Specifications of Implemented Refactorings

Max Schäfer

March 22, 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.

The names of refactorings are written in SMALL CAPS, whereas utility functions appear in `monospace`. A list of utility functions with brief descriptions appears in Fig. 2.

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. We also use the higher-order function `map`, with lambda expressions to denote the function being mapped over the list. The notation $[\![d]\!]$ denotes a locked name that binds to declaration d.

Creation of a node is denoted by $\mathit{NodeType}(a_1, \ldots, a_n)$, where $\mathit{NodeType}$ is the type of the node being created and a_i are child nodes or other arguments.

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 Local to Member Class

This refactoring converts a local class to a member class. Implemented in `TypePromotion/LocalClassToMemberClass.jrag`.

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

Require: Java
Ensure: Java ∪ locked names

```
$c \leftarrow$ getClassInstanceExpr($A$)
$s \leftarrow$ [EXTRACT TEMP]($c$, unCapitalise($n$))
$b \leftarrow$ enclosingBodyDecl($s$)
lockTypeNames($b$, $n$)
$t \leftarrow$ asNamedClass($A$, $n$)
removeTypeDecl($c$)
setTypeAccess($c$, $[\![t]\!]$)
return insertLocalClass($s$, $t$)
```

Algorithm 2 CONVERT LOCAL TO MEMBER CLASS(L : *LocalClass*) : *MemberType*

Require: Java
Ensure: Java ∪ locked names

```
$h \leftarrow$ enclosingType($L$)
closeOverTypeVariables($L$)
closeOverLocalVariables($L$)
if inStaticContext($L$) then
  addModifier($L$, static)
end if
lockTypeNames(programRoot(), name($L$))
lockNames($L$)
removeStmt($L$)
return insertMemberType($h$, $L$)
```

Algorithm 3 CONVERT ANONYMOUS TO NESTED(A : *AnonymousClass*) : *MemberType*

Require: Java
Ensure: Java

```
$L \leftarrow$ CONVERT ANONYMOUS TO LOCAL($A$)
return CONVERT LOCAL TO MEMBER CLASS($L$)
```

2.3 Convert Anonymous to Nested

This refactoring converts an anonymous class to a member class.

Note that this specification does not handle the special case where A occurs in a field initialiser.

3 Node Types

Node Type	**Description**
AnonymousClass	anonymous class declaration; is an *Expr*
Assignment	expression statement consisting of a simple assignment; is a *Stmt*
Block	block of statements; is a *Stmt*
Class	class declaration; is a *Type*
Callable	either a method or a constructor
Expr	expression
Field	field declaration
LocalClass	local class declaration, contains a *Class*; is a *Stmt*
LocalVarDecl	local variable declaration, part of a *LocalVarDeclStmt*
LocalVarDeclStmt	local variable declaration statement; is a *Stmt*
MemberType	type declared inside another type; is a *Type*
Method	method declaration; is a *Callable*
MethodCall	method call
Parameter	parameter declaration
Return	return statement; is a *Stmt*
Stmt	statement
SuperCall	super call of a method; is a *MethodCall*
Type	type declaration
VirtualMethod	non-`private` instance method; is a *Method*
With	`with` construct (language extension)

Table 1: Node Types

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

4 Utility Functions

Name	**Description**
`addParameter`(c, p)	gives callable c a new parameter p
`addToplevelType`(p, T)	introduces a new toplevel type T into package p; fails if a type of the same name exists
`asNamedClass`(A, n)	constructs a class declaration with name n that has the same body as anonymous class A

`callables`(T)	returns the set of all callables defined in type T
`chainingInvocations`(c)	returns all invocations of constructor c from within constructors of the same class
`constructors`(T)	returns the set of constructors of type T
`copy`(t)	returns a copy of the subtree t
`copyWithLockedNames`(t)	returns a copy of the subtree t, where all names have been locked to their declarations
`definesMethod`(T, s)	checks whether type T defines a method with signature s
`discardQualifier`(e)	delete any qualifier that expression e might have
`enclosingBodyDecl`(s)	returns the innermost syntactically enclosing body declaration around statement s
`enclosingInstances`(e)	returns the list of enclosing instances of an expression
`enclosingTypes`(T)	returns the list of enclosing types of type T
`getBody`(c)	returns the body of callable c, fails if c is a method without body
`getClassInstanceExpr`(A)	returns the class instance expression for anonymous class A
`getInit`(d)	returns the initialiser of variable declaration d; fails if there is none
`getParms`(c)	returns the list of parameters of callable c
`hasBody`(c)	checks whether callable c has a body
`hasInit`(d)	checks whether variable declaration d has an initialiser
`hostType`(b)	returns the host type of body declaration b
`insertField`(T, f)	inserts field f into type T
`insertLocalClass`(s, c)	wraps c into a *LocalClass* and inserts it right before statement s; fails if s is not directly enclosed by a block, or if an enclosing type of s has the same name as c or a type declared syntactically within c
`insertMethod`(T, m)	inserts method m into type T
`insertStmt`(c, s)	inserts statement s as the first statement into the body of callable c
`insertStmtAfter`(s, s')	inserts statement s' after statement s; fails if s is not directly enclosed by a block
`instantiations`(C)	returns the set of all class instance expressions constructing instances of class C
`invocations`(c)	returns the set of all invocations of constructor c, including instantiations
`isAbstract`(q)	checks whether type or method q is abstract
`isChaining`(c)	checks whether constructor c recursively invokes a constructor of the same class
`isClass`(T)	checks whether type T is a class
`isInterface`(T)	checks whether type T is an interface
`lhs`(a)	returns the left hand side of assignment a
`lock`(t)	locks all naming, flow, and synchronization dependencies in subtree t

`lockNames`(t)	locks all naming dependencies in subtree t
`lockTypeNames`(t, n)	locks type names referring to a type named n in subtree t
`makeAbstract`(q)	puts an `abstract` qualifier on type or method q (if it does not have one already)
`memberMethods`(T)	returns the set of member methods of T, including methods inherited from an ancestor type that are not overridden in T
`monoCalls`(m)	returns the set of all calls that statically resolve to method m
`name`(e)	returns the name of program entity e
`nonChainingInvocations`(c)	`invocations`$(c) \setminus$ `chainingInvocations`(c)
`polyCalls`(m)	returns an over-approximation of the set of all calls that may dynamically resolve to method m
`programRoot`$()$	returns the root node of the AST
`removeInit`(d)	removes the initialiser of variable declaration d, if any
`removeBodyDecl`(b)	removes the body declaration b from its host type
`removeStmt`(s)	removes the statement s from its enclosing block; fails if s is not in a block
`removeTypeDecl`(c)	removes the type declaration attached to class instance expression c, if any
`replaceExpr`(e, e')	replaces expression e with e', wrapping e' into parentheses if needed
`rhs`(a)	returns the right hand side of assignment a
`setBody`(c, b)	makes block b the body of callable c
`setTypeAccess`(c, a)	replaces the type access for class instance expression c with a
`signature`(m)	returns the signature of method m
`subtree`(t)	returns the set of all nodes in the subtree t
`thisCall`(c)	returns the recursive constructor invocation of constructor c; fails if it does not have one
`unCapitalise`(n)	returns the name n with its first letter converted to lowercase

Table 2: Utility Functions