A Collection of Refactoring Specifications for Fortran 95, BC, and PHP 5

Jeffrey L. Overbey
Matthew J. Fotzler
Ashley J. Kasza
Ralph E. Johnson

Technical Report
Department of Computer Science
University of Illinois at Urbana-Champaign

August 18, 2010 Revision

Contents

IMPORTANT

Although these specifications have been carefully constructed, reviewed, and used as the basis of implementation, some errors, oversights, and ambiguities are inevitable. The first author will post errata, clarifications, and links to updated versions of this document at http://jeff.over.bz/papers.

Introduction

This technical report contains detailed specifications of several automated refactorings for Fortran, BC, and PHP. The specifications are written somewhat like an ANSI or ISO programming language specification, mathematically informal but precise, in English prose but with sufficient detail to serve as a basis for implementation.

To the extent possible, the constructs in each language are described syntactically. For example, an External Subprogram in Fortran is defined to be a ⟨*function-subprogram*⟩ or a ⟨*subroutine-subprogram*⟩ nested under an ⟨*external-subprogram*⟩. Such ⟨*bracketed-names*⟩ correspond to nonterminal symbols in a normative grammar for each programming language: the grammar in the ISO standard for Fortran 95 [1], the grammar in the POSIX specification for BC [2], and the Yacc grammar in the source code for the official distribution of PHP 5 [3]. The BC and PHP grammars use recursive productions to form lists of elements; in these cases, we will often ignore the recursive structure and, instead, refer to the list as a whole (e.g., "remove *X* and an appropriate adjacent comma from the list"), since implementations are likely to represent them as a list structure rather than a tree in abstract syntax anyway.

All algorithms are described imperatively, as a sequence of steps that may be executed to test the precondition or perform the transformation. It is not essential that an implementation execute these steps in the order listed; in many cases, the steps can be reordered and still produce the same results. For example, many precondition checks require a number of conditions to be checked, but these conditions are mutually disjoint, and therefore the order in which they are checked is inconsequential.

Terminology

This document contains four types of descriptions. **Predicates** return either TRUE or FALSE and are used in the specification of preconditions. **Preconditions** either PASS or FAIL and are used in the specification of refactorings. **Refactorings** consist of a list of preconditions and a program transformation. All of the preconditions must PASS if the program transformation is to be applied. **Procedures** describe algorithms used in the definition of a predicate, precondition, refactoring, or another procedure. Generally they will return a value.

The following conventions are used throughout.

in the immediate context of. A syntactic construct occurs *in the immediate context of* another if the former is an (immediate) child of the latter in a parse tree. For example, the Fortran 95 grammar contains the production

⟨*program-stmt*⟩ ::= PROGRAM ⟨*program-name*⟩;

so if a program contained the statement `program hello`, then `hello` would be a ⟨*program-name*⟩ which occurred in the immediate context of a ⟨*program-stmt*⟩.

in the context of. A syntactic construct occurs *in the context of* another if the former is a descendent of the latter in a parse tree. It may be a child, grandchild, great-grandchild, etc.

contains. A syntactic construct *contains* another syntactic construct if the former is an ancestor of the latter in a parse tree. (Note that *A* contains *B* if, and only if, *B* occurs in the context of *A*—i.e., these terms are opposites.)

existing vs. new. When it is not clear from context, syntactic constructs will be qualified as either an *existing* (i.e., the construct exists in the program being analyzed/transformed) or *new* (i.e., the construct is constructed from scratch or supplied by the user). For example, the Rename refactoring takes two names as input: an existing name—this is the entity in the program that will be renamed—as well as a new name for that entity.

← When a refactoring must construct new syntax to be inserted into a program, the new construct is given in the concrete syntax of the language. Consider the following example.

given a ⟨subroutine-name⟩ N, append to the ⟨program⟩

⟨*subroutine-subprogram*⟩ ← `subroutine` *N* ↲
`end subroutine` ↲

(The symbol ↲ indicates an end-of-line.) This means, "Construct a new ⟨*subroutine-subprogram*⟩ corresponding to the given concrete syntax (with the new subroutine name substituted for *N*), and append it to the ⟨*program*⟩." (The meaning of "the ⟨*program*⟩" would presumably be clear from context.) The left arrow is intended to denote that the new construct may be parsed from the given concrete syntax (although implementations may choose to construct the equivalent abstract syntax programmatically).

♦ In refactoring specifications, steps in which source code may be modified have been labeled with a black diamond.

◊ In some refactorings specifications, the precondition checks and the transformation traverse the program in similar ways. In these cases, it was simpler to intermix precondition checking steps with transformation steps. Precondition steps have been labeled with a white diamond.

Organization

The remainder of this technical report is organized as follows. One part is devoted to each language: Fortran, BC, and PHP. Each part begins with a list of definitions specific to that language. Defined terms are subsequently capitalized in order to make their usage more apparent. Following the list of definitions is a list of *requirements*—expectations that are made about the semantic analysis capabilities of the refactoring tool. For the most part, these are roughly equivalent to the capabilities of a (partial) compiler front end coupled with a cross-reference database.

The list of requirements is followed by a set of common predicates, preconditions, and procedures. These have been "factored out" of the refactoring specifications in order to keep the latter more concise and to avoid redundancy. These have each been given a two-letter abbreviation, enclosed in square brackets. Predicates and preconditions are abbreviated using two capital letters, e.g., [SI] or [LC]. Procedures are appreviated with a capital and lowercase letter, e.g., [Pr]. These abbreviations are used subsequently to indicate explicitly that a particular predicate, precondition, or procedure is being referenced.

Each part concludes with specifications of refactorings. Again, capitalization and abbreviations are used to indicate references to defined terms, predicates, preconditions, and procedures.

References

[1] International Organization for Standardization and International Electrotechnical Commission. *ISO/IEC 1539:1997: Information technology—Programming languages—Fortran.* Geneva, 1997.

[2] Institute of Electrical and Electronics Engineers. *IEEE Std 1003.1-2008: IEEE Standard for Information Technology – Portable Operating System Interface (POSIX) Base Specifications, Issue 7.* 2008.

[3] *PHP: Hypertext Preprocessor.* `http://www.php.net/`

[4] Adams, J.C., Brainerd, W.S., Martin, J.T., Smith, B.T., and Wagener, J.L. *Fortran 95 Handbook: Complete ISO/ANSI Reference.* MIT Press, Cambridge, MA, 1997.

Differential Refactoring

The specifications in this technical report are used to support “Differential Refactoring Engines,” a paper submitted for publication. The *Notes* under certain preconditions and refactorings relate to that material. In that paper, the evaluation is based on counting the number of *steps* each refactoring comprises. Each numbered item in a predicate, procedure, precondition, or refactoring counts as a single step. The total number of precondition checking steps for a refactoring is the sum of the steps for all predicates, preconditions, and procedures required to implement that refactoring’s precondition checks. The following tables indicate the number of steps in each predicate, precondition, procedure, and refactoring. The *Trad.* line indicates the number of steps in the traditional specification, while the *Diff.* line indicates which precondition checking steps were eliminated by use of a differential refactoring engine.

Fortran

Precondition	Steps
LC	4
SH	4
IC	5 + LC + SH = 13
SK	5 + LC + SH = 13
IN	9 + LC + SH + IC + SK = 27
SI	4
PR	5 + LC + SH = 13
Ou	5 + LC + SH = 13
OU	2 + Ou = 7
PP	3 + PR = 8
Pr	3 + PR = 8
Us	9 + LC + SH + IC + SK = 27
RN	4 + PR = 7
Rn	3 + PR = 8

							Total
Add Empty	Precond:	IN					
(2 steps)	Trad:	27					27
	Diff:	(elim)					0
Safe Delete	Precond:	SI					
(5 steps)	Trad:	4					4
	Diff:	(elim)					0
Rename	Precond:	IN	Warn	Custom1			
(4 steps)	Trad:	27	1	1			29
	Diff:	(elim)	1	1			2
Intro Implicit	Precond:						
(6 steps)	Trad:	0					0
	Diff:	0					0
Permute Sub	Precond:	Custom1	Custom2	Custom3	Custom4		
(13 steps)	Trad:	1	1	1	1		4
	Diff:	1	1	1	1		4
Add Use	Precond:	Custom1	Custom2	IN			
(5 steps)	Trad:	1	1	27			29
	Diff:	(elim)	(elim)	(elim)			0
Move	Precond:	RN	PP	Custom1			
(58 steps)	Trad:	7	8	8	(4 + LC)		23
(38+Ou+Pr+Us+Rn)	Diff:	(elim)	(elim)	3			3

BC

						Total
Precondition	**Steps**					
Ds	17					
IN	3 + Ds = 20					
Cv	7					
RS	3					
						Total
Add Var	Precond:	IN				
(4 steps)	Trad:	20				20
	Diff:	(elim)				0
Repl Block	Precond:					
(1 step)	Trad:	0				0
	Diff:	0				0
Insert Asgt	Precond:	Custom1				
(1 step)	Trad:	1				1
	Diff:					0
Move Expr	Precond:	Custom1	Custom2			
(2 steps)	Trad:	1	1			2
	Diff:	1	1			2
Ext Local	Precond:	RS				
(4 steps)	Trad:	3				3
	Diff:	(elim)				0
Add Func	Precond:	Custom1				
(1 step)	Trad:	1				1
	Diff:	(elim)				0
Pop Func	Precond:	Custom1	Custom2			
(10 + Cv = 17)	Trad:	1	1			2
	Diff:	(elim)	(elim)			0
Repl Seq	Precond:					
(6 + Cv = 13)	Trad:	0				0
	Diff:	0				0
Extr Func	Precond:	Custom1				
(3 steps)	Trad:	1				1
	Diff:	1				1

PHP

Precondition	**Steps**					
II	4					
						Total
Copy Up	Precond:	Warn	II	Custom		
(1 step)	Trad:	1	4	7		12
	Diff:	1	(elim)	2		3
Pull Up	Precond:					
(2 steps)	Trad:	0				0
	Diff:	0				0

Part I

Fortran

1 Definitions

Body. The statements between the header statement and the end-statement of a construct. E.g., for a ⟨*module*⟩, the Body consists of the statements between the ⟨*module-stmt*⟩ and ⟨*end-module-stmt*⟩.

Declaration. An occurrence of a name that first introduces it into a Lexical Scope. Syntactically, one of the following:

1. ⟨*type-name*⟩ in the immediate context of a ⟨*derived-type-stmt*⟩
2. ⟨*component-name*⟩ in the immediate context of a ⟨*component-decl*⟩
3. ⟨*object-name*⟩ in the immediate context of an ⟨*entity-decl*⟩
4. ⟨*namelist-group-name*⟩ in the immediate context of a ⟨*namelist-stmt*⟩
5. ⟨*common-block-name*⟩ in the immediate context of a ⟨*common-stmt*⟩
6. ⟨*where-construct-name*⟩ in the immediate context of a ⟨*where-construct-stmt*⟩
7. ⟨*forall-construct-name*⟩ in the immediate context of a ⟨*forall-construct-stmt*⟩
8. ⟨*if-construct-name*⟩ in the immediate context of a ⟨*if-then-stmt*⟩
9. ⟨*case-construct-name*⟩ in the immediate context of a ⟨*select-case-stmt*⟩
10. ⟨*do-construct-name*⟩ in the immediate context of a ⟨*label-do-stmt*⟩ or ⟨*nonlabel-do-stmt*⟩
11. ⟨*program-name*⟩ in the immediate context of a ⟨*program-stmt*⟩
12. ⟨*module-name*⟩ in the immediate context of a ⟨*module-stmt*⟩
13. ⟨*local-name*⟩ in the immediate context of a ⟨*rename*⟩ or ⟨*only-rename*⟩
14. ⟨*block-data-name*⟩ in the immediate context of a ⟨*block-data-stmt*⟩
15. ⟨*generic-name*⟩, ⟨*defined-operator*⟩, or = in the immediate context of an ⟨*interface-stmt*⟩
16. ⟨*external-name*⟩ in the immediate context of an ⟨*external-stmt*⟩
17. ⟨*intrinsic-procedure-name*⟩ in the immediate context of an ⟨*intrinsic-stmt*⟩
18. ⟨*function-name*⟩ in the immediate context of a ⟨*function-stmt*⟩
19. ⟨*subroutine-name*⟩ in the immediate context of a ⟨*subroutine-stmt*⟩
20. ⟨*entry-name*⟩ in the immediate context of an ⟨*entry-stmt*⟩
21. ⟨*function-name*⟩ in the immediate context of a ⟨*stmt-function-stmt*⟩
22. The first occurrence of a variable name which causes that variable to become implicitly declared.

Definition. A Declaration that is *not* any of the following:

1. ⟨*external-name*⟩ in the immediate context of an ⟨*external-stmt*⟩
2. ⟨*intrinsic-procedure-name*⟩ in the immediate context of an ⟨*intrinsic-stmt*⟩
3. ⟨*function-name*⟩ or ⟨*subroutine-name*⟩ in the immediate context of a ⟨*function-stmt*⟩ or ⟨*subroutine-stmt*⟩ in the immediate context of an ⟨*interface-body*⟩

Except for COMMON blocks, every entity is assumed to have at most one Definition (assuming the Fortran program is valid).[†]

External Subprogram. A subprogram defined in File Scope; i.e., a ⟨*function-subprogram*⟩ or ⟨*subroutine-subprogram*⟩ in the immediate context of an ⟨*external-subprogram*⟩. (See File Scope.)

File Scope. A ⟨*program*⟩. (A File Scope is one kind of Lexical Scope; see Lexical Scope.[‡])

Global Entity. A Program Unit or a ⟨*common-block*⟩. (§14.1.1) (Note that a Global Entity may have multiple Declarations: An External Subprogram may also be declared in INTERFACE blocks and/or EXTERNAL statements, and a common block will usually be declared in several different COMMON statements in other scopes.)

Host. A program unit that may contain a CONTAINS statement and internal subprograms or module subprograms. Syntactically, one of ⟨*main-program*⟩, ⟨*module*⟩, ⟨*function-subprogram*⟩, ⟨*subroutine-subprogram*⟩, with the exception that a ⟨*function-subprogram*⟩ or ⟨*subroutine-subprogram*⟩ in the immediate context of an

⟨*internal-subprogram*⟩ cannot be a Host. [4, pp. 448, 544]

Import. If a Named Entity in a Scoping Unit S is use associated (§11.3.2) with a Named Entity N from a module M, we will say S *Imports* N from M.

Internal Subprogram. A subprogram following a CONTAINS statement in a Host, i.e., a ⟨*function-subprogram*⟩ or ⟨*subroutine-subprogram*⟩ in the immediate context of an ⟨*internal-subprogram*⟩. [4, pp. 534–537]

Lexical Scope. A ⟨*program*⟩ or a Scoping Unit.‡

Local Entity (Class 1, 2, 3). Cf. §14.1.2. Refactorings herein deal exclusively with Class 1 Local Entities, which are "named variables that are not statement or construct entities (14.1.3), named constants, named constructs, statement functions, internal procedures, module procedures, dummy procedures, intrinsic procedures, generic identifiers, derived types, and namelist group names."

Local Scope. A Scoping Unit. (§14.1.2) [4, p. 534]

Name. A ⟨*name*⟩, or any syntactic construct named ⟨*xyz-name*⟩ (e.g., ⟨*module-name*⟩).

Named Entity. A Name, the assignment symbol "=", or a ⟨*defined-operator*⟩. (§14) [4, p. 532]

Outer Scope. A Lexical Scope that properly contains a given Lexical Scope in a parse tree; i.e., a Lexical Scope which is an ancestor of a given Lexical Scope).

Program Unit. One of: ⟨*main-program*⟩, External Subprogram, ⟨*module*⟩, or ⟨*block-data*⟩. (§11; R202)

Reference. Any occurrence of a Name that is not a Definition.

Scoping Unit. One of: ⟨*derived-type-def*⟩, ⟨*main-program*⟩, ⟨*module*⟩, ⟨*block-data*⟩, ⟨*function-subprogram*⟩, ⟨*subroutine-subprogram*⟩. [4, p. 532]

Subprogram. One of: ⟨*function-subprogram*⟩ or ⟨*subroutine-subprogram*⟩.

Subprogram Part. (The part of a Host that contains Internal Subprograms.) ⟨*module-subprogram-part*⟩ or ⟨*internal-subprogram-part*⟩.

Subroutine. A ⟨*subroutine-subprogram*⟩.

† Some entities may be declared in several locations. For example, an external subroutine may be defined in one file, while an INTERFACE block makes it available in another scope. In such cases, the declaration in the INTERFACE block is both a Declaration and a Reference, but it is *not* a Definition.

‡ Our concept of a Lexical Scope is different from the concept of "scope" in the Fortran standard [4, pp. 534–537]. Specifically, implied-DO variables, FORALL index variables, and statement-function parameters exist in a new scope according to the ISO specification, but for our (refactoring) purposes, we will treat them as references to a local variable in the enclosing scope. Also, the concept of File Scope is new.

2 Requirements

We will assume that the refactoring tool's capabilities are roughly those of a parser coupled with a syntax tree rewriter, name binding analysis (symbol tables), and cross-reference database. This means that the tool is able to construct and traverse a syntax tree, manipulate source code based on that syntax tree, find all Declarations of a Global Entity, find all Declarations in a given Lexical Scope (including implicit variables), find all References to a given Declaration, determine what type of entity a given name refers to (common block, local variable, function, etc.), determine an entity's attributes (PARAMETER, PUBLIC, etc.), find all Lexical Scopes which USE a particular module, and determine what entities are imported from that module.

3 Predicates, Preconditions, & Procedures

3.1 Predicate [LC]: Introducing N into S introduces a local conflict with N'

```
subroutine s
    integer :: n
    common /c/ n
contains
    !! subroutine n cannot be introduced here
    !! subroutine c can be introduced here
end subroutine
```

This determines whether two declarations cannot simultaneously exist in the same Lexical Scope.

Input. A new Named Entity N and an existing Named Entity N' with a Declaration in a Lexical Scope S. N and N' have the same name.

Procedure.

1. If N and N' both name Global Entities, return TRUE. (§14.1.1)
2. If N is the name of a common block and N' names a Local Entity, or vice versa, return FALSE. (§14.1.2)
3. If N is the name of an external procedure and N' is a generic name given to that procedure, return FALSE. (§14.1.2)
4. Otherwise, return TRUE. (§14.1.2)

Notes. This is a compilability check. A compiler uses these same rules when determining if a symbol can be added to the symbol table for a particular scope. If this predicate returns TRUE but N is introduced into S anyway, the program will not compile.

3.2 Predicate [SH]: Named Entity N in S cannot be shadowed in S'

```
subroutine s
    integer :: n
contains
    !! subroutine :: s cannot be introduced here
    subroutine t
        !! integer :: n can be introduced here
        !! integer :: s can be introduced here
        !! integer :: t cannot be introduced here
    end subroutine
end subroutine
```

If there is a Named Entity N defined in S, this check determines if another entity in a contained Lexical Scope S' cannot also be named N.

Input. A Named Entity N defined in a Lexical Scope S, and a Lexical Scope S' contained in S.

Procedure.

1. If S is the File Scope, return FALSE. (Entities defined at File Scope are Global Entities. They are accessible to, but not inherited by, contained scopes.)
2. If S' is a Scoping Unit and N is the name of S', return TRUE. (The name of a main program, module, or subprogram has limited use within its definition. – §11.1, 11.3, 14.1.2)
3. If S' is an Internal Subprogram, return FALSE. (Declarations in Internal Subprograms may shadow Declarations in their Hosts. – §14.6.1.3)
4. Otherwise, return TRUE.

Notes. This is a compilability check. A compiler uses these same rules when determining if a symbol can be added to the symbol table. If this predicate returns TRUE but N is introduced into S' anyway, the program will not compile.

3.3 Predicate [IC]: Introducing N into S introduces conflicts into an importing scope S'

```
module m1                module m2                                   subroutine s
    integer :: a             !! integer :: a cannot be introduced here      use m1; use m2
    integer :: b             !! integer :: b can be introduced here         print *, a
end module               end module                                  end subroutine
```

Suppose a new Named Entity N is to be introduced into a module, and another scope S' imports that module and will import N if it is introduced. This check determines whether S' already contains an entity with the same name as N.

Input. A new Named Entity N, a module S, and a Lexical Scope S' that (directly or indirectly) imports entities from the module S.

Procedure.

1. If there is an entity N' in scope in S' with the same name as N...
   (a) If N' is imported from a module but is unreferenced[†] in S', return FALSE. (§11.3.2) (This includes both the case where N' is imported without renaming and the case where N' is a ⟨*local-name*⟩ for a renamed module entity.)
   (b) If N' is inherited in S' from an Outer Scope, return TRUE iff N' cannot be shadowed by N in S [SH].
   (c) Otherwise, return TRUE iff introducing N introduces a local conflict [LC] with N'.
2. Otherwise, return FALSE.[‡]

Notes. This is a compilability check. If this predicate returns TRUE but N is introduced into S anyway, the program will not compile.

[†] There is some ambiguity as to what "unreferenced" means. The relevant clause of the ISO standard (§11.3.2) states: "Two or more accessible entities, other than generic interfaces, may have the same name only if the name is not used to refer to an entity in the scoping unit." The question is what "refer to" means. Specifically, (1) is `USE M, X => A, X => B` legal if the name *X* is never actually used, and (2) if *M* contains a module entity named *X*, should `USE M, X => A` be permitted (in which case the local name *X* would presumably shadow the module entity *X*)? IBM XL Fortran 12.1, GNU Fortran 4.4.2, PGI Fortran 10.0, and Intel Fortran 10.1 all exhibit different behaviors.

[‡] There may be an entity with the same name in a contained scope, but it will be allowed to shadow the imported entity *N*; cf. [SH].

3.4 Predicate [SK]: Introducing N into S skews references in S'

```
module m
    integer n
contains
    subroutine s
        !! integer :: n cannot be introduced here
        call t
    contains
        subroutine t
            n = 1
        end subroutine
    end subroutine
end module
```

In the above code, a local variable named `n` *cannot be introduced into* `s` *because it would change the meaning of the reference to* `n` *in* `t`*, which could change the behavior of a program. This predicate detects situations such as this.*

Suppose a new Named Entity N is to be introduced into a scope but shadows an existing entity N'. This check determines whether any references to N' will instead become references to N if it is introduced.

Input. A new Named Entity N, a Lexical Scope S into which N is intended to be introduced, and a Lexical Scope S' which is either S itself or a Lexical Scope contained in S.

Procedure.

1. For each reference in S' to a Named Entity N' with the same name as N...

(a) If N' is inherited from a scope S'' (where S' is contained in S''), return TRUE. (§14.6.1.3) (If N is introduced into S', N will shadow N', changing the reference.)

(b) If N' is a reference to a procedure whose name has not been established (§14.1.2.4.3), return TRUE. (If N is introduced into S', the name will be established, changing the reference.)

2. For each Lexical Scope S'' contained in S', return TRUE if introducing N into S skews references [SK] in S''.

3. Otherwise, return FALSE.

Notes. This is both a compilability and a semantic preservation check. If bindings are skewed, say, from a variable to a subroutine, the program will not compile; if they are skewed, e.g., from one variable to another, behavior might not be preserved. In any case, if this predicate returns TRUE, name bindings will not be preserved if the transformation proceeds.

3.5 Precondition [IN]: Introducing N into S must be legal and name binding-preserving

This precondition makes two guarantees: (1) if a particular declaration is added to a program, the resulting program will compile (i.e., the addition of the declaration is legal); and (2) if the declaration will shadow another declaration, it will not inadvertently change references to the shadowed declaration.

Input. A new Named Entity N and a Lexical Scope S.

Procedure.

1. If there is a Named Entity N' in scope in S which has the same name as N...

   (a) If N' is local to S or is imported into S, FAIL if introducing N introduces a local conflict [LC] with N' in S.

   (b) If N' is declared in an Outer Scope, FAIL if N' cannot be shadowed [SH] by N in S.

   (c) FAIL if the introduction of N in S skews references [SK] in S.

2. For each Lexical Scope S' contained in S, if there is a Named Entity N' with the same name as N that is local to S' or is imported into S', FAIL if N cannot shadow [SH] N' in S'.

3. For each Lexical Scope S' that imports S, if S' will import N due to the absence of an ONLY clause...

   (a) FAIL if the introduction of N in S introduces conflicts [IC] into the importing scope S'.

   (b) FAIL if the introduction of N in S' skews references [SK] in S'.

4. PASS.

Notes. This precondition combines the previous four predicates into a single check which guarantees that, if N is introduced into S, then (1) the program will compile, and (2) name bindings will be preserved. The previous four predicates enumerate all of the conditions required for this guarantee to be made *a priori.* In a differential refactoring engine, this precondition can be eliminated entirely, since introducing N into S and testing for compilability and name binding preservation satisfies this precondition's checks: If name bindings will be skewed, predicate [SK] will fail. If the resulting program will not compile, one of predicates [LC], [SH], or [IC] will fail.

3.6 Precondition [SI]: Non-generic Internal Subprogram S must have only internal references

This precondition guarantees that there are no calls to a given Internal Subprogram except for directly recursive calls.

Input. An Internal Subprogram S in a Host H. S must not be a generic subprogram.

Procedure.

1. For each Reference R to S, FAIL if *neither* of the following hold:

(a) R occurs in the context of an $\langle$*access-stmt*$\rangle$ in the $\langle$*specification-part*$\rangle$ of H.
(b) R occurs in the Definition of S.

2. Pass.

3.7 Predicate [PR]: Private Entities in D are referenced outside D

Given a set D of module entities, this predicate determines whether any entities in D with PRIVATE *visibilities are referenced by definitions that are not in D.*

Input. A set D of Named Entity Definitions in a Module M.

Procedure.
1. For each Named Entity N in D...
   (a) If N has PRIVATE visibility, then...
      i. For each Reference R to N...
         A. If R does not occur in the Definition of an entity in D, return TRUE.
2. Return FALSE.

Notes. See Precondition [PP] and the refactoring Move Module Entities.

3.8 Procedure [Ou]: Determine Named Entities in $M - D$ referenced by D

Given a set D of module entities, this predicate determines whether any definitions in D reference entities in the module that are not included in D.

Input. A set D of Named Entity Definitions in a Module M.

Output. A set E of Named Entities in a Module M.

Procedure.
1. Initially, let $E \coloneqq \emptyset$.
2. For each Named Entity N in D...
   (a) For each Reference R in the Definition of N...
      i. If R names a public module entity from M that is not in the set D, define $E \coloneqq E \cup \{N\}$.
3. Return E.

3.9 Predicate [OU]: D references Named Entities in M outside D

Given a set D of module entities, this predicate determines whether any definitions in D reference entities in the module that are not included in D.

Input. A set D of Named Entity Definitions in a Module M.

Procedure.
1. Determine the set E of Named Entities in $M - D$ referenced by D [Ou].
2. Return TRUE iff $E \neq \emptyset$.

Notes. See Precondition [PP] and the refactoring Move Module Entities.

3.10 Precondition [PP]: D must partition private references in M

Given a set D of module entities, this precondition ensures that references to PRIVATE *entities occur such that either (1) both the entity and the reference are in D, or (2) neither the entity nor the reference is in D.*

Input. A set D of Named Entity Definitions in a Module M.

Procedure. Let $\overline{D}$ denote the set of all module entities declared in M that are not members of the set D.

1. FAIL if private entities in D are referenced outside D [PR].
2. FAIL if private entities in $\overline{D}$ are referenced outside $\overline{D}$ [PR].
3. PASS.

3.11 Procedure [Pr]: Construct a Set of Pairs from USE Statement U

Given a USE *statement, this procedure returns a set of ordered pairs which model the module entities imported by that* USE *statement. The first component of each pair is the name of the module entity; the second component is its name in the local scope, which may be the same or different from the original name. For example, suppose a module* `MOD` *contains entities named* `a`*,* `b`*, and* `c`*. For the statement* `USE MOD`*, this procedure would return* $\{(a,a),(b,b),(c,c)\}$*; for the statement* `USE MOD, x => c`*, it would return* $\{(a,a),(b,b),(c,x)\}$*; and for the statement* `USE MOD, ONLY: a, x => b`*, it would return* $\{(a,a),(b,x)\}$.

Input. A ⟨*use-stmt*⟩ U.

Output. A set of ordered pairs of Names.

Procedure. Let N_M denote the set of names of all public entities in the module referenced by U.

1. If U contains neither a ⟨*rename-list*⟩ nor an ⟨*only-list*⟩, return

$$\bigcup_{N \in N_M} (N,N).$$

2. If U contains a ⟨*rename-list*⟩, return

$$\bigcup_{N \in N_M} \begin{cases} \{(N,N')\} & \text{if } N' \texttt{ => } N \text{ appears in the } \langle \textit{rename-list} \rangle \text{, for some } N' \\ \{(N,N)\} & \text{if } N \text{ does not appear as a } \langle \textit{use-name} \rangle \text{ in the } \langle \textit{rename-list} \rangle \end{cases}$$

3. If U contains an ⟨*only-list*⟩, return

$$\bigcup_{N \in N_M} \begin{cases} \{(N,N')\} & \text{if } N' \texttt{ => } N \text{ appears in the } \langle \textit{only-list} \rangle \text{, for some } N' \\ \{(N,N)\} & \text{if } N \text{ appears in the } \langle \textit{only-list} \rangle \\ \varnothing & \text{if } N \text{ does not appear in the } \langle \textit{only-list} \rangle \end{cases}$$

3.12 Procedure [Us]: Construct a USE Statement for Module M from Sets of Pairs X and Y

This procedure is essentially the opposite of Procedure [Pr]: It takes as input a set of ordered pairs and uses them to construct a USE *statement. For example, for the module name* `mod` *and ordered pairs* $\{(a,a),(b,x)\}$*, it would return the statement* `USE MOD, ONLY: a, x => b`*.*

Input.

1. A Name M of a module.
2. A set X of ordered pairs of Names. (This set denotes the entities that the USE statement should import.)

3. A set Y of ordered pairs of Names of entities with public visibility in M. (This set denotes *all* of the public entities available to import from M. This set is provided as input to accommodate the Move Module Entities refactoring: it will need to construct a USE statement assuming that some entities have been moved out of one module and into another.)

Output. A new $\langle$*use-stmt*$\rangle$ U.

Procedure.

1. If $\{N \mid \exists N'.\ (N, N') \in X\} = \{L \mid \exists L'.\ (L, L') \in Y\}$, then every entity in M is imported.
   (a) If $X = Y$, then every entity in M is imported, and no entites are renamed, so return
   $$\langle \textit{use-stmt} \rangle \quad \leftarrow \quad \texttt{use}\ \ M \hookleftarrow$$
   (b) If $\{N \mid \exists N' \neq N.\ (N, N') \in X\} \neq \emptyset$, then every entity in M is imported, but at least one entity is renamed. Let $(N_1, N_1'), (N_2, N_2'), \ldots, (N_k, N_k')$ denote the members of the set $\{(N, N') \in X \mid N \neq N'\}$, and return
   $$\langle \textit{use-stmt} \rangle \quad \leftarrow \quad \texttt{use}\ \ M\texttt{,}\ \ N_1'\ \texttt{=>}\ N_1\texttt{,}\ \ N_2'\ \texttt{=>}\ N_2\texttt{,}\ \ \ldots\texttt{,}\ \ N_k'\ \texttt{=>}\ N_k \hookleftarrow$$
2. Otherwise, not all members of M are imported.
   (a) Initally, let U denote the $\langle$*use-stmt*$\rangle$
   $$\langle \textit{use-stmt} \rangle \quad \leftarrow \quad \texttt{use}\ \ M\texttt{, only:}\ \hookleftarrow$$
   which has an empty $\langle$*only-list*$\rangle$.
   (b) For each pair (N, N') in X...
      i. If $N = N'$, append
      $$\langle \textit{only-use-name} \rangle \quad \leftarrow \quad N$$
      to the $\langle$*only-list*$\rangle$ of U (with a separating comma, if necessary).
      ii. If $N \neq N'$, append
      $$\langle \textit{only-rename} \rangle \quad \leftarrow \quad N'\ \texttt{=>}\ N$$
      to the $\langle$*only-list*$\rangle$ of U (with a separating comma, if necessary).
   (c) Return U.

3.13 Precondition [RN]: Module M' must not rename entities D from Module M

Given a set D of entities defined in a module M, this precondition ensures that, if any entities in D are directly imported into M′, they are not renamed.

Input. A set D of Named Entity Definitions in a Module M.

Procedure.

1. If M' contains a $\langle$*use-stmt*$\rangle$ U' with a $\langle$*module-name*$\rangle$ naming M...
   (a) Construct a set of pairs X from U' [Pr].
   (b) If X contains an element (N, N') where $N \in D$ and $N \neq N'$, FAIL.
2. PASS.

3.14 Procedure [Rn]: Replace References in C according to X

This procedure replaces occurrences of one name with a different name.

Input.

1. A set X of ordered pairs (N, N') where N is an existing Name and N' is a new Name.
2. Any syntactic construct C.

Output. C is modified such that References to N have their name changed to N'.

Procedure.

1. For each pair $(N, N') \in X$...
   (a) For each Reference R to N in C...
      i. Replace the occurrence of N in R with N'.

4 Refactorings

4.1 Add Empty Internal Subroutine

Requires: [LC],[SH],[IC],[SK],[IN]

This refactoring adds a new Subroutine as an Internal Subprogram of a given Host. The Subroutine initially has an empty body. The refactoring fails if the Subroutine will conflict with an existing declaration. Although this refactoring may be used by itself, but it is perhaps more useful as a building block for other refactorings (like Extract Subroutine).

Input.

1. A Host H into which the empty subroutine will be added as an internal subprogram.
2. A new Name N for the subroutine.

Preconditions. Introducing an Internal Subprogram into H with name N must be legal and name binding-preserving [IN].

Transformation.

1. ♦ If H does not contain a Subprogram Part, append to H

   Subprogram Part ←
   ```
   contains ↲
     subroutine N ↲
     end subroutine ↲
   ```

2. ♦ If H contains a Subprogram Part P, append to P

   ⟨*internal-subprogram*⟩ ←
   ```
   subroutine N ↲
   end subroutine ↲
   ```

Notes. In a differential refactoring engine, precondition [IN] can be eliminated as described in its description. The new subroutine must not shadow an existing entity (i.e., skew references), which would be manifested as an incoming binding. It must not conflict with an existing entity, which would result in a compilation error. The addition of a new subroutine cannot introduce an outgoing name binding (although introducing a new function could, depending on its return type). Control flow and du-chains are intraprocedural and, therefore, are unaffected. The only new name binding edge will be an internal edge from the ⟨*end-name*⟩ to the ⟨*subroutine-name*⟩. Therefore, the differential version of this refactoring consists of a single step: introducing the subroutine with rule $N^{\circlearrowleft}_{\subseteq}$.

4.2 Safe-Delete Non-Generic Internal Subprogram

Requires: [SI]

This refactoring removes an Internal Subprogram from a given Host. The refactoring fails if there are any references to the subprogram.

Input. An Internal Subprogram S in a Host H.

Preconditions. S must have only internal references [SI].

Transformation.

1. For each Reference to S in an ⟨*access-stmt*⟩ A in the ⟨*specification-part*⟩ of H...
   (a) ♦ If the ⟨*access-id-list*⟩ of A contains only one ⟨*access-id*⟩ (i.e., a ⟨*use-name*⟩ with the name of S), remove A.
   (b) ♦ If there is more than one ⟨*access-id*⟩ in the ⟨*access-id-list*⟩ of A, remove the ⟨*use-name*⟩ with the name of S and an appropriate adjacent comma.
2. ♦ If H contains only one Internal Subprogram (S), remove the Subprogram Part of H.
3. ♦ If H contains more than one Internal Subprogram, remove S.

Notes. This specification requires that the subprogram not be used in an ⟨*interface-block*⟩. Extending the refactoring to remove this restriction is straightforward.

In a differential refactoring engine, the precondition [SI] can be eliminated. There must be no incoming bindings to the entity to delete; deleting a referenced subroutine would be manifested as a missing incoming binding. A subroutine may contain variable references and subroutine calls, so outgoing bindings may be deleted. Internal name binding edges, representing recursive calls and the link from the ⟨*end-name*⟩ to the ⟨*subroutine-name*⟩, will also be deleted. Control flow and du-chains are intraprocedural and, therefore, are unaffected. Therefore, this refactoring consists of a single step: deleting the subroutine according to rule $N^{\rightarrow\circlearrowright}_{\subseteq\subseteq}$.

4.3 Rename

Requires: [IN],[LC],[SH],[SK],[IC]

This refactoring changes the name of an entity, both in declarations and references. It fails if the new name will conflict with an existing name, or if it will shadow an existing name in such a way that existing name bindings will change.

Input.

1. A Declaration of a Name N in a Lexical Scope S. N must designate a Global Entity or Class 1 Local Entity.
2. A new Name N' for N.

Preconditions.

1. Introducing N' into S must be legal and name binding-preserving [IN].
2. Warn if a reference to N appears in the context of a ⟨*namelist-group-object*⟩: To preserve behavior, the user may need to manually update input files to reflect the new variable name.
3. If N names a subprogram, matching declarations in interface blocks should uniquely bind to N.

Transformation.

1. For each Declaration D of the Named Entity N...
   (a) ♦ Replace D with N'.
   (b) For each Reference R to D...
      i. ♦ Replace R with N'.

Notes. In a differential refactoring engine, precondition [IN] can be eliminated as described in its description. This refactoring should preserve the program graph in its entirety.

4.4 Introduce Implicit None

Requires: none

This refactoring adds an implicit none *into a Lexical Scope and all nested scopes and adds type declaration statements for all implicit variables. Its specification is greatly simplified by the infrastructural assumptions stated in Section 2.*

Input. A Lexical Scope S.

Preconditions. (none)

Transformation.

1. If implicit none does *not* appear in the ⟨*specification-part*⟩ of S...

   Let I' denote

   ⟨*implicit-stmt*⟩ ⟵ `implicit none` ↲

   (a) ♦ If an ⟨*implicit-stmt*⟩ I appears in the ⟨*specification-part*⟩ of S, replace I with I'.
   (b) ♦ If such an ⟨*implicit-stmt*⟩ does *not* appear, insert I' into the ⟨*specification-part*⟩ of S. (Note that the Fortran grammar requires that I' appear after all occurrences of ⟨*use-stmt*⟩ but before all occurrences of ⟨*declaration-construct*⟩.)

(c) For each implicitly-typed variable N which is local to S . . .

Let T be a new ⟨*type-spec*⟩ corresponding to the type of N. (If the ⟨*implicit-stmt*⟩ I existed in Step 1a above, it is preferable to copy the concrete syntax of the ⟨*type-spec*⟩ from the existing ⟨*implicit-stmt*⟩, when possible, in order to ensure that formatting and symbolic representations of kinds are reproduced verbatim.)

i. ♦ Insert the following into the ⟨*specification-part*⟩ of S:

$$\langle \textit{declaration-construct} \rangle \quad \leftarrow \quad T \; :: \; N \hookleftarrow$$

2. Repeat Step 1 for each Lexical Scope S' contained in S.

Notes. This refactoring has no preconditions, since it is always legal to add explicit type declaration statements. If a scope is already IMPLICIT NONE, the transformation has no effect.

In a differential refactoring engine, this transformation will change name bindings such that they point to the variable declaration rather than the first occurrence of the variable name (which implicitly declared the variable). Therefore, the affected forest must consist of both the first occurrence of the variable and the explicit declaration; then, the refactoring will introduce a new internal name binding edge (from the first use to the explicit declaration) but will otherwise preserve the program graph in its entirety. Therefore, this refactoring consists of a single step: introducing the explicit declaration according to rule $N^{\circlearrowleft}_{\supseteq}$.

4.5 Permute Subroutine Parameters

Requires: none

This refactoring permutes the arguments to a subroutine, adjusting any call sites accordingly. Note that, if the actual arguments at a call site include function invocations with side effects, reordering these function calls may not preserve behavior.

Input.

1. A ⟨*subroutine-subprogram*⟩ S with n dummy arguments, $n \geq 2$, and
2. A permutation $\sigma = \begin{pmatrix} 1 & 2 & \cdots & n \\ j_1 & j_2 & & j_n \end{pmatrix}$ providing a new order for the arguments of S.

Preconditions.

1. Alternate return specifiers must retain the same relative order. That is, if the ⟨*dummy-arg-list*⟩ in S's ⟨*subroutine-stmt*⟩ has * for the ⟨*dummy-arg*⟩s at indices $i_1, i_2, \ldots, i_k$ where $i_1 < i_2 < \cdots < i_k$, then $\sigma(i_1) < \sigma(i_2) < \cdots < \sigma(i_k)$.
2. The permutation must not place an optional argument before an alternate return.
3. Matching declarations in INTERFACE blocks should uniquely bind to S.
4. (Checked during transformation)

Transformation.

1. ♦ Permute the ⟨*dummy-arg*⟩s in the ⟨*dummy-arg-list*⟩ of S's ⟨*subroutine-stmt*⟩ according to σ.
2. For each ⟨*call-stmt*⟩ C which references S . . .

The ⟨*actual-arg-spec-list*⟩ of C contains m ⟨*actual-arg-spec*⟩s, for some $m \leq n$.

(a) Initially, let $K :=$ FALSE.

(b) Initially, let L' be an empty ⟨*actual-arg-spec-list*⟩.

(c) For $i := \sigma(1), \sigma(2), \ldots, \sigma(n)$:

Let D denote the i-th dummy argument of S *before* its dummy arguments were permuted. If C contains an ⟨*actual-arg-spec*⟩ corresponding to D, denote it by A_i.

i. If A_i is not defined, define $K \coloneqq$ TRUE. (An OPTIONAL argument was omitted, so all subsequent arguments must have keywords.)

ii. If A_i is defined...

Let A_i denote the ⟨*actual-arg-spec*⟩.

A. If A_i contains ⟨*keyword*⟩ =, define $K \coloneqq$ TRUE.

B. If K = FALSE or A_i contains ⟨*keyword*⟩ =, append A_i (with a separating comma, if necessary) to L'.

C. ◇ FAIL if K = TRUE and A_i is an alternate return argument. (Permuting call sites must not place an alternate return argument after an argument with ⟨*keyword*⟩ =, since every subsequent actual argument must contain ⟨*keyword*⟩ =, but alternate return arguments cannot be given keywords.)

D. If K = TRUE and A_i does not contain ⟨*keyword*⟩ =, let N denote the ⟨*dummy-arg-name*⟩ of the i-th ⟨*dummy-arg*⟩ in S's ⟨*subroutine-stmt*⟩ before it was permuted, and append

⟨*actual-arg-spec*⟩ ← $N = A_i$.

to L'.

(d) ♦ Replace C's ⟨*actual-arg-spec-list*⟩ with L'.

3. For each ⟨*subroutine-stmt*⟩ S' in the context of an ⟨*interface-block*⟩ such that S' matches S...

   (a) ♦ Permute the ⟨*dummy-arg-list*⟩ of S' according to σ.

Notes. In a differential refactoring engine, none of this refactoring's preconditions can be eliminated, because they are all related to input validation rather than compilability or preservation checking.

4.6 Add Use of Named Entities E in Module M to Module M' [Prerequisite]

Requires: [IN],[LC],[IC],[SH],[SK]

This refactoring adds the statement `use M, only: E` *to the module M', if a similar statement does not already exist. It fails if this will result in a naming conflict, the introduction of circular dependencies between modules, or if a statement* USE *M already exists but renames entities in E.*

Input.

1. A Module M.
2. A set E of public Named Entities in M.
3. A distinct Module M'. The statement USE M will be inserted into M', if necessary.

Preconditions.

1. FAIL if M uses M'. (It would be necessary to introduce the statement USE M into M', but this would introduce a circular dependency.)
2. (Checked during transformation)

Transformation.

1. If M' contains a ⟨*use-stmt*⟩ U' with a ⟨*module-name*⟩ naming M, and U' contains an ⟨*only-list*⟩...

   (a) For each Named Entity N in E that does not occur as a ⟨*use-name*⟩ in the context of U''s ⟨*only-list*⟩...

   i. ◇ Ensure that introducing N into M' is legal and name binding-preserving [IN].

   ii. ♦ Append a separating comma and

   ⟨*only*⟩ ← N

   to the ⟨*only-list*⟩ of U'.

2. If M' does not contain a ⟨*use-stmt*⟩ with a ⟨*module-name*⟩ naming M...

Let $E_1, E_2, \ldots, E_k$ denote the elements of E.

(a) For each Named Entity E_i, $1 \le i \le k\ldots$

i. ◊ Ensure that introducing E_i into M' is legal and name binding-preserving [IN].

(b) ♦ Insert the statement

⟨*use-stmt*⟩ ← `use` M`, only:` E_1`,` E_2`, ...,` E_k ↲

into the ⟨*specification-part*⟩ of M'.

Notes. This refactoring fails precondition checking if a USE statement already exists but renames an entity in E: This is to simplify Move Module Entities, for which this refactoring is a prerequisite. Instead, Move Module Entities could rename references according to the new local names.

The first precondition can be eliminated in a differential refactoring engine since introducing a circular dependency will result in a compilation error. The checks for precondition [IN] can also be eliminated as described in its description. Adding the USE statements will introduce outgoing name bindings (to the imported entities), but the used names should be unreferenced; therefore, the USE statements should be inserted according to rule $N_{\subseteq}^{\rightarrow}$.

4.7 Move Module Entities

Requires: [OU],[Ou],[LC],[RN],[PP],[PR],[Pr],[Us],[Rn]

This refactoring moves a set of entities from one module to another, updating USE *statements as necessary. It fails if the changes will result in a naming conflict, a visibility problem, or the introduction of circular dependencies between modules.*

Allowing the user to move a set of entities often simplifies the refactoring process since it allows a PRIVATE *module variable and all of the procedures that use it to be moved at once. If they are moved one at a time, it becomes necessary to temporarily increase the visibility of the module variables in the interim.*

There are 21 different declaration constructs that can appear in a ⟨module⟩. To keep this specification to a resonable length, we require the entities to move to be referenced only in ⟨type-declaration-stmt⟩s, ⟨access-stmt⟩s, and procedure definitions (see Precondition 1a). Extending it to support other constructs should be straightforward.

Input.

1. A set D of Named Entity Declarations in a Module M.
2. A distinct Module M' into which the entities will be moved.

Preconditions.

1. For each Named Entity N in $D\ldots$
   (a) For each reference R to N which occurs in the context of $M\ldots$
      i. If R does *not* occur in the context of any of the following, FAIL:
         - ⟨*type-declaration-stmt*⟩
         - ⟨*access-stmt*⟩
         - ⟨*subroutine-subprogram*⟩
         - ⟨*function-subprogram*⟩
   (b) For each Named Entity N' declared in $M'\ldots$
      i. Introducing N into M' must not introduce a local conflict [LC] with N'.
   (c) Introducing N into M' must not skew references [SK] in M'.
2. M' must not rename entities D from M [RN].
3. D must partition private references in M [PP].

Transformation.

1. *(If any of the entities being moved from M use other entities in M, add* `use M` *to M'.)* If D references Named Entities in M outside D [OU]...

Let E denote the set of Named Entities in M outside D that are referenced by D.

(a) ♦ Add Use of Entities E in M to M' [Prerequisite].

(b) Construct a Set U_E of Pairs from the Use Statement [Pr] created in the previous step. Let X denote the set $\{(N, N') \in U_E \mid N \neq N'\}$.

Let $\overline{D}$ denote the set of all module entities declared in M that are not members of D.

2. *(If any of the entities being moved from M are used by other entities in M that are not being moved, add* `use M'` *to M.)* If $\overline{D}$ references Named Entities in M outside $\overline{D}$ [OU]...

   Let E denote the set of Named Entities in M outside $\overline{D}$ that are referenced by D.

   (a) ♦ Add Use of Entities E in M' to M [Prerequisite].

3. *(If M′ already contained a ⟨use-stmt⟩, remove any of the references to the entities that are being moved, since they will no longer be in M.)* If M' contains a ⟨*use-stmt*⟩ U' with a ⟨*module-name*⟩ naming M...

   (a) If U' contains an ⟨*only-list*⟩...

      i. ♦ If every ⟨*only-use-name*⟩ in the ⟨*only-list*⟩ is in D, remove the ⟨*use-stmt*⟩ U'.

      ii. ♦ Otherwise, remove from the ⟨*only-list*⟩ every ⟨*only*⟩ whose ⟨*use-name*⟩ is in D (also removing an appropriate adjacent comma).

4. *(Update* USE *statements.)* For each ⟨*use-stmt*⟩ U with a ⟨*module-name*⟩ naming M...

   Let S denote the Lexical Scope containing the ⟨*use-stmt*⟩.

   If S contains a ⟨*use-stmt*⟩ whose ⟨*module-name*⟩ names M', let U' denote this ⟨*use-stmt*⟩.

   (a) Construct a set U_M of pairs from U [Pr].

   (b) If U' does not exist, define $U_{M'} := \emptyset$; otherwise, Construct a set $U_{M'}$ of pairs from U' [Pr].

   Let U_D denote the subset of U_M consisting of pairs whose first component names an entity in D. $U_D := \{(Q, Q') \mid Q \in D \wedge (Q, Q') \in U_M\}$.
   Let P_M denote the set of pairs of public entities in M and P_D denote the subset of P_M consisting of pairs whose first component names an entity in D. $P_D := \{(C, C') \mid C \in D\}$.

   (c) Construct a Use Statement K for Module M with $X := U_M - U_D$ and $Y := P_M - P_D$ [Us].

   (d) Construct a Use Statement K' for Module M' where $X := U_{M'} \cup U_D$ and $Y := P_M \cup P_D$ [Us].

   (e) i. ♦ If K does not have an empty ⟨*only-list*⟩, replace U with K.

      ii. ♦ If K has an empty ⟨*only-list*⟩, remove U.

   (f) i. ♦ If U' exists, then remove U'.

      ii. ♦ If K' does not have an empty ⟨*only-list*⟩, insert K' into S.

5. *(Move the declarations from M to M′.)* For each Named Entity N in D...

   (a) If N is a variable, and its Declaration is a ⟨*type-declaration-stmt*⟩ T...

      i. ♦ If X is defined (from Step 1b), replace references in T according to X [Rn].

      ii. ♦ If T's ⟨*entity-decl-list*⟩ contains only one ⟨*entity-decl*⟩, (i.e., an ⟨*entity-decl*⟩ with the name of N), move T into the list of ⟨*declaration-construct*⟩s in M'.

      iii. If T's ⟨*entity-decl-list*⟩ contains more than one ⟨*entity-decl*⟩...

         Let E denote the ⟨*entity-decl*⟩ with the name of N in T's ⟨*entity-decl-list*⟩.

         A. Create a copy T' of T.

B. Replace T''s ⟨*entity-decl-list*⟩ with a list containing the single entry E.

C. ♦ Remove E and an appropriate adjacent comma from T.

D. ♦ Insert T' into the list of ⟨*declaration-construct*⟩s in M'.

(b) If N is a Subprogram whose Definition occurs in the context of a ⟨*module-subprogram*⟩ S...

i. ♦ If X is defined (from Step 1b), replace references in S according to X [Rn].

ii. ♦ If M' does not contain a ⟨*module-subprogram-part*⟩, move S to construct the ⟨*module-subprogram-part*⟩ of M':

⟨*module-subprogram-part*⟩ ← `contains` ↲
S

iii. ♦ If M' contains a ⟨*module-subprogram-part*⟩ P, move S into P.

(c) For each Reference R to N...

i. If R occurs in the context of an ⟨*access-stmt*⟩ A and A has not been moved into M' by the following step...

A. ♦ If every ⟨*access-id*⟩ references a Named Entity in D, move A into the list of ⟨*declaration-construct*⟩s in M'.

B. ♦ Otherwise...

Let S denote the ⟨*access-spec*⟩ of A.

(1) Remove the ⟨*use-name*⟩ of R and an appropriate adjacent comma.

(2) Insert a new ⟨*access-stmt*⟩

⟨*access-stmt*⟩ ← S :: R ↲

into the list of ⟨*declaration-construct*⟩s in M'.

6. ♦ If, after completing Step 5, the ⟨*module-subprogram-part*⟩ of M is empty but M still contains a ⟨*contains-stmt*⟩, remove the ⟨*contains-stmt*⟩ from M.

Notes. In a differential refactoring engine, precondition [PP] can be eliminated: When entities are moved from M to M', name bindings to PRIVATE entities in M will be eliminated (or skewed), resulting in a preservation failure. Precondition [RN] can also be eliminated, since the renamed entities will no longer exist, resulting in a compilation error and/or skewed bindings. Checks [LC] and [SK] can be eliminated from Step 1 as described in their descriptions. Step 1(a)(i) cannot be eliminated since it restricts the number of constructs on which the transformation can operate. The preservation analysis is only applied in Step 5 (after the USE statements have been updated): new incoming name binding edges (from the updated USE statements) will appear, but, otherwise, name bindings should be preserved. Therefore, this step proceeds according to rule $N^{\leftrightarrows}_{\supseteq}$.

Part II

BC

5 Definitions

Array Declaration. A declaration of an array variable in a $\langle$*define_list*$\rangle$: LETTER `[ ]`

Global Variable. A LETTER.

Name. A LETTER.

Scalar Declaration. A declaration of a scalar variable in a $\langle$*define_list*$\rangle$: LETTER

Variable Declaration. An Array Declaration or a a Scalar Declaration.

6 Predicates, Preconditions, & Procedures

6.1 Procedure [Ds]: Compute Dynamic Shadowing for Program *P*

Since BC is dynamically scoped, this procedure uses a simple, interprocedural data flow analysis to determine what local variables may be accessed by other functions.

Input. A $\langle$*program*$\rangle$ *P*.

Output. A function *uses*, which maps a $\langle$*function*$\rangle$ to a set of Variable Declarations.

Procedure.

1. *(Compute the call graph for P.)* Construct a directed graph whose node set consists of all $\langle$*function*$\rangle$s in *P* and the whose edges are determined by the **calls** relation defined as follows:
   (a) For each $\langle$*function*$\rangle$ *F* in *P*...
      i. For each $\langle$*expression*$\rangle$ in the context of *F* which has the form $G(A)$ for some LETTER *G* and $\langle$*opt_argument_list*$\rangle$ *A*...
         A. Define *F* **calls** *G*.
2. *(Compute the solution to the reaching definitions problem on the call graph.)*
   Let *X* denote the set of all Variable Declarations in *P*.
   (a) For each $\langle$*function*$\rangle$ *F* in *P*...
      i. Define *gen*(*F*) to be the set of all Variable Declarations in *F*. (Note that this is a subset of *X*.)
      ii. Define *kill*(*F*) to be the set of all Variable Declarations in *X* which have the same name as a Variable Declaration in *F*.
      iii. Initially, let *reaches(F)* := ∅.
   (b) For each $\langle$*function*$\rangle$ *G* in *P*...
      i. Let *reaches*(*G*) be the least solution to the equation

$$reaches(G) = \bigcup_{F \,\in\, \mathbf{calls}^{-1}(G)} (gen(F) \cup (reaches(F) \cap \neg kill(F)))\,.$$

3. *(Compute du-chains on the call graph.)* Define a function *uses* as follows.
   For each $\langle$*function*$\rangle$ *F* in *P*...
   (a) For each reference in *F* to a variable *V*...
      i. If *F* does not contain a Variable Declaration for *V*...
         A. every Variable Declaration in *reaches*(*F*) with the name *V* is included in *uses*(*F*).
         B. every Global Variable with the name *V* is included in *uses*(*F*).
4. Return the function *uses*.

6.2 Precondition [IN]: Introducing Variable Declaration V into Function F must be legal and name binding-preserving

This precondition makes two guarantees: (1) if a particular declaration is added to a program, the it will not introduce duplicate local variables, and (2) if the declaration will shadow another declaration, it will not inadvertently change references to the shadowed declaration.

Input. A new Variable Declaration V and a Function F.

Procedure. Compute dynamic shadowing for F [Ds] to obtain the function *uses*.

1. FAIL if the ⟨*opt_auto_define_list*⟩ in the context of F contains a declaration matching V.
2. FAIL if *uses*(F) contains a Variable Declaration with the same name as V which does not occur in the context of F.
3. PASS.

6.3 Procedure [Cv]: Classify Local Variables in Statement Sequence S

This procedure is used by Extract Function to determine which local variables need to be passed as parameters to, and/or returned from, the extracted function.

Input.

1. A sequence $S := S_1, S_2, \ldots, S_n$ of consecutive ⟨*statement*⟩s from a ⟨*statement_list*⟩ in the immediate context of a ⟨*function*⟩ F.

Output.

1. A set X of Variable Declarations.
2. A function *isParam* : $X \to \{\text{TRUE}, \text{FALSE}\}$.
3. A function *isReturn* : $X \to \{\text{TRUE}, \text{FALSE}\}$.

Procedure.

1. Initially, define $X := \emptyset$.
2. For each local variable V declared in F...
   (a) If V is referenced in S...
      i. Define $X := X \cup \{V\}$.
      ii. If there is a du-chain for V whose definition lies outside S and whose use lies inside S, define *isParam*(V) := TRUE. Otherwise, define *isParam*(V) := FALSE.
      iii. If there is a du-chain for V whose definition lies inside S and whose use lies outside S, define *isReturn*(V) := TRUE. Otherwise, define *isReturn*(V) := FALSE.
3. Return X, *isParam*, and *isReturn*.

7 Refactorings

7.1 Add Unreferenced Local Variable Declaration [Prerequisite] Requires: [IN] [Ds]

This refactoring adds a declaration for an (unused) local variable.

Input.
1. A Variable Declaration V.
2. A Function F in which V will be declared as a local variable.

Preconditions.
1. Introducing V into F must be legal and name binding-preserving [IN].

Transformation.
1. If F contains an ⟨*opt_auto_define_list*⟩ L,
   (a) ♦ If L is nonempty, append

   list element ← , V

   to the ⟨*define_list*⟩ of L.

   (b) ♦ If L is empty, append

   list element ← V

   to the empty ⟨*define_list*⟩ of L.
2. ♦ If F does not contain an ⟨*opt_auto_define_list*⟩, insert

   ⟨*opt_auto_define_list*⟩ ← `auto` V

   to the ⟨*define_list*⟩ of L.

Notes. In a differential refactoring engine, Precondition [IN] can be eliminated, since, depending on the implementation, a conflict will either result in a compilability error or the introduction of additional (ambiguous) name binding edges, and shadowing will result in skewed name binding edges. The new variable should have no incoming name bindings. Therefore, this refactoring should preserve the program graph in its entirety.

7.2 Replace Statement with Block [Prerequisite] Requires: (none)

This refactoring replaces a statement S with a block { S }.

Input. A ⟨*statement*⟩ S.

Preconditions. None.

Transformation. ♦ Replace S with

⟨*statement*⟩ ← { S }

Notes. This refactoring is always legal: If S is a ⟨*statement*⟩, { S } is also a ⟨*statement*⟩, according to the BC grammar. The BC specification does not contain any extra-grammatical restrictions on where particular statements may or may not occur.

7.3 Insert Assignment to Unreferenced Local Variable [Prerequisite] Requires: none

This refactoring inserts an assignment statement which assigns the value 0 to an otherwise unreferenced local variable.

Input.
1. A Scalar Variable V to be assigned.
2. A ⟨*statement_list*⟩ into which an assignment statement will be inserted, and the position at which it should be inserted.

Preconditions. There must be no references to V.

Transformation. ♦ Insert

$$\langle statement \rangle \leftarrow V \texttt{ = 0} \hookleftarrow$$

at the given position in the given ⟨*statement_list*⟩.

Notes. The precondition for this refactoring is automatically satisfied when this refactoring is part of the Extract Function refactoring. Nevertheless, it is unnecessarily strong: The purpose of the precondition is to avoid introducing an assignment that would change the behavior of the program, i.e., to avoid introducing a new def-use edge. The assignment statement will have an outgoing name binding edge to the variable declaration, and control flow will not be preserved, but def-use edges should be preserved. Therefore, this refactoring proceeds according to the rule $N\overset{\rightarrow}{\supseteq}D$.

7.4 Move Expression Into Assignment [Prerequisite] — Requires: none

This refactoring moves an expression from its original context into an assignment statement and then replaces the original expression with a use of the assigned variable.

Input.

1. An ⟨*expression*⟩ E occuring in the context of a ⟨*function*⟩.
2. An assignment statement A of the form V `= 0` for a Scalar Variable V.

Preconditions. Let S denote the least ⟨*statement*⟩ containing E.

1. S must exist in the immediate context of a ⟨*statement_list*⟩. Let L denote this ⟨*statement_list*⟩.
2. There must be no references to V except for the reference in A.
3. The assignment statement A must exist in the immediate context of L.
4. For each ⟨*statement*⟩ S' occuring after A but before S in L...
   (a) Fail if S' assigns a Variable occuring in E.

Transformation.

1. ♦ In the assignment statement, replace the RHS expression `0` with E, removing E from its current context.
2. ♦ In E's original context, insert

$$\langle expression \rangle \leftarrow V$$

Notes. Preconditions 2 and 4 can be eliminated in a differential refactoring engine since they are effectively preserving def-use edges. (For that matter, they are unnecessarily strong.) The affected forest should include both of the replaced expressions (in the assignment statement and in the original context). Then, there will be one internal def-use edge introduced (from the new variable to the assignment statement), but otherwise no def-use edges should be introduced, and all name bindings should likewise be preserved. Therefore, this refactoring should proceed according to the rule $ND^{\circlearrowleft}_{\supseteq}$.

7.5 Extract Local Variable — Requires: (prerequisites)

Extract Local Variable removes an expression or subexpression from a statement, assigns it to a local variable, and replaces the original expression with a reference to that local variable.

Although the refactoring ensures that du-chains for local variables are preserved, it is the user's responsibility to ensure that the extracted expression is side effect-free or that the program will exhibit the correct behavior if it is not.

Input.

1. An ⟨*expression*⟩ E in a ⟨*function*⟩ F.

2. A new Name N for the local variable that will be created.

Preconditions.
1. E must have scalar type.
2. F must *not* declare or reference a scalar named N.

Transformation.
1. Add an Unreferenced Local Variable Declaration for N [Prerequisite].

   Let S denote the least $\langle$*statement*$\rangle$ in which E occurs.
2. ♦ If S is the $\langle$*statement*$\rangle$ providing the body of a for-statement, if-statement, or while-statement, Enclose S in a Block [Prerequisite], and, in the remaining steps, assume that S exists in this new context.

   (Note that, by construction, S must now exist in the immediate context of a $\langle$statement_list$\rangle$.)
3. ♦ Insert an Assignment to the Unreferenced Local Variable N [Prerequisite] immediately before S.
4. ♦ Move E Into the Assignment statement inserted in the previous step [Prerequisite].

Notes. —

7.6 Add Empty Function

Requires: none

This refactoring adds a new $\langle$function$\rangle$ to a $\langle$program$\rangle$. The $\langle$function$\rangle$ initially has an empty body. The refactoring fails if a $\langle$function$\rangle$ with the same name already exists.

Input.
1. A $\langle$*program*$\rangle$ P.
2. A new name (LETTER) N for the function.

Preconditions. FAIL if any $\langle$*input_item*$\rangle$ in P is a $\langle$*function*$\rangle$ whose name (LETTER) matches N.

Transformation. ♦ Append to P

$\langle$*input_item*$\rangle$ $\leftarrow$ `define` N`() {`↲
`}`↲

Notes. In a differential refactoring engine, the precondition can be eliminated: Depending on the implementation, introducing a function with the same name as an existing function will either result in a compilability error or the introduction of new def-use or name binding edges. Therefore, this refactoring should preserve the program graph in its entirety.

7.7 Populate Unreferenced Function

Requires: [Cv]

This refactoring copies statements from one function into another, replacing local variables with function arguments and returning the value of a variable if necessary. The refactoring fails if more than one value must be returned.

Input.
1. A sequence $S := S_1, S_2, \ldots, S_n$ of consecutive $\langle$*statement*$\rangle$s from a $\langle$*statement_list*$\rangle$ in the immediate context of a $\langle$*function*$\rangle$.

Preconditions.
1. There must not be a `return` statement in the context of S.
2. (Checked during transformation)

Transformation.
1. Classify local variables in S [Cv] to obtain the set X and the functions *isParam* and *isReturn*.
2. ◇ FAIL if $|isReturn(X)| > 1$.
3. *(Construct an $\langle$opt_auto_define_list$\rangle$ A and an $\langle$opt_parameter_list$\rangle$ P.)*

(a) Initially, let A and P be empty.
(b) For each V in X...
  i. If *isParam*(V) = TRUE, append V (and a comma, if necessary) to P.
  ii. If *isParam*(V) = FALSE...
    A. If A is empty, define A to be

$$\langle \textit{opt\_auto\_define\_list} \rangle \quad \leftarrow \quad \texttt{auto}\ V$$

    B. Otherwise, append V (and a comma, if necessary) to A's ⟨*define-list*⟩.

4. *(Construct a return statement R.)*
   (a) Initially, let R be empty.
   (b) If *isReturn*(X) = $\{V\}$ for some V...
     i. Define R to be

$$\langle \textit{statement} \rangle \quad \leftarrow \quad \texttt{return}\ N(V) ↲$$

5. ♦ Replace F with

```
⟨function⟩  ←  define N(P) { ↲
                   A ↲
                   S1 ↲
                   S2 ↲
                   ...
                   Sn ↲
                   R ↲
               } ↲
```

where A and R are omitted if they are empty.

Notes. In a differential refactoring engine, both preconditions can be eliminated, as long as this refactoring is being used only in the Extract Function composite: This is because a failure to meet these preconditions will cause Replace Statement Sequence to fail. If the statement sequence includes a RETURN statement, this control flow will be lost when the statement sequence is replaced. Similarly, if more than one value needs to be returned, a def-use chain will be lost when the statement sequence is replaced.

7.8 Replace Statement Sequence S — Requires: [Cv]

This refactoring replaces a sequence of statements with an equivalent function call. This refactoring is not intended to be used except as part of Extract Function.

Input.
1. A sequence $S := S_1, S_2, \ldots, S_n$ of consecutive ⟨*statement*⟩s from a ⟨*statement_list*⟩ in the immediate context of a ⟨*function*⟩.
2. A new Name N.

Preconditions. (none)

Transformation.
1. Classify local variables in S [Cv] to obtain the set X and the functions *isParam* and *isReturn*.
2. *(Construct an ⟨opt_parameter_list⟩ P.)*
   (a) Initially, let P be empty.
   (b) For each V in X...
     i. If *isParam*(V) = TRUE, append V (and a comma, if necessary) to P.
3. ♦ If *isReturn*(X) = $\{V\}$ for some V, replace S with

$$\langle statement\_list \rangle \quad \leftarrow \quad V = N(P) \hookleftarrow$$

4. ♦ Otherwise, replace S with

$$\langle statement\_list \rangle \quad \leftarrow \quad N(P) \hookleftarrow$$

Notes. In a differential refactoring engine, this replacement is expected to preserve incoming and outgoing control flow and du-chains if it is to preserve behavior. Clearly, the set of name bindings will change. Therefore, this refactoring proceeds according to the rule $C^{\circlearrowleft}_{\subseteq}D^{\circlearrowleft}_{\subseteq}$.

7.9 Extract Function

Requires: (prerequisites)

Extract Function creates a new method from a sequence of statements and replaces the original statements with a call to that method.

Input.

1. A sequence $S := S_1, S_2, \ldots, S_n$ of consecutive ⟨*statement*⟩s from a ⟨*statement_list*⟩ in the immediate context of a ⟨*function*⟩.
2. A new Name N.

Preconditions. (none)

Transformation.

1. ♦ Add an empty function named N [Prerequisite]. Call this function F.
2. ♦ Populate F according to S [Prerequisite].
3. ♦ Replace S with a call to F [Prerequisite].

Notes. —

Part III

PHP

8 Definitions

Class Declaration. An ⟨*unticked-class-declaration-statement*⟩.

Class Name. The T_STRING in an ⟨*unticked-class-declaration-statement*⟩.

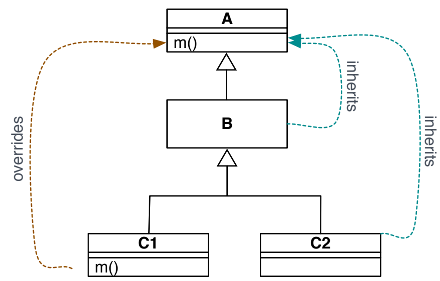
Method Declaration. A ⟨*class-statement*⟩ matching

⟨*method-modifiers*⟩ ⟨*function*⟩ ⟨*is-reference*⟩ T_STRING (⟨*parameter-list*⟩) ⟨*method-body*⟩

Method Name. The T_STRING in a Method Declaration.

9 Preconditions

9.1 Precondition [II]: Introducing M into Class C' must not introduce unexpected inheritance



In a situation such as the one illustrated above, method m cannot be pulled up from $C1$ into B because this would cause $C2$ to inherit the pulled up method. This precondition prevents situations like this, where a class would inherit the "wrong" override of a method.

Input. A Method Declaration M in a Class Declaration C with a direct superclass C'.

Procedure.
1. If M does not override a concrete superclass method, PASS.
   Otherwise, suppose M overrides M', which is defined in class P.
2. For each (direct or indirect) subclass D of P...
   (a) FAIL if all of the following hold:
      i. D inherits M' from P.
      ii. D is a (direct or indirect) subclass of C'.
      iii. $D \neq C$.
3. PASS.

10 Refactorings

10.1 Copy Up Method [Prerequisite]

Requires: [II]

Copy Up Method copies a method from one class into its immediate superclass.

Input. A Method Declaration M in the context of a Class Declaration C.

Preconditions.

1. There must be an ⟨*extends-from*⟩ node in the immediate context of C, and its ⟨*fully-qualified-class-name*⟩ must (uniquely) identify a Class Declaration C' in the same file as C.
2. M's ⟨*method-modifiers*⟩ must not contain T_ABSTRACT or T_STATIC.
3. C' must not contain a Method Declaration with the same name as M.
4. If there are any refrences to M that are not recursive references contained in M, then M must not have private visibility.
5. M must not contain any references to `self` or `__CLASS__`.
6. M must not contain any references to private members of C.
7. Moving M to C' must not introduce unexpected inheritance [II].
8. If M overrides a concrete method, and C' is not an abstract class, WARN the user that M will replace the overridden method in C', possibly changing the behavior of objects of that type.
9. If C' defines or inherits `__call`, and M does not override a superclass method, WARN the user: the program's behavior may change, since M will be invoked instead of `__call` for objects of type C'.

Transformation. ♦ Move the ⟨*class-statement*⟩ containing M from C's ⟨*class-statement-list*⟩ into C''s ⟨*class-statement-list*⟩, replacing all references to `parent` with `self`.

Notes. We require the superclass to be in the same file as C in order to avoid dealing with `include` directives.

In a differential refactoring engine, precondition 3 will be caught by a compilability check. Preconditions 4–6 are simply preserving name bindings. A program that failed precondition 7 would introduce an incoming inheritance edge. If a program failed precondition 8, an outgoing inheritance edge from C' would vanish. Preconditions 1 and 2 cannot be eliminated because they perform input validation; precondition 9 checks for behavior that is not modeled by a program graph. This refactoring proceeds with preservation rule $NO^{\circlearrowleft}_{\supseteq}I$.

10.2 Pull Up Method

Requires: (prerequisites)

Pull Up Method moves a method from one class into its immediate superclass.

Input. A Method Declaration M in the context of a Class Declaration C.

Preconditions. None.

Transformation.

1. ♦ Copy Up M [Prerequisite].
2. ♦ Delete the ⟨*class-statement*⟩ containing M from C's ⟨*class-statement-list*⟩

Notes. All of the preconditions for this refactoring are handled by Copy Up Method. The delete operation proceeds with preservation rule $NO^{\circlearrowleft}_{\subseteq}I^{\leftarrow}_{\supseteq}$.