ∪ locked names

1: **assert** m has a body
2: **assert** the parameters in P are in contiguous positions $i, \ldots, i+k$
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: $[p_1; \ldots; p_n] \leftarrow$ parameters of r
9: $p \leftarrow$ new parameter of type W and name n
10: replace parameters $p_i, \ldots, p_{i+k}$ with p
11: **for all** $j \in \{i, \ldots, i+k\}$ **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: $[a_1; \ldots; a_n] \leftarrow$ arguments of c
17: replace arguments $a_i, \ldots, a_{i+k}$ with `new` W`(`a_i`,...,`a_{i+k}`)`
18: **end for**
19: **end for**

Algorithm 39 MAKE METHOD STATIC(m : *Method*)

Require: Java
Ensure: Java ∪ return `void`, fresh variables, `with` statement, locked names, demand `final` modifier

1: **assert** m has a body
2: *newMethod* ← `copy`(m)
3: *delegator* ← m
4: `lockMethodNames`(`name`(*delegator*))
5: add `static` modifier to *newMethod*
6: add new parameter to *newMethod* with fresh name, type locked to `hostType`(m), and demand final
7: put a `with` statement around the body of *newMethod* mapping `this` to the new parameter
8: CLOSE OVER VARIABLES(*newMethod*)
9: change the block of *delegator* method to a call to *newMethod* with `this` and parameters of *delegator* as arguments
10: INSERT METHOD(`hostType`(*delegator*), *newMethod*)

2.20 Promote Temp to Field

This refactoring turns a local variable into a field. Implemented in `PromoteTempToField/PromoteTempToField.`
see Algorithms 45, 46.

2.21 Pull Up

This refactoring pulls up a method m from its host class B to the super class A. Implemented in `PullUp/PullUpMethod.jrag`; see Algorithm 47.

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.

2.22 Push Down

This refactoring pushes a method down to all subclasses of its defining class. Implemented in `PushDown/PushDownMethod.jrag`; see Algorithms 48, 49, 50, 51, 52.

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

2.23 Rename

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

Refactoring Rename Field (Algorithm 53) 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 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.

Refactoring Rename Local (Algorithm 54) 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 (Algorithm 55) 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 (Algorithm 56) 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.

Refactoring Rename Package (Algorithm 57) changes the name of a package P to n and renames also all subpackages.

2.24 Self-Encapsulate Field

This refactoring makes a field private, rerouting all accesses to it through getter and setter methods. Implemented in `SelfEncapsulateField/SelfEncapsulateField.jrag`; see Algorithm 58.

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.

Algorithm 40 MOVE MEMBER TYPE TO TOPLEVEL(M : *MemberType*)

Require: Java
Ensure: Java ∪ locked names

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

Algorithm 41 INSERT TYPE(p : *Package*, T : *ClassOrInterface*)

Require: Java
Ensure: Java ∪ 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

Algorithm 42 MAKE TYPE STATIC(M : *MemberType*)

Require: Java
Ensure: Java ∪ `with`, locked names

1: $[A_n; \ldots; A_1] \leftarrow$ enclosing types of M
2: **for all** $i \in \{1, \ldots, n\}$ **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
`with(this$n, ..., this$1, this) {...}`
31: **end if**
32: **end for**

Algorithm 43 MOVE METHOD(m : *InstanceMethod*, v : *Variable*)

Require: Java
Ensure: Java ∪ locked names, `return void`, fresh variables, demand `final`

assert v is either a parameter of m or a field
$T \leftarrow$ type of v
assert T is a non-library class
assert m has a body and is not from library
$m' \leftarrow$ copy of m with `synchronized` removed and all names locked
$xs \leftarrow$ list of locked accesses to parameters of m
if v is a parameter **then**
    $i \leftarrow$ position of v in parameter list of m
    remove ith parameter from m'
    remove ith element of xs
else
    $i \leftarrow 0$
end if
$v' \leftarrow$ `final` local variable declaration with same name and type as v, initialised to `this`
insert v' as first statement into m'
lock all uses of v inside m' to v'
$qs \leftarrow []$
for all enclosing classes C of m **do**
    $p_C \leftarrow$ demand `final` parameter with fresh name, of type C
    make p_C the ith parameter of m'
    $e \leftarrow$ access to C`.this`
    insert e as ith element into xs
    $qs \leftarrow [\![p_C]\!] :: qs$
end for
wrap body of m' into `with(`qs`) {...}`
set body of m to `return` $[\![v]\!]$`.`$[\![m]\!]$`(`xs`);`
INSERT METHOD(T, m')
eliminate `with` statement in m'
INLINE TEMP(v')
for all p_C **do**
    REMOVE PARAMETER(p_C) **or** ID()
end for

Algorithm 44 MOVE STATIC METHOD(m : *StaticMethod*, *target* : *Type*)

Require: Java
Ensure: Java ∪ locked names

`lockAllNames`() in m
`lockNames`(`name`(m))
remove m from `hostType`(m) and insert it into the *target*

Algorithm 45 PROMOTE TEMP TO FIELD(d : *LocalVar*)

Require: Java
Ensure: Java ∪ locked dependencies

1: ⌊SPLIT DECLARATION⌋(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: ⌊INSERT FIELD⌋(`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 46 INSERT FIELD(T : *ClassOrInterface*, d : *Field*)

Require: Java
Ensure: Java ∪ 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

Algorithm 47 PULL UP METHOD(m : *Method*)

Require: Java
Ensure: Java ∪ 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

Algorithm 48 TRIVIALLY OVERRIDE(B : *Type*, m : *VirtualMethod*) : `option` *MethodCall*

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

assert m is not `final`
if m not a member method of B **then**
  return `None`
end if
$m' \leftarrow$ copy of m with locked names
if m is `abstract` **then**
  INSERT METHOD(B, m')
  return `None`
else
  $xs \leftarrow$ list of locked accesses to parameters of m'
  $c \leftarrow$ `super.`m`(`xs`)`
  set body of m' to `return` c`;`
  INSERT METHOD(B, m')
  return `Some` c
end if

Algorithm 49 REMOVE METHOD(m : *Method*)

Require: Java
Ensure: Java

assert $(\texttt{uses}(m) \cup \texttt{calls}(m)) \setminus \texttt{below}(m) = \emptyset$
$o \leftarrow \{m' \mid m <: m'\}$
if $o \neq \emptyset \wedge \forall m' \in o.m'$ is abstract **then**
  for all B in `typesToMakeAbstract`(m) **do**
    MAKE TYPE ABSTRACT(B)
  end for
end if
remove m

Algorithm 50 MAKE METHOD ABSTRACT(m : *Method*)

Require: Java
Ensure: Java

assert m is not `native`, `static`, `private`, nor `final`
assert there are no static calls to m (e.g., `super`-call)
for all B in `typesToMakeAbstract`(m) **do**
  MAKE TYPE ABSTRACT(B)
end for
make m `abstract`

Algorithm 51 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 52 PUSH DOWN VIRTUAL METHOD(m : *VirtualMethod*)

Require: Java
Ensure: Java ∪ locked names

1: **for all** types $B <: \texttt{hostType}(m)$ **do**
2: $c \leftarrow \lfloor$TRIVIALLY OVERRIDE$\rfloor(B, m)$
3: **if** $c \neq \texttt{None}$ **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 53 RENAME FIELD(f : *Field*, n : *Name*)

Require: Java
Ensure: Java ∪ locked names

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

Algorithm 54 RENAME LOCAL(v : *Local*, n : *Name*)

Require: Java
Ensure: Java ∪ 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: $\texttt{lockNames}(\texttt{block}(v), \{n, \texttt{name}(f)\})$
4: set name of v to n

Algorithm 55 RENAME METHOD($m : \mathit{Method}, n : \mathit{Name}$)

Require: Java
Ensure: Java ∪ locked names, locked overriding

1: **assert** n is a valid name
2: `lockMethodNames`($\{\texttt{name}(m), n\}$)
3: `lockOverriding`($\{\texttt{name}(m), n\}$)
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 56 RENAME TYPE($T : \mathit{Type}, n : \mathit{Name}$)

Require: Java
Ensure: Java ∪ 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`($\{\texttt{name}(T), n\}$)
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

Algorithm 57 RENAME PACKAGE(P : *Package*, n : *Name*)

Require: Java
Ensure: Java ∪ locked names

assert n is a valid name
assert there is no package with name n
`lockAllPackageAccesses`()
for all *package* in {packages with name starting with `name`(P)} **do**
    RENAME PACKAGE(*package*, `name`(*package*).`replaceFirst`(`name`(P), n))
end for
set name of P to n

Algorithm 58 SELF-ENCAPSULATE FIELD(f : *Field*)

Require: Java \ abbreviated assignments
Ensure: Java ∪ locked names

create getter method g for f
if f is not `final`, create setter method s for it
for all all uses u of f and its substituted copies **do**
    if $u \notin$ `below`(g) ∪ `below`(s) **then**
        if u is an rvalue **then**
            replace u with locked access to g
        else
            if f is not `final` **then**
                $q \leftarrow$ qualifier of u, if any
                $r \leftarrow$ RHS of assignment for which u is LHS
                replace u with locked access to s on argument r, qualified with q if applicable
            end if
        end if
    end if
end for

3 Node Types

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

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

Figure 1: Node Types

4 Utility Functions

See Fig. 2.

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

List of Algorithms