Specifications of Implemented Refactorings

Max Schäfer

March 26, 2010

This document collects the pseudo-code specifications of all refactoring implemented in our engine.

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`.

2.2 Convert Anonymous to Nested

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

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 ∪ locked names

1: $c \leftarrow$ class instance expression containing A
2: $d \leftarrow$ [EXTRACT TEMP](c, `unCapitalise`(n))
3: $b \leftarrow$ enclosing body declaration of s
4: `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*) : *MemberType*

Require: Java
Ensure: Java

1: $L \leftarrow$ CONVERT ANONYMOUS TO LOCAL(A)
2: **return** 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`.

TODO: provide specification of close over type variables and close over local variables (implemented in TypePromotion/CloseOverVariables.jrag)

2.4 Extract Class

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

This is only a bare-bones specification. The implementation additionally allows t 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`.

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 is not on the right-hand side of a dot.

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: close L over type variables
3: close L over local variables
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 EXTRACT CLASS(C : *Class*, *fs* : `list` *Field*, n : *Name*, *fn* : *Name*)

Require: Java
Ensure: Java ∪ locked dependencies, 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: remove f
12: INSERT FIELD(W, f)
13: **if** f has initialiser **then**
14: lock flow dependencies of f
15: $e \leftarrow$ initialiser of f
16: remove initialiser of f
17: add e as argument to initialisation of w
18: $p \leftarrow$ new parameter of same name and type as f
19: **for all** constructors *cd* of W **do**
20: add copy of p as parameter of W
21: add assignment from parameter to f to body of *cd*
22: **end for**
23: **end if**
24: **end for**

Algorithm 5 EXTRACT CONSTANT($e : Expr, n : Name$)

Require: Java
Ensure: Java ∪ locked dependencies

1: **assert** e is extractible
2: $A \leftarrow$ enclosing type of e
3: $t \leftarrow$ effective type of e
4: $f \leftarrow$ new `public 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

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

2.7 Extract Temp

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

Algorithm 6 EXTRACT TEMP($e : Expr, n : Name$)

Require: Java
Ensure: Java

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)

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 7 INSERT LOCAL VARIABLE($s : \mathit{Stmt}, v : \mathit{LocalVar}$

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 8 EXTRACT ASSIGNMENT($v : \mathit{LocalVar}, e : \mathit{Expr}$) : *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`.

Algorithm 9 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**

2.8 Inline Constant

This refactoring inlines a constant field into all its uses. Implemented in `InlineTemp/InlineConstant.jrag`.

Algorithm 10 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 11 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 12 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**

2.9 Inline Method

2.10 Inline Temp

This refactoring inlines a local variable into all its uses. Implemented in `InlineTemp/InlineTemp.jrag`.

Algorithm 13 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 14 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**

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`.

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*.

Algorithm 15 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 16 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 17 INTRODUCE FACTORY(cd : *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**

2.12 Introduce Indirection

2.13 Introduce Parameter

2.14 Introduce Parameter Object

2.15 Move Inner To Toplevel

This refactoring converts a member type to a toplevel type. Implemented in `TypePromotion/MoveMemberTypeToToplevel.jrag`.

Algorithm 18 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 19 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

2.16 Move Instance Method

2.17 Move Members

2.18 Promote Temp to Field

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

2.19 Pull Up

2.20 Push Down

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

Algorithm 20 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: put modifier `static` on M
26: **for all** callables m of M **do**
27: **if** m has a body **then**
28: surround body of m by
`with(this$n, ..., this$1, this) {...}`
29: **end if**
30: **end for**

Algorithm 21 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 22 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 23 TRIVIALLY OVERRIDE(B : *Type*, m : *VirtualMethod*) : `option` *MethodCall*

Require: Java \ 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: $c \leftarrow$ `super.`m`(`xs`)`
12: set body of m' to `return` c`;`
13: insert method m' into B
14: **return** `Some` c
15: **end if**

Algorithm 24 REMOVE METHOD(m : *Method*)

Require: Java
Ensure: Java

1: **assert** $(\texttt{uses}(m) \cup \texttt{calls}(m)) \setminus \texttt{below}(m) = \emptyset$
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 25 MAKE METHOD ABSTRACT(m : *Method*)

Require: Java
Ensure: Java

1: **assert** $\texttt{calls}(m) \setminus \texttt{below}(m) = \emptyset$
2: **for all** types B that inherit m **do**
3: MAKE TYPE ABSTRACT(B)
4: **end for**
5: make m `abstract`

Algorithm 26 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 27 PUSH DOWN VIRTUAL METHOD(m : *VirtualMethod*)

Require: Java
Ensure: Java $\cup$ 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()

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

2.21 Rename

2.22 Self-Encapsulate Field

3 Node Types

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

4 Utility Functions

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
`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