# Specifications of Implemented Refactorings

Max Schäfer

March 23, 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 *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`.

We first retrieve the class instance expression $c$ of which $A$ is a part. Then we apply the `Extract Temp` refactoring to move $c$ into its own statement. All references to types named $n$ are locked within the enclosing body declaration $b$. Then $A$ is converted into a class $t$ with name $n$. We remove $A$ from $c$, make sure that $c$ constructs an object of type $t$, and insert $t$ as a local class right before the statement containing $c$.

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

**Require:** Java
**Ensure:** Java ∪ locked names

1: $c \leftarrow$ `getClassInstanceExpr`($A$)
2: $s \leftarrow$ [EXTRACT TEMP]($c$, `unCapitalise`($n$))
3: $b \leftarrow$ `enclosingBodyDecl`($s$)
4: `lockTypeNames`($b, n$)
5: $t \leftarrow$ `asNamedClass`($A, n$)
6: `removeTypeDecl`($c$)
7: `setTypeAccess`($c$, $[\![t]\!]$)
8: **return** `insertLocalClass`($s, t$)

## 2.2 Convert Anonymous to Nested

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

**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($L$)

We can implement this refactoring simply as the composition of CONVERT ANONYMOUS TO LOCAL and CONVERT LOCAL TO MEMBER CLASS. Note that this specification does not handle the special 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`.

We start by computing the enclosing type $h$ of $L$, into which we want to eventually insert $L$ as a member type. Then we close $L$ over type variables and local variables from the enclosing body declaration.

The utility function `closeOverTypeVariables`($L$) collects all type variables $V$ of the enclosing body declaration of $L$ which are used inside $L$. Every such $L$ is added as a type parameter to $L$, and every use of $L$ is augmented by a corresponding type access.

Similarly, `closeOverLocalVariables`($L$) adds a field $f_v$ to $L$ for every local variable $v$ of the enclosing body declaration of $L$. All uses of $v$ from within $L$

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

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

```
1: h ← enclosingType(L)
2: closeOverTypeVariables(L)
3: closeOverLocalVariables(L)
4: if inStaticContext(L) then
5:   addModifier(L, static)
6: end if
7: lockTypeNames(programRoot(), name(L))
8: lockNames(L)
9: removeStmt(L)
10: return insertMemberType(h, L)
```

are replaced by locked accesses to $f_v$, and the constructors and instantiations of $L$ are adjusted to initialise $f_v$ to the value of $v$.

If $L$ is in a static context (for example because its enclosing body declaration is static), a `static` qualifier is added. All references to types with the same name as $L$ throughout the program are locked; likewise, all names within $L$ are locked. Then $L$ is removed from its enclosing body declaration and inserted as a member type into $h$.

## 2.4 Extract Class

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

It first determines the maximum visibility $v$ of any of the fields *fs* to be extracted. Then it creates a static wrapper class $W$ of visibility $v$ and name $n$, and inserts it as a member type into $C$. Likewise, it creates a wrapper field $w$ of visibility $v$ and type $W$, which is initialised to a new instance of $W$, and inserts it as a member field into $C$.

Now it examines every field $f \in fs$ in turn. None of the fields can be static. Every access $v$ that binds to $f$ is qualified by $[\![w]\!]$ (note that this will work even if $v$ is already qualified). Then $f$ is moved from $C$ to $W$. If $f$ has an initialiser $e$, that initialising expression is added as an argument to the class instance expression initialising $w$, after locking its flow dependencies. Additionally, every constructor $c$ of $W$ (there will only be one) is given a new parameter $p$ and an assignment that assigns the value of $p$ to the field $f$.

It might be nicer to implement this refactoring as a step-by-step transformation using a Scala-like `object` construct: We first construct a wrapper object $w$ without any members. Then, one by one, the fields in *fs* are moved into $w$, adding their initialisers as constructor arguments if necessary.

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

**Require:** Java
**Ensure:** Java ∪ locked dependencies, first-class array init

$v \leftarrow$ `max(getVisibility(`*fs*`))`
$W \leftarrow$ *ClassDecl*([`static`; $v$], $n$)
`insertMemberType(`$C$, $W$`)`
$w \leftarrow$ *Field*($v$, $[\![W]\!]$, *fn*, *ClassInstanceExpr*($[\![W]\!]$))
`insertField(`$C, w$`)`
**for all** $f \in$ *fs* **do**
  **assert** ¬`isStatic(`$f$`)`
  **for all** $v \in \{v \mid v \rightarrow_b f\}$ **do**
    `replaceExpr(`$v$, `Dot(`$[\![w]\!]$, $v$`))`
  **end for**
  `removeField(`$f$`)`
  `insertField(`$W, f$`)`
  **if** `hasInit(`$f$`)` **then**
    `lockFlow(`$f$`)`
    $e \leftarrow$ `removeInit(`$f$`)`
    `addArgument(getInit(`$w$`)`, $e$`)`
    $p \leftarrow$ `Parameter(`$[\![$`type(`$f$`)`$]\!]$, `name(`$f$`))`
    **for all** $cd \in$ `constructors(`$W$`)` **do**
      $p' \leftarrow$ `copy(`$p$`)`
      `addParameter(`$cd, p'$`)`
      `addStmt(`$cd$, *Assignment*($[\![f]\!]$, $[\![p']\!]$))
    **end for**
  **end if**
**end for**

## 2.5 Extract Constant

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

**Algorithm 5** EXTRACT CONSTANT($e : \mathit{Expr}, n : \mathit{Name}$)

**Require:** Java
**Ensure:** Java ∪ locked dependencies

1: **assert** `extractible`($e$)
2: $h \leftarrow$ `enclosingType`($e$)
3: `lock`($e$)
4: $f \leftarrow \mathit{Field}([\texttt{static}; \texttt{final}; \texttt{public}], [\![\texttt{effectiveType}(e)]\!], e)$
5: `replaceExpr`($e$, $[\![f]\!]$)
6: `insertField`($h, f$)

We first ensure that $e$ is extractible: this means that its type cannot be `void`, and it cannot be a reference to a type or package, nor can it be the keyword `super`; furthermore, it cannot be on the right-hand side of a dot.

Then all dependencies within $e$ are locked, and we construct a `public static final` field $f$ that is initialised to $e$. The type of $f$ is the *effective type* of $e$, which 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.

Now $e$ is simply replaced by a locked access to $f$, and $f$ is inserted into the enclosing type.

## 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($e : \mathit{Expr}, n : \mathit{Name}$)

**Require:** Java
**Ensure:** Java

1: $v \leftarrow$ [INSERT LOCAL VARIABLE](`enclosingStmt`($e$), `effectiveType`($e$), $n$)
2: [EXTRACT ASSIGNMENT]($v, e$)
3: MERGE DECLARATION($v$)

We first perform the INSERT LOCAL VARIABLE refactoring to create a local variable $v$ with the same type as $e$ and with name $e$ in the enclosing body declaration. Then we use EXTRACT ASSIGNMENT to extract $e$ into an assignment $a$ to $v$. Finally, we merge $a$ with the declaration $v$, turning it into its initialiser if possible.

### 2.7.1 Insert Local Variable

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

**Algorithm 7** INSERT LOCAL VARIABLE($s : \mathit{Stmt}, t : \mathit{Type}, n : \mathit{Name}$) : *LocalVarDecl*

**Require:** Java
**Ensure:** Java ∪ locked names

1: $b \leftarrow$ `enclosingBlock`$(s)$
2: **assert** `canIntroduceLocal`$(b, n)$
3: `lockNames`$(b, n)$
4: $v \leftarrow \mathit{LocalVarDecl}([\![t]\!], n)$
5: `insertStmtBefore`$(s, v)$
6: **return** $v$

The refactoring ensures that a variable of name $n$ can be introduced into the enclosing block $b$. This is not possible, for instance, if there already is a local variable of the same name in an enclosing scope. Then all references to variables of name $n$ are locked within $b$, and the local variable declaration is constructed and inserted into $b$.

### 2.7.2 Extract Assignment

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

**Algorithm 8** EXTRACT ASSIGNMENT($v : \mathit{LocalVarDecl}, e : \mathit{Expr}$) : *Assignment*

**Require:** Java
**Ensure:** Java ∪ locked dependencies

1: **assert** `extractible`$(e)$
2: $a \leftarrow \mathit{Assignment}([\![v]\!], e)$
3: **if** `inExprStmt`$(e)$ **then**
4: `replaceExpr`$(e, a)$
5: **else**
6: $s \leftarrow$ `enclosingStmt`$(e)$
7: `lock`$(e)$
8: `insertStmtBefore`$(s, a)$
9: `replaceExpr`$(e, [\![v]\!])$
10: **end if**
11: **return** $a$

The refactoring ensures that $e$ is an extractible expression and constructs the assignment $a$. If $e$ is in an expression statement, we can directly replace it with

$a$. Otherwise, we insert it before the enclosing statement, locking dependencies in $e$ and replacing it by a variable access.

### 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($v$ : *LocalVarDecl*)

**Require:** Java \ multi-declarations
**Ensure:** Java

1: **if** `hasInit`($v$) **then**
2: **return**
3: **end if**
4: $s \leftarrow$ `followingStmt`($v$)
5: **if** `isAssignmentTo`($s, v$) **then**
6: `setInit`($v$, `rhs`($s$))
7: `removeStmt`($s$)
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
**Ensure:** Java

1: **for all** $u \in \{u \mid u \rightarrow_b f\}$ **do**
2: [INLINE CONSTANT]($u$)
3: **end for**
4: **if** $\nexists u.(urightarrow_b f) \wedge$ (`hasInit`($f$) $\rightarrow$ `isPure`(`getInit`($f$))) **then**
5: `removeField`($f$)
6: **end if**

The basic algorithm is like for INLINE TEMP: inline every use, then remove the field if it is no longer used. Of course, removing the field means discarding its initialiser, so we can only do this if the field either has no initialiser, or the initialiser is a pure expression without side effects.

To inline a use $u$ of a field, we make sure that the referenced field is static and final and has an initialiser. We obtain a locked copy $e'$ of the initialiser, and replace the field use (along with its qualifier) by $e'$. Again, this is only possible if either $u$ has no qualifier, or the qualifier is side-effect free.

**Algorithm 11** INLINE CONSTANT($u$ : *FieldAccess*)

**Require:** Java
**Ensure:** Java ∪ locked dependencies

1: $f \leftarrow$ `decl`()
2: **assert** `isFinal`($f$) ∧ `isStatic`($f$) ∧ `hasInit`($f$)
3: $e \leftarrow$ `lockedCopy`(`getInit`($f$))
4: **assert** `isQualified`($u$) → `isPure`(`getQualifier`($u$))
5: `replaceExpr`(`getUnqualified`($u$), $e$)

## 2.9 Inline Method

## 2.10 Inline Temp

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

**Algorithm 12** INLINE TEMP($d$ : *LocalVarDecl*)

**Require:** Java
**Ensure:** Java

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

We proceed in three steps: first, we split off the variable initialisation into an assignment, then inline all uses of the assignment, and finally remove the variable.

**Algorithm 13** SPLIT DECLARATION($d$ : *LocalVarDecl*) : *Assignment*

**Require:** Java \ multi-declarations
**Ensure:** Java ∪ locked names, first-class array init

1: **assert** `hasInit`($d$)
2: $a = \mathit{Assignment}([\![d]\!],$ `getInit`($d$))
3: `insertStmtAfter`($d, a$)
4: `removeInit`($d$)
5: **return** $a$

To split a variable declaration, it needs to have an initialiser, which we turn into its own statement.

To inline an assignment $a$, we check that it is indeed an assignment to a local variable $x$. For every use $u$ we check that $a$ is its only reaching definition, that $u$ is not an lvalue, and that $u$ and $a$ are in the same body declaration. (If $x$ is final, $u$ could be inside a local class, for instance.) Assuming that all these conditions are fulfilled, we can replace $u$ by a locked copy of the right-hand side of $a$. Finally, if $x$ was inlined at least once we can delete the assignment.

**Algorithm 14** INLINE ASSIGNMENT($a$ : *Assignment*)

**Require:** Java
**Ensure:** Java ∪ locked dependencies

1: $x = \texttt{lhs}(a)$
2: **assert** $\texttt{decl}(x)$ is local variable
3: $U = \{u \mid u \rightarrow_r x\}$
4: **for all** $u \in U$ **do**
5: **assert** $\neg\exists x'.u \rightarrow_r x' \wedge x \neq x'$
6: **assert** $u$ is not an lvalue
7: **assert** $u, a$ are in same body declaration
8: $\texttt{replaceExpr}(u, \texttt{lockedCopy}(\texttt{rhs}(a)))$
9: **end for**
10: **if** $U \neq \emptyset$ **then**
11: $\texttt{removeStmt}(a)$
12: **end if**

**Algorithm 15** REMOVE DECL($d$ : *LocalVarDecl*)

**Require:** Java \ multi-declarations
**Ensure:** Java

1: **if** $\neg\texttt{hasInit}(d) \wedge \neg\exists u.u \rightarrow_b d$ **then**
2: $\texttt{removeStmt}(d)$
3: **end if**

To remove a variable declaration $d$, we need to ensure that it does not have an initialiser, and that it is never used.

## 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`.

**Algorithm 16** INTRODUCE FACTORY($cd$ : *ConstructorDecl*)

**Require:** Java
**Ensure:** Java ∪ locked names

1: $f \leftarrow$ `createFactoryMethod`($cd$)
2: **for all** $c \in \{c : \mathit{ClassInstanceExpr} \mid \texttt{original}(\texttt{decl}(c)) = cd \wedge \neg\texttt{hasTypeDecl}(c)$ **do**
3: `replaceExpr`($c$, *MethodCall*($[\![f]\!]$, `getArgs`($c$)))
4: **end for**

We first use `createFactoryMethod` 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 $cd$.

Then every class instance expression that uses $cd$ or a parameterised instance of $cd$ and does not have its own anonymous type declaration is replaced by a call to the factory method.

## 2.12 Introduce Indirection

## 2.13 Introduce Parameter

## 2.14 Introduce Parameter Object

## 2.15 Move Inner To Toplevel

**Algorithm 17** MOVE INNER TO TOPLEVEL($M$ : *MemberType*)

**Require:** Java \ implicit constructor invocations
**Ensure:** Java ∪ `with`, locked names

```
1: lockTypeNames(programRoot(), name(M))
2: lockNames(M)
3: if ¬isStatic(M) then
4:   [A_n; ...; A_1] = enclosingTypes(M)
5:   [f_n; ...; f_1] = [Field([[A_n]]); ...; Field([[A_1]])]
6:   for all i ∈ {1, ..., n} do
7:     addField(M, f_i)
8:     for all c ∈ constructors(M) do
9:       p := Parameter([[A_i]])
10:      addParameter(c, p)
11:      if isChaining(c) then
12:        addArgument(thisCall(c), [[p]])
13:      else
14:        addStmt(c, Assignment([[f_i]], [[p]]))
15:      end if
16:    end for
17:  end for
18:  for all c ∈ constructors(M) do
19:    for all u ∈ nonChainingInvocations(c) do
20:      es = enclosingInstances(u)
21:      assert |es| = n
22:      addArguments(u, es)
23:      discardQualifier(u)
24:    end for
25:  end for
26:  for all m ∈ callables(M) do
27:    if hasBody(m) then
28:      b := getBody(m)
29:      b' := With([[[f_n]]; ...; [[f_1]]; this], b)
30:      setBody(m, b')
31:    end if
32:  end for
33: end if
34: removeBodyDecl(M)
35: addToplevelType(M)
```

| 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