Changing Software Correctly

An inquiry into
verified refactorings with
program specifications

Fabian Bannwart

2006
Software Component Technology Group
ETH Zürich

Changing Software Correctly

Institution	ETH Zürich
Department	D-INFK
Author	Fabian Bannwart
Date	February 21, 2006

Modifications of code that do not alter the functioning of object-oriented software are called refactorings. Refactoring *tools*, not programmers, should guarantee correctness to be most useful.

When do refactorings retain equivalence? To tackle this question, I conceive a semantic model of refactorings. It aims at making it simple to prove refactorings correct, i.e., to prove that equivalence is retained. It is parameterized over an operational semantics. The crucial point is that the correspondence between original and refactored program need not be defined for all states and all program points. The formalism is crafted so as to offer a solution to a second problem: How can refactorings be applied *in practice* without giving up equivalence *guarantees*? The idea is this: The proof procedure for refactorings yields minimal obligations that must be satisfied by the transformed program. These conditions can be formulated as pre- and postconditions of statements in the code – they are local. Every program you want to apply the refactoring to needs to satisfy those conditions. It is not practical to check them by hand. I suggest that a refactoring tool does it for you: It will check simple conditions before applying the refactoring as "preconditions" and it will add specification and instrumentation to the transformed program for all those properties that cannot be readily checked or sensibly approximated – these are called "postconditions". It will also add *all* conditions as specifications that encode architectural constrains. With every refactoring, the programmer thus codifies part of the tacit knowledge that is needed to know that the respective refactoring is applicable for the program. The idea to extract specifications from interactions between program and programmer can be applied to any change with defined semantics.

A refactoring catalogue that is included does not mainly serve as an illustration of theoretical concepts. It shows that most refactorings can be reduced to individual code changes that are easy to verify using the theoretical framework and then be composed to the conventional refactorings as we know them. I.e., refactorings in this text hardly ever affect an unbounded number of code locations. The exception are data-refactorings. They are covered separately.

Contents

1. Introduction 5

2. Refactoring: A Motivating Example For Simple Refactorings 25

3. Formalizing Equivalence For Refactorings 39

4. Simple Refactorings Proved Correct 75

5. Refactoring Access Paths: Moving Data Between Objects 117

6. Implementation Example 153

7. Conclusion 175

A. Code For Program Representation 179

B. Notations 181

C. About the Operational Semantics 183

1. Introduction

Definition

Refactorings are parameterized equivalence transformations. The intent of these transformations is to reduce the code's resistance to new features you want to add, i.e., equivalence transformations to improve the design of existing code.

For refactorings, equivalence is confined to the equality of externally observable behavior. Properties like the internal machine state or power consumption or execution time are not taken into account. Refactorings retain the value of data *processing* applications.

Importance

Refactorings are practically important because they reduce the resistance to change of existing code and can therefore provide tangible benefits. They are important as part of the software development process and are thus tool supported.

State of the art

Existing refactoring tools – such as the one that comes with Eclipse – are fairly simple. They take the original program and transform it to a program that may not even compile. For the *correctness* of the transformations, i.e., *whether or not equivalence has been retained*, the programmer has to rely on unit tests. If the unit tests still pass after the transformation, the refactored program is considered equivalent to the original program. This of course only works if the unit test coverage is complete. This can hardly be expected and the existing approach is thus insufficient.

Mission and vision

I envision a tool that is applicable to the broadest set of programs possible, transforms them and guarantees that the result is always and for all possible inputs correct. Such a tool does not exist yet and neither do the preliminaries for its implementation. Establishing preliminaries and illustrating their application is what this text is about.

It does not provide a working implementation even though chapter 6 does contain some implementation code for a simple refactoring.

Refactorings have conditions

Correctness is not only a question of the right *transformation.* It is also about the conditions the program must satisfy so as to make the refactoring correct. For example rendering a virtual method `C::f` static is only a valid refactoring if (i) the transformation is "proper" and (ii) a static `C::f` does not exist with conflicting signature and (iii) there are no call-sites of `C::f` that resolve to a different method implementation. The example makes clear that only allowing refactorings that are correct *for all programs* is not an adequate solution. "Proper" means two things: the refactoring must produce syntactically and semantically correct results for programs that satisfy conditions (ii) and (iii).

These conditions are what this thesis is focused on: The goal of this thesis is to prove – for some reasonable equivalence criteria formulated as pre- and postconditions of program statements – the conditions of validity for refactorings. There is no universal method to date that would allow to do this for general sequential programs that support difficult features like pointers and exceptions. The system should also be flexible enough to handle other difficult aspects that are conventionally not considered important like finalization and garbage collection [21].

Part of this text deals with definining such a method. The viability of the techniques presented here is tested against some representative refactorings, most of them from Fowler [16], the standard text on refactorings. If a refactoring is not defined in the text, a description can be found in [16] unless stated otherwise. Names of refactorings are always written in quotes.

The central idea that renders formalized refactorings practical is to emphasize local equivalence criteria instead of global ones. Local criteria are criteria that depend on the pre- and/or the poststate of a – possibly complex – statement. Criteria that reason about all intermediary states are not local. It is easy to state global criteria ("the program has to return the same results after transformation", see [37]) but it is very difficult to check them.

Modularity

It is more difficult to state criteria that are locally checkable but still weak enough to yield applicable transformation criteria. Local verifiability is not the same as modularity, but local criteria can be translated to modular criteria by asking which conditions have to be maintained when putting together various software components. These are the proof obligations for the separate system parts if refactorings have to be applied. This is as far as you can get to modular refactorings because refactorings that change abstraction[1] interfaces always require changes to the instantiations of these abstractions, which may be dispersed in the program. An important design goal for a complete set of refactorings and its implementation in a refactoring tool is to offer as few refactorings as possible that have dispersed effects.

Whay are refactorings necessary that are unconditionally correct?

The introductory explanations already provide some legitimation for formally verified refactorings that always yield correct results. Relying on unit tests has certain advan-

[1] *Abstractions* are results of *abstracting*. *Abstracting* in general is the act of separating a thing from its direct associations. Programs and procedures are abstractions of operations over input data. Classes are abstractions of data and operations. Templates are abstractions of types. Abstractions are instantiated and therefore have an *abstraction interface* that defines how exactly the instantiation is to be done. The *abstraction principle* states that any syntactically meaningful unit may be abstracted in a programming language, i.e., that abstraction is completely supported [45]. The problem with abstractions is that refactoring abstraction interfaces affects all uses of an abstraction even though such changes are often driven by the needs of a specific instantiation. Translation between abstraction interfaces becomes important in that case as exemplified in chapter 4 where changes to abstraction interfaces are always treated as local refactorings. It is not easily possible to translate between abstraction interfaces of storage like fields, which I call data abstractions. There should always be only one data representation (cf. chapter 5).

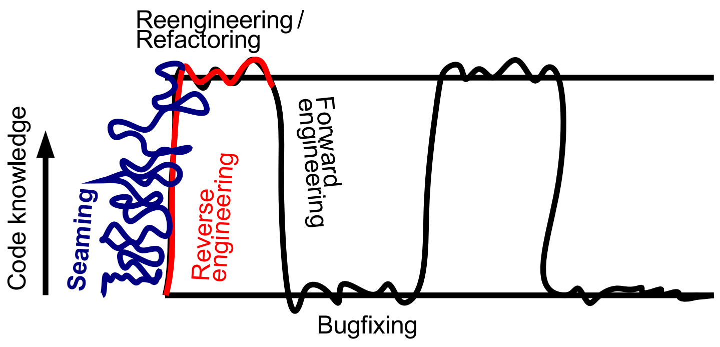


Figure 1.1.: The software reengineering lifecycle (adopted from [13])

tages however that cannot be easily discarded. Unit tests (i) provide a kind of documentation, (ii) guarantee loose dependencies[2] and (iii) are never overly restrictive because all the programmer's implicit assumptions are encoded in the unit tests. There is a more foundational reason for having a provably correct refactoring and tool support in addition to unit tests that has to do with the software development lifecycle. It is described in the next section 1.1.

1.1. Beyond unit tests: Verified refactorings for the software reengineering lifecycle

Knowing that automated refactorings do not change the behavior of existing code does not help much if we do not know whether the behavior is correct in the first place. Correctness in the context of software maintenance is thought to be guaranteed by unit tests that can be executed quickly after every change.

Unit tests

Example 1.1. If a unit test runs for more than 30ms on today machines, it is probably too slow.[3] Here is why: In a system of usual size with 3000 classes and 20 unit tests for each class, running all the tests takes more than half an hour. Even if you can make sure you're using only a subset to work with, it will still take a few minutes.[4] In any case, this will keep you from running the tests.

All is fine if you do have unit tests that allow you to perform the changes safely *and* quickly. Before you can perform changes, tests have to be put in place. Even if there

[2]Unit tests by definition only affect one "unit", either a single class, or a restricted group of classes that are tightly coupled. The fact that they can be tested in isolation means they are only loosely dependent on the rest of the system.

[3]I would suspect either the unit or the test is flawed in this case. The unit might be too big, or the test may not be focused enough.

[4]I wonder why determining (a conservative superset of) unit tests that are affected by a change is not a standard feature in today's IDEs. Erich Gamma [17] claims that it would involve much more than a crude dependency analysis.

Seaming code without knowing it.[5]

are tests, this can be problematic. For code that needs reengineering, you often have to introduce such tests or update and rectify older "tests" that do not faithfully reflect the state of the code. You may not understand the test hooks the code offers. You have to change code in most cases to introduce and/or modify the tests. How can you (unit) test a class that expects an opaque "database connection"[6] or an opaque GUI toolkit? Setting up a database or a database simulator may be just about feasible for tests that should allow you to quickly check a refactoring even if they make the development process closely dependent on the environment configuration. Automating the interactions with a GUI however is a daunting task. The same is true for report generation and printing, file handling and interfaces to other substantial subsystems of the application. Michael Feathers [15] calls this the legacy code dilemma: Before being able to introduce unit tests that would allow safe and rapid modifications, modifications to the code have to be made to render the code testable. This is called *seaming* and mostly consists of breaking external dependencies. Tests that use database connections, GUIs, files, etc. are not unit tests.

Definition "seaming"

It is a fallacy to believe that the maintenance of realistic, yet "properly engineered" code equiped with unit tests is not subject to this conflict – although on a microscopic scale, which renders the process much less painful. Software maintenance (and development) is usually a cyclic exercise that involves (i) quick introduction of new features or change of existing features (forward engineering), which increase the complexity of the system. (ii) Test and possibly use of the features. (iii) A reverse engineering phase to discover what the code was about. (iv) A reengineering phase that involves realigning the code structure, writing tests for the features and updating old tests to make the code ready for the next generation of changed requirements. This is shown in figure 1.1

The difference between legacy code and "properly maintained" code is the mere fact that properly maintained code is subject to this cycle regularly and recurrently while legacy code is barely forward engineered. Laudible practices like XP [4], TDD [30] and "continuous refactoring" demand an extreme form of the software reengineering lifecycle,[7] they do not eliminate it. They aim at reordering it, blurring the distinction between its phases and render them so small as to make them indistinguishable – it is certainly easier to figure out what three lines of code added a few seconds ago are doing than to crack the meaning of 30000 lines of code that is six months old. The primary benefit of this culture of constant change is not mainly to keep the code up to date with formal requirements – this could be done in chunks – but to *safeguard the tacit knowledge of existing code.*[8]

[5] *Argument summary*: Modifications have to be made to make code testable. This is not possible by hand because making changes by hand requires knowledge about the code. Refactoring tools can help.

[6] In Java, the database connection could be a class instead of an interface. In C, consider `fprintf` with a `FILE*` handle.

[7] The term is coined in [13].

[8] It is true that other programming paradigm variants like modular programming do have limitations

Unfortunately, tacit knowledge tends to get lost in practice, turning the program into legacy. When modifications are made, this knowledge has to be rediscovered. This is best done by writing and updating unit tests, which has the desirable effect that the software becomes even more maintainable. Before effective unit tests can be introduced, dependencies have to be weakened. The code has to be made structure shy.[9] This is a refactoring itself – in fact, it is the greatest challenge of updating legacy code. Feathers [15] writes that the "trick is to do these initial refactorings very conservatively". This is a laudible advice, but it is better done with formally verified refactorings. They can help refactor the code safely without having extensive tests.

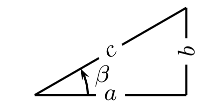
The general idea, briefly mentioned

The refactoring tool can add checks that guarantee correct execution whenever the code is exercised – at least after you start writing tests. These assertions help debug the newly written tests as they enforce backward equivalence. The checks can reveal the expected "fixture" for the functioning of a class. The next sections describe this process in detail.

1.2. Automated refactoring

Pre- and postconditions

This text focuses on local conditions for refactorings. Local criteria are also practically useful because they are easy (even though they may be expensive) to check dynamically by the runtime system after the transformation. This mandates that the local conditions be specified as "post"-conditions checked in the refactored program as opposed to "pre"-conditions checked in the original program (suggested in [37] and normally implemented in refactoring browsers, see figure 1.2). A refactoring tool could instrument the object code with such tests. It should – in general – be possible to translate between "pre"- and "post"-conditions. If this is not or not conveniently possible, a conservative approximation must be found.



Example 1.2. Consider a program doing calculations on a right triangle with hypotenuse c and catheti (legs) a and b. The aim of the application is to print the length of c given a and b. I show only the method that calculates c:

that are probably more severe than those of object orientation. It should not be contested however that tacit knowledge of even beautifully designed applications is urgently needed particularly in object-oriented systems because object-oriented programs tend to conceal the runtime data structures and its interaction with inheritance and dynamic dispatch. Moreover, most OO languages force the programmer to split shared responsibilities into separate classes and/or compilation units breaking modularity and encapsulation. This makes relevant code hard to find.

[9] This makes the case for *explicit* constructs in programming systems that allow *implicit* disassociation between different code parts. Such implicit means do not exist to date (unless we accept completely dynamic languages that cannot provide safety guarantees). In Java-like languages, disassociation *can* be achieved with interfaces and dynamic dispatch. In C++, it can also be done with templates. In Haskell, it is done with type classes. SML has the most support for dissassociation, which it calls signatures and functors. The possibility to minimize "hard links" between program parts is often cited as a condition for reusable components (see for instance [19, 36])

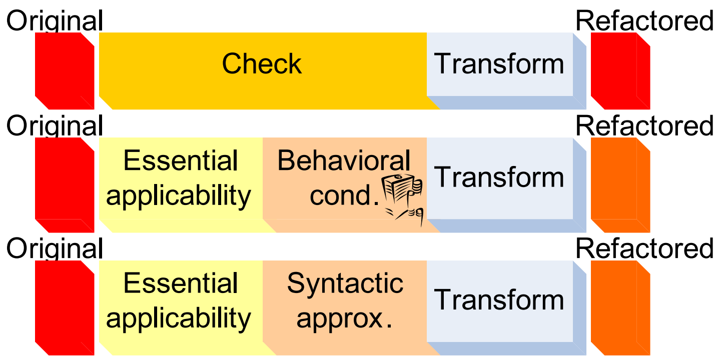


Figure 1.2.: Automated refactoring: the conventional perspectives

```
public class RightTriangle {
    double a, b; // fields

    public double get_c() {
        double c = a/Math.cos(Math.atan(b/a));
        return c;
    }

    ...
}
```

There is also a JUnit test for the the program:

```
public class RightTriangleTest extends TestCase {
    public void testGet_c(){
        RightTriangle t = new RightTriangle(1,2);
        double exp_c = 2.2;
        Assert.assertEquals(exp_c, t.get_c(), 0.1);
    }
}
```

At some point, the developer may discover the Pythagorean theorem $c^2 = a^2 + b^2$. He wants to simplify the calculations so he asks the refactoring tool to "Replace Expression" `a/Math.cos(Math.atan(b/a))` by `Math.sqrt(a*a+b*b)`. The following bulleted list describes how each of the known and some hypothetical approaches to automated refactoring would handle the problem and how the approach of this thesis solves this task.

- If the tool is a conventional refactoring aid, the, tool will (hopefully) check whether `Math.sqrt(a*a+b*b)` is indeed a valid expression. This corresponds to the first line in figure 1.2. The transformed `get_c` method will look like this:

```
public double get_c() {
    double c = Math.sqrt(a*a+b*b);
    return c;
}
```

- If the tool is a bit more sophisticated, corresponding to the second possibility in figure 1.2, the tool will first check the syntactic validity of `Math.sqrt(a*a+b*b)`, and then it will try to prove some sort of equivalence between the code that calculates `c` in the original and the transformed code, i.e., equivalence between `a/Math.cos(Math.atan(b/a))` and `Math.sqrt(a*a+b*b)`. That's not exactly trivial, but that's what formal verification is about! The code will look just like above.

- The tool that corresponds to the third line in figure 1.2 will try to find a syntactic approximation to the required equality. It may for instance expand the expressions `a/Math.cos(Math.atan(b/a))` and `Math.sqrt(a*a+b*b)` to verify whether the two code sequences are identical. It could also do some abstract interpretation of the code. If the result of the interpretations are the same, the refactoring is applied and the code is being transformed.

- Roberts [39] proposed a new approach that is introduced here to evoke a deja-vu in sections 1.2 and 1.4.2.[10] Before doing the transformation, all unit tests are run. The JVM is instrumented, so each time the program counter reaches the line the calculation of `c` starts, the program forks. One instance of the application uses the conventional algorithm `a/Math.cos(Math.atan(b/a))`, the other instance uses the new one `Math.sqrt(a*a+b*b)`. If the resulting state spaces of both instances are always compatible, the two expressions are considered equivalent and the refactoring is applied as before.

- What I briefly proposed at the end of section 1.1 and elaborated on at the beginning this section is this: Before the refactoring is performed, the tool checks the precondition, making sure that the new expression will be valid in the respective context. It will then generate a predicate that checks the equivalence the two old and the new expressions. The predicate may contain code, so it should make sure that adding the predicates does not alter the semantics of the transformed programs. This is all done before the actual transformation because this cannot be determined effectively at runtime. The tool will then apply the refactoring and add the predicate as a specification to the program at the appropriate location. For the present case, the specification could be a simple Java assert statement:

[10]Roberts actually investigated simple predicates that could easily be evaluated by the runtime system itself. I describe here how his idea can be extended to a more general setting.

```
public double get_c() {
    double c = Math.sqrt(a*a+b*b);
    assert Help.approxEq(c,
        a/Math.cos(Math.atan(b/a)));
    return c;
}
```

No unit tests are run to exercise the new code. Instead, the assert statement remains in the code. The advantage is obvious: The code will not change its behavior without being noticed even if there are no or only incomplete unit-tests like the one above because the test is performed every time the application is run (with the right flags). Even if the assertion can be verified or reduced by a static checker, the assertion is useful to document certain assumptions. In this example for instance it is required that the value of `a` is positive.[11] The new code in the presence of a checker could then be

```
public double get_c() {
    assert a > 0;
    double c = Math.sqrt(a*a+b*b);
    return c;
}
```

The rest of this section describes the vision of refactorings with specifications and discusses various considerations.

Not everything is a postcondition

Preconditions are not made obsolete by postconditions. They are still necessary to define when a refactoring is appropriately applied. You cannot apply “Replace Typecode with Subclass” if you do not use a typecode. Moreover, an automated tool may well assume some standard structure it can apply its algorithms to. These preconditions have a different status from the postconditions that ensure the programs preserve equivalence. Only *essential applicability conditions* have to be formulated as preconditions that are tied to the structure and the concept of the refactoring, not its operational properties.

Criteria do not suggest themselves as either pre- and postconditions in all cases. Some people may even find it debatable whether non-aliasing properties belong to the pre- or the postconditions. Instead of lengthy deliberation, a pragmatic solution has to be adopted for the sake of practical applicability: It must be possible to sensibly approximate the semantic preconditions syntactically. And if this is possible, it should actually be done, favoring pragmatism over unrestricted generality. This is why I reserve postconditions for a few difficult refactorings.

[11] This is an illustration that the original program is not always the best normative measure.

Conditions should only be checked in the running program if it is not reasonably possible to check them statically. They can be added to the program as specifications, which might get translated to some static verification and dynamic checks for whatever cannot be verified statically. These conditions can also be implemented as simple dynamic checks. Or, alternatively, these conditions can be verified online: code is monitored while running and transformed afterwards if the conditions are met. Even though these conditions are not checked *after* the transformation, they fit well with the notion of postconditions as conditions that are *not* checked *before* considering the transformation.

Why minimize postconditions?

What is the disadvantage of postconditions? Postconditions are inserted into the program as specifications, conventional instrumentation or they can be added to unit tests. Every semantic change (by the programmer!) to the program functionality may invalidate these generated parts of the program. He may not be familiar with the annotation or additions that may have been injected under the hood and yet the refactoring tool cannot help determine whether the specifications are still valid after the programmer makes changes to the program. The programmer has to remove the instrumentation manually. An important consequence is therefore that the postconditions be visible to the programmer, so as to communicate, with every refactoring step, which assumptions about the code are implied by the application of a certain refactoring.[12]

The fact that the programmer has to learn new annotations is a disadvantage. It renders the programming environment more complex. Still, postconditions are not be as bad as they may seem because they mostly involve intuitive properties e.g. about aliasing in the object graph. These are easy to communicate and appropriate for translation into aliasing control type systems for instance. Some people [22, 23] believe, including myself, that alias control and other sorts of constraints – as the ones inserted into the application by a clever refactoring browser – benefit the quality (not only the safety) of programs.

It is a central concept in this text (and a novel idea) to inject refactoring postconditions as specifications in the program and expose them to the programmers. With every transformation, the programmer codifies tacit knowledge about the program's behavior – tacit knowledge that may be proved wrong.[13]

[12]Again, tests cannot be assumed, so adding, testing, removing [39] is not an viable option for all programs.

[13]Previously, postconditions were viewed as a mere tool to avoid expensive verification and inaccurate static analyses.

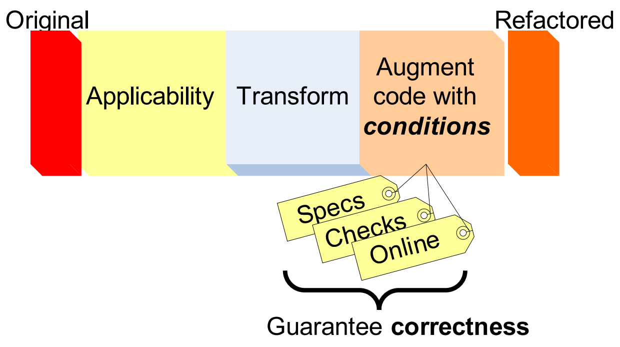


Figure 1.3.: Automated refactoring employing postconditions

1.3. Primitive and composite refactorings

The refactorings presented in Fowler [16] are hierarchically organized ("composite" refactorings, see [37]). There are atomic ("primitive") refactorings[14] that are not composed from, and therefore independent of other refactorings. This can be seen as the definition of primitive/composite refactorings: "A primitive refactoring is a refactoring that *is* not decomposed into simpler refactorings". ([10], emphasis added) The ideal is that the primitive refactorings form a core to which all refactoring transformations can be reduced. It has to be complete[15] and convenient for assemblage, but apart from that, no other criteria are inevitable. This suggests that the modest definition is quite sufficient.

Most refactorings are composite. They depend on other, simpler refactorings for parts of their transformation. This is by no means limited to the refactorings in [16, chap. 12]. If possible, I want to avoid composite refactorings and focus on atomic ones because composition is simple, but correctness for atomic refactorings has to be tackled directly.

There have been attempts to show equivalence of refactorings for composite refactorings ([9]). There is just one way of composing refactorings: applying them after each other. In the process I envision, each refactoring in a chain of applied refactorings adds assertions to the source program. As long as these assertions use the abstractions of the source program only (i.e., they can be expressed as ordinary program expression, they are transformed together with the following transformations) I.e., as long as postconditions can be evaluated inside the program, unlimited composition of refactorings is possible.

As I said in the text, I am confining myself to primitive refactorings. What are primitive refactorings? A refactoring is certainly primitive at least in cases when it *cannot* be decomposed to other primitive refactorings. This self-referential definition does not

[14] or refactorings that can be made atomic.
[15] Every equivalent program has to be reachable using only primitive refactorings.

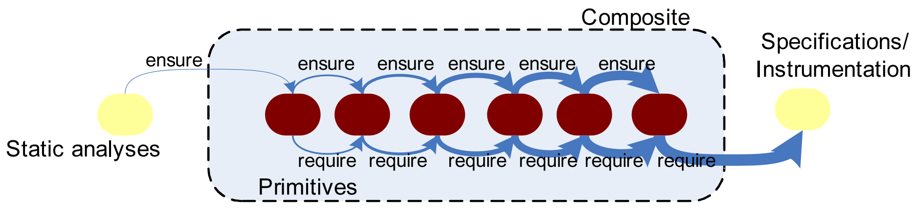


Figure 1.4.: Composition of refactorings

reveal a unique set of primitive refactorings. Likewise, not every primitive transformation is a refactoring even if equivalence is retained in some cases. It should actively provide for equivalence. Custom and conventional wisdom have to be respected in both cases.

1.4. Comparison to related efforts

The related work discussed here is only a small (more probably *tiny*) fraction of the vast literature on refactoring. It merely discusses those efforts that are geared towards reasoning about the correctness and conditions of refactorings and not the refactorings qua unconstrained program-transformations as covered for instance in Lämmel et al. [8, 26]. Texts that focus on the design rationale of refactorings such as the book by Kerievsky [25] are not considered either.

I summarize each of the various efforts in chronological order and compare them to the approach chosen for this research. The section ends with a tabular overview of related work.

1.4.1. Opdyke [37]

Contribution

It is not quite clear what the single most important academic achievement of Wiliam Opdyke in his doctoral thesis [37] was. Maybe that's the reason why his contribution is better known as "Opdyke's thesis" than anything else. Fact is that Opdyke's thesis is the first readable account on refactorings and the first serious attempt to examine them thoroughly. In fact, he even invented the term "refactoring"! The language he uses is a subset of C++.

Opdyke's seven properties for behavior preservation

The part most relevant to this research is [37, chapter 4]. It discusses conditions for refactorings to maintain program behavior. Opdyke lists seven conditions, six of which are trivial and concern syntactic properties that must hold after the refactoring in his system: unique superclass and acyclic subclass relation, distinct class names, distinct member names within a class, distinct member variables across subclasses, compatible signatures of overriding methods and type-safe assignments. There are certainly more than these six properties a compiler has to check for well-formedness, but these are the ones he

Q: What is equivalence?
A: Two programs are equivalent if their output is the same for the same input.

lists.[16] What Opdyke lists as *number 7* is quite interesting indeed, and not at all trivial: *“Semantically equivalent references and operations”*.

Opdyke defines semantic equivalence as “the resulting set of output values must be the same”. It may seem trivial but I found that it is quite insightful and significant compared to what other people have come up with later when trying to formalize behavior preservation. Other definitions of equivalence are discussed later in the context of their respective proponents. I list them briefly: Two programs are equivalent if (variant i) the internal state is the same at the end given the same initial state [11]; if (variant ii) some lexical properties of the program structure are retained [29]; if (variant iii) they are considered equivalent “somehow” [39, 9].

There are some denotational problems with Opdyke’s definition. What does “output” mean? Does timing[17] count as output? What about network traffic latency, etc. It has to be made more concrete for formal purposes, but it still seems the most sensible definition as it comes closest to the intuitive understanding of what “externally visible behavior” encompasses (certainly not true for variants i & ii) and it is at the same time not fatalisticly unapproachable (unlike variant iii). Opdyke’s definition is the one to be used in this text.

Opdyke introduces quite a few important concepts that had not been stated as clearly before. He recognized that common refactorings “are only behavior preserving under certain *preconditions*” (emphasis added) He did not only invent refactoring preconditions, he also formalized these conditions for the refactorings he examined. The functions he used closely resemble the ones that were later used in the famous description of the Smalltalk Refactoring Browser [39] I am going to discuss in the next section 1.4.2.

Opdyke informally argues for the correctness of the transformations. An example from *delete_function_argument*, [37, p. 46]: “[. . .] Expressions passed to Arg (in calls to its containing function) have no side effects. Therefore, program property seven (semantically equivalent references and operations) is preserved. The other program properties are trivially preserved. [. . .]”

He does not have a formal model what correctness encompasses, which by no means disqualifies his work but renders it more flexible and useful than later approaches. He does however have a formal model of the program representation. He also introduces a concept to capture the reach of a refactoring outside of which the program and its semantics stay the same. It roughly corresponds to what I call “locality” in later chapters.

[16]Syntactic well-formedness of refactoring results is not discussed explicitly in this text. It is implied that the preconditions guarantee it.
[17]Consider calls to `sleep(1000000)` in C/C++!

Summary: How I complement Opdyke's work Opdyke's work has a different focus. It established fundamental ideas but it adds little that is practically useful for the goals of this thesis: He does not have a formal notion of equivalence. He consequently does not have a proof method that directly yields minimal conditions for the correctness of refactorings. He didn't consider of postconditions or specifications. He does not cover the decomposition and he does not explicitly recognize the value of confining the locality of a refactoring.

1.4.2. Roberts' Refactoring Browser [39]

Roberts' thesis – submitted in 1999, seven years after Opdyke's seminal work – provides more practical insights into the refactoring process. In particular, he adds the concepts of postconditions to avoid having to analyze the program statically, which – he consistently argues – is too inaccurate to be useful.[18] He proposes a dynamic approach to refactoring and emphasizes unit tests as a program specification. The postconditions are evaluated while the unit-tests are executed. He coins the term "online refactoring" for a variant where the program is transformed and monitored while it is running.

In Roberts' formalism, postconditions are formulated as predicate transformers that ought to facilitate chaining of refactorings where refactorings in the chain set up program properties for refactorings later in the chain.[19] Unlike this text, he does not distinguish pre- and postconditions by their intent because he does not recognize the value of correctness conditions as program specifications.

The notion of postconditions used in this research is a little different from the one proposed in [39]. There, postconditions in their original definition specify how valid assertions are transformed by the transformation into other valid assertions as long as the transformation is correct – even though this notion is reduced to test cases. "Semantic equivalence" in [39] is supposed to be retained if the preconditions of a refactoring are satisfied. Postconditions in [39] are therefore trivially induced by the program transformation. In this text however, postconditions are conditions that are *necessary for the correctness of the refactoring*, but are checked in the transformed program instead of the original program. Postconditions in this text are similar to the *instrumentation introduced by what Roberts refers to as "dynamic refactoring"* even though it is before the program transformation that the instrumentation and testing is supposed to be performed. [39, p. 60] briefly mentions that tests can be performed after the transformation. The matter is more difficult because he considers functions as well as

[18]From [39, p. 9] "Some of these approximations are particularly poor, such as the analysis for object ownership presented by Opdyke [. . .]"

[19]I disagree with his view. There are only very few properties where it is actually useful (such as aliasing, "IsExclusive"). Moreover, the degree to which refactorings that depend on such predicates are preceeded with refactorings that guarantee them (such as introducing a new field and assigning a fresh object to it), is severly limited.

predicates. functions are used to extract properties that are needed for the refactoring transformation itself while predicates can be tested afterwards. This is not an issue for modern languages that are sufficiently static for all known refactorings. So in [39] the instrumentation is removed after the tests and before the refactoring takes place. This contrasts with our approach that *leaves* all dynamic checks in the program and makes sure that executing the program will *never* cause unintended results due to refactoring independent of whether or not tests are available. Unfortunately, "dynamic refactoring" has not been implemented in the (at least not in the publicly available) Refactoring Browser.

Comparison In this text, for the refactoring to be correct, both the pre- and the postconditions must hold. Postconditions are intended to be checked during execution of the transformed program as *code instrumentation* or verified using static analysis. The code remains there in either case. Code instrumentation is the key to composable refactorings in the face of difficult transformations: The code that checks postconditions is transformed together with subsequent transformations. Composite preconditions can thus still be derived from the transformation – keeping all the advantages of [39] and adding the possibility to analyse – instead of merely argue about – conceptually advanced refactorings like "Move Field" based on the operational semantics of the programming language.

The disadvantage of the approach taken with postconditions here is that annotations must remain in the program and can only be removed by the programmer who needs to know when he introduces transformations that invalidate the requirements of earlier refactorings. This can also be seen as an ingenious feature: refactoring transformations actually make assumptions the programmer has about the program explicit and transform them into annotation that can be useful for subsequent program verification.

Roberts negates the necessity for formal correctness proofs. [39, p. 19 top] The significance of Roberts' work is the implementation of the "Smalltalk Refactoring Browser", a practical and useful refactoring tool. His main research contribution, dynamic refactoring "has not yet been incorporated into the publicly available tool".

Roberts' work on composite refactorings has been extended by Cinneide [9, 10]. A formal legitimation for primitive refactorings is not given. Cinneide reckons that behavior preservation is too complex, formal semantics are too intractable and approaches based on them "cannot therefore be currently expected to produce a working software tool". I aim at showing the opposite in this thesis. I also illustrate that postconditions are helpful even without complete unit test coverage.

1.4.3. Graph-based formalization [29]

The papers written by Tom Mens et al. focus more on specifying the actual transformation precisely rather than proving correctness of refactorings [29]. They specify programs as graphs and refactorings as graph transformations. The graphs that represent programs are very similar to abstracts syntax trees and quite similiar to the tree form I am using in programs. The difference is that *abstraction uses* have back-pointers to the *abstraction definition*. I.e., method calls for instance have a back-edge to the method implementation, classes have an edge to the superclass, etc. Refactorings as graph transformations are advertised as particularly concise, transparent, elegant and expressive. The article lists a few refactorings that are claimed to be "*the* list of primitive refactorings" (emphasis added). The list does not contain refactorings to reorganize data.

The kind of behavior preservation the paper examines is not based on an operational model. Instead, three kinds of "behavior preservation" are presented: access preservation, update preservation, call preservation, i.e., whether the implementation accesses/updates the same variables and calls the same methods after the transformation as before the transformation. These notions are formalized by "graph expressions" that are *regular* paths specifications in the graph. Unfortunately, this notion of behavior preservation is not based on an operational model. I.e., every non-trivial example will violate behavior preservation even if access/update and call preservation is guaranteed. Even though this has not been the primary goal, it is not easy to see where the presented formalism can be most useful.

1.4.4. Model refactoring [18]

The research reviewed so far describes refactoring as a transformation on source code or an abstracted version of source code like a variant of an abstract syntax tree. This is also the approach taken in this text. Many other attempts to formalize refactorings are based on a *model* of the source code. The model is meant to capture certain aspects and relevant design decisions of the program. One example of a modelling language is UML.

UML refactoring operates on the UML model of the program instead of on the actual program representation. The advantage of model-driven approaches is that they are more language independent than source-code refactoring. The disadvantages are the same as the known weaknesses of model-driven development advertised as a general development strategy: in general, the model is an incomplete specification, code and model can diverge, late design and other changes cannot be propagated back, model validation has limited significance. A particularly obvious instance is the fact that abstraction uses (e.g., method calls) in class diagrams cannot be adjusted.

There are various efforts to formalize model-refactoring or focus on design changes in refactorings [5, 28, 43]. I only discuss one [18].

UML class models provide an approximation to the static class definitions of a program. Gheyi et al. describe in [18] how refactoring could be done for Alloy, a modeling language that avoids some of the complexities of UML class diagrams and has a simple semantics based on sets.[20] The original and its transformed counterpart are now called equivalent if all instantiations of the model (i.e., set assignment that satisfy the model constraints) have a corresponding instantiation in the transformed model that is given by some mapping function and vice versa.

1.4.5. Refactorings as refinements [11]

Contribution

Márcio Lopes Cornélio [11] proposes to use the refinement relation as the equivalence criterion for refactorings. He uses the notation introduced in [33]. Refinement guarantees functional properties, i.e., the relation between initial and final state. Refactorings as understood in the context of my research is concerned with the externally visible behavior however. Externally visible behavior has to be explicitly modelled (see [33, p. 133]). Behavioral equivalence is thus not built into the formalism, but has to be built on top of it.

In chapter 3, additional constraints[21] are introduced that describe a relation between data in the original and transformed program. Such a relation is needed in [11] as well. It is conventionally called the *coupling invariant* of a refinement. Coupling invariants have to hold all over the program unlike the corresponding concept used in this text. This makes it more difficult to formalize new refactorings that cannot be decomposed into existing refinement steps. Refinement does not allow to reason about the program directly. This is done on the level of the refinement calculus. The reason is that programs are embedded in a shallow manner. In this research, programs are syntactic entities that are interpreted by a machine, so there is no switching back and fourth between calculus and program.

An additional challenge with refactoring as refinement [11] is that refinement calculi are traditionally less often applied to reference semantics. In fact, it is difficult to find a text on refactoring that admits itself to such complexities. Gheyi [11] also assumes copy-semantics. Many difficulties however stem from the fact that references and aliasing are omnipresent in object-oriented languages (see also the extensive discussion of this topic in Roberts' thesis [39]).

The fundamental difficulty is that refinement was not conceived for reasoning about improvements to existing code but for deriving code from a specification – integrating specifications and executable constructs. Most seriously, the resulting program is supposed to terminate if the original does. That means that conditions cannot be checked

[20] A tutorial can be found at http://alloy.mit.edu/tutorial3/alloy-tutorial.html
[21] called β

dynamically after the transformation. They cannot be easily checked ahead of the transformations either. Yet, the conditions that have to be checked for refactorings necessarily remain the same.

Example: The transformation rule "Move Attribute" [11, Rule 4.3] roughly corresponds to the "Move Field" refactoring. A field is moved from one object to a different object. A required *invariant* is that the object where the field is moved does actually exist. This is itself not trivial to check.

	Refinement [11]	Refactoring (this text)
Termination	Respected	Ignored
Composition	Preconditions only	Pre-/postconditions
Dynamic checks	Impossible	Where necessary
Determinism	Increasing	Indifferent
Embedding	Shallow	Deep
Faithfulness	Limited to the calculus – which isn't the programming language[a]	Limited by your imagination – consider [42]!
Level	Abstraction/model	Actual program
I/O	Model-dependent/unspecified	Uninterpreted

[a]Refinement calculi are quite fixed compared to program semantics, which often depend on the properties to be examined.

Table 1.1.: Main dissimilarities between refactorings as refinements and this research

One difference in the way abstractions are used and introduced, i.e., in their *style*. In refinement, the program is considered a *mathematical* entity and everything it deals with is defined *semantically* as well. I/O is a good example: It is modelled *inside* the program. Visible intermediate states may be ignored. The same is true for objects and references even though the more features that are introduced that are close to some actual programming languages, the more similar refinement becomes to the conventional approach to define programs as (to a greater or lesser degree) syntactic entities that are interpreted by an operational semantics. Operational semantics allow to intuitively reason about the "inside" and the "outside" of the program. They are traditionally much less definitive than refinement calculi partly because the operational semantics may also be subject to refinement (for instance when formulated as ASMs [6]).

Having said that, I must stress that refactoring as refinement and the framework I present here are related – both are using different varying correspondences between initial and refactored code. Instead of sticking with the powerful but rigid idea and framework of refinement however, I have basically taken the liberty of coming up with novel framework that has proven useful to answer the unique challenges of refactoring in practical applications.

1.5. Scope

In the present text, I confine myself to semantic equivalence. Semantic equivalence is only possible if refactorings always result in a compilable program. This is not discussed explicitly, but it is silently assumed. This is not a audacious assumption: I give correctness proofs. They show that semantic equivalence is retained. For any sensible notion of semantic equivalence, the program that results from refactorings must be interpretable as well. This is a weaker notion than compilability, but it is as far as I can get without specifying a formal static semantics of the programming language discussed here, which is beyond the scope of this thesis.

I try to cover all kinds of refactorings in this text, which I treat as source transformations. Only source transformations capture the whole operational complexity that is needed for behavioral equivalence. Behavioral equivalence is defined in the broadest way possible without refering to pecularities of external system components.

The technique presented here does not depend on program specifications or formal languages that are not normally usable in software systems but nonetheless required by other formally justified methods for refactoring.

Unfortunately, there is nothing apart from existing informal textbooks on refactoring to measure the established criteria against since the local criteria established here will be necessarily conservative in all but the most trivial examples.

The quest to find formal justification for atomic refactorings forces you to break down a refactoring into truly atomic steps. This is something that has been advertised in textbooks on refactorings, but has hardly been adhered to.

Audience This Thesis meant to serve as the prolegomena to a complete treatise on practical refactoring of source-code with specifications as executable by machines is written to be readable by both the uninitiated student, practicioners and researchers in the field, not just the supervising professor.

Different people differ in their opinion what refactorings encompass, how they should be used and how they should be investigated – just compare the software engineering text by Kerievsky [25] with stringent model refactorings of Alloy specifications in [18] that is hardly practical and yet formally elegant. As people interested in refactoring come from various backgrounds, I have to limit my use of specific notations that may be well known in certain communities while looking alien in others. I explain everything in terms that should be known to any undergraduate in computer science. The numerous footnotes may well be skipped. They contain remarks I deemed interesting but not essential for understanding the text.

1.6. Organisation and overview

This research is split into chapters. Each chapter is self-contained except for chapter 3 on whose definitions the subsequent chapters depend.

Chapter 2 contains the obligatory introductory example of the stepwise evolution of a small sample application. Each step contains a pointer to a section in chapter 4 that discusses the refactoring and proves it correct. This chapter exemplifies my personal view how complex transformations should be decomposed. This chapter is meant to make the thesis more readable and can be skipped if you are only interested in concrete results.

Chapter 3 explains the mathematical model used for equivalence and the rationale behind it. It shows the operational semantics on which the equivalence notion relies and classifies refactorings according to the proof strategy they require. It also compares and contrasts the approach to other approaches.

Chapter 4 considers a few refactorings that are essential for any practical refactoring tool. In particular, all refactorings in chapter 2 are discussed and all the refactorings used in the introductory example in [16]. I show that all these refactorings are "simple" in the sense that they do not strictly require postconditions[22] that are evaluated at runtime yet they are important as illustrated by the fact that both the examples in [16] and the present text are considered.

All the refactorings implemented in customary refactoring tools for Java are also of this category. Up to renaming, they are both limited in the lexical scope of their changes and the number of changes that have to be made. This was a goal, not a coincidence as I explain in section 1.3. The requirement forces me to change the definition of some refactorings. The analysis to be performed for all these refactorings is merely local or can easily and well be approximated.[23] Chaining refactorings of this kind is simple and does not need the rather penetrative heavyweight conceptual framework in [39] and its successors for reasoning about precondition transformers. This chapter is also meant to illustrate that truly difficult refactorings have completely different intricacies that must be tackled differently.

Chapter 5 is the most delicate chapter because it covers refactorings that are not easily implementable without dynamically enforced postconditions.

Chapter 6 is a tutorial that discusses how to extend the Eclipse IDE, Visual Studio and use the Microsoft's Common Compiler Infrastructure to implement tool support for your own – possibly domain specific – refactorings. The refactoring implemented as an example is that of chapter 5. It is almost completely unrelated to the more theoretical considerations in the other chapters.

[22]Even though they may be used to generalize the refactorings.
[23]An alias analysis is required if full precision is needed.

1.7. Contributions

The contributions of this thesis are the following. They will be discussed and explained in the text and are depicted in figure 1.5.

1. A scheme for instrumenting source code with postconditions as specifications that guarantee correctness of refactorings.
2. A definition of equivalence that
   - Approximates intuition well
   - Is parameterized over concrete operational semantics
   - Suggests a way to derive minimal correctness conditions
3. An implementation of these ideas.

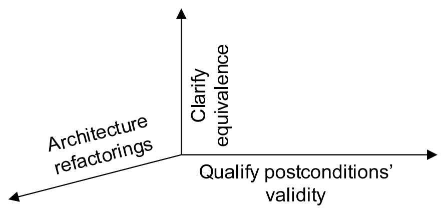


Figure 1.5.: Orthogonal axes of contribution

2. Refactoring: A Motivating Example For Simple Refactorings

In this chapter, I consider an example similar to the introductory example in [16, chap. 1]. The program is refactored step by step. It ought to be illustrating the practical applicability of the approach chosen in this text: All refactorings used here are analyzed. The full sourcecode for the example and all steps can be found in the `ex`NNN subdirectory.

This chapter does not use the "Move Field" refactoring. It is illustrated by the sample project `TheBank` that comes with this text.

I have to repeat the disclaimer in [16]: The sample program is necessarily simple, it should be imagined in the context of a much larger system. It calculates the summary (or receipt) of a shopping cart as a text string. There are three types of groceries that can be bought at this store: bread, cheese and wine. They have different characteristics, the relevant of which is how the price is calculated. In general, the price customers have to pay is given by the wholesale price times some multiplier that abstracts the gross margin on that product. There are three classes: `Product`, `Purchase`, `ShoppingCart`. `ShoppingCart` contains a list of `Purchase`s, which in turn refer to the corresponding `Product` abstractions. The difference between `Purchase` and `Product` is simply that `Purchase`s are always objects inside `ShoppingCart`s whereas `Products` may be used elsewhere.

Here is an example receipt returned by the program:

```
Shopping cart for Fabian
 Multigrain Loaf                                                 2.3
 Société Roquefort                                               3.31
 Rioja DOCa 1995, Gran Reserva Imperial C.V.N.E.                 45.0
 Ciabatta                                                        2.1
 Mango Bread                                                     1.2
 Rioja DOCa 1995, Gran Reserva Imperial C.V.N.E.                 45.0
 Gruyère                                                         6.15
 Pinot Noir AOC 2004, Château d'Auvernier                        13.5
 Cabernet Sauvignon Grande 2001, Réserve Los Vascos Rothschild 15.0
 Ciabatta                                                        2.1
Amount is 135.66
```

The initial version of the `ShoppingCart` contains the core routine `summary()` for the receipt. It also calulates the prices and then adds them up to get the total amount to be paid by the customer. The field that contains the `Purchase`s is `items`. Inside `summary`, the margin multiplier is calculated depending on the product category. The multiplier depends on some external system variables that are different for cheese, wine and bread. In this example, I get their values from static function calls on the `Apollo` and `Hablo` classes.

ShoppingCart, v. 00

```java
import java.util.ArrayList;
import java.util.Collection;

public class ShoppingCart {
    public String customerName;
    private Collection<Purchase> items = new ArrayList<Purchase>();

    public ShoppingCart(String name) {
        this.customerName = name;
    }

    public String getName() {
        return customerName;
    }

    public void add(Purchase r){
        items.add(r);
    }

    public String summary(){
        double totalAmount = 0;
        String result = "Shopping␣cart␣for␣" + getName() + "\n";
        for (Purchase x : items) {
            double multiplier;

            switch(x.getProduct().productCategory){
            case Product.CHEESE:
                multiplier = Apollo.getDVal("CHEESE");
                break;
            case Product.WINE:
                multiplier = Apollo.getDVal("WINE");
                break;
            case Product.BREAD:
                multiplier = Hablo.cachedConfig(6823);
                break;
            default:
                throw new RuntimeException();
            }
            double price = Math.ceil(100*x.getProduct().wholesalePrice*
                multiplier)/100;

            result += "\t"+x.getProduct().title+"\t"+price+"\n";
            totalAmount += price;
        }
        result += "Amount␣is␣" + totalAmount + "\n";
        return result;
    }
}
```

The `Purchase` class is a mere intermediary for the `Product` that would take into account discounts and the like in a more realistic version of the program. In this exemplary setting however, the class is pretty much only an indirection to `Product`. Purchase, v. 00

```
public class Purchase {
    private Product product;
    public Purchase(Product prod) {
        this.product = prod;
    }
    public Product getProduct() {
        return product;
    }
}
```

`Product` is the class that contains the actual information about a product: it includes the wholesale price, the category and the name. All of these are public fields. The category is an integer that can assume the values of the constants `BREAD`, `CHEESE` and `WINE`. Product, v. 00

```
public class Product {
    public static final int BREAD  = 0;
    public static final int CHEESE = 1;
    public static final int WINE   = 2;

    public String title;
    public int productCategory;
    public double wholesalePrice;

    static Product create(int code, double wholeSalePrice, String title){
        return new Product(code, title, wholeSalePrice);
    }

    private Product(int category, String title, double price) {
        this.productCategory = category;
        this.title = title;
        this.wholesalePrice = price;
    }
}
```

"Extract Method"

The most obvious deficiency of the program is the method `summary()` which is too long and too cluttered with unrelated activities. The refactoring for this is "Extract Method" and the most obvious piece of code to extract is the calculation of the margin multiplier. We call this new method `getMarginMultiplier`. Its only argument `x` is the `Purchase` that is a local variable in `summary` and must accordingly be available in `getMarginMultiplier`. Note that the method could just as well be made static. It is not strictly necessary to implement it as a static method because it is only called from an instance context. Therefore, I didn't do it.

ShoppingCart, v. 01

```
    public String summary(){
        double totalAmount = 0;
        String result = "Shopping␣cart␣for␣" + getName() + "\n";
        for (Purchase x : items) {

            double multiplier = getMarginMultiplier(x);
            double price = Math.ceil(100*x.getProduct().wholesalePrice*
                multiplier)/100;

            result += "\t"+x.getProduct().title+"\t"+price+"\n";
            totalAmount += price;
        }
        result += "Amount␣is␣" + totalAmount + "\n";
        return result;
    }

    private double getMarginMultiplier(Purchase x) {
        double multiplier;
        switch(x.getProduct().productCategory){
        case Product.CHEESE:
            multiplier = Apollo.getDVal("CHEESE");
            break;
        case Product.WINE:
            multiplier = Apollo.getDVal("WINE");
            break;
        case Product.BREAD:
            multiplier = Hablo.cachedConfig(6823);
            break;
        default:
            throw new RuntimeException();
        }
        return multiplier;
    }
```

"Extract Method"

The most obvious next candidate is the calculation of the price. Again, we need to pass the purchase made to the routine:

ShoppingCart, v. 02

```
    private double getPrice(Purchase x) {
        double multiplier = getMarginMultiplier(x);
        double price = Math.ceil(100*x.getProduct().wholesalePrice*
            multiplier)/100;
        return price;
    }
```

"Extract Method"

The two methods we have introduced both take an argument of type `Purchase` but they both operate only on the `Product` that is returned from `getProduct()`. It suggests itself to extract methods that take `Product`s instead of `Purchase`s. The remaining `getMarginMultiplier` method looks like this: ShoppingCart, v. 03

```
    private double getMarginMultiplier(Purchase x) {
        double multiplier;
        Product prod = x.getProduct();
        multiplier = getMarginMultiplier(prod);
        return multiplier;
    }
```

"Inline Method"

As noted before, `getPrice` suffers from the same illness as `getMarginMultiplier`: It does not really need the `Purchase`, it merely needs the attached `Product`. This is not immediately obvious because we're calling `getMarginMultiplier` with a `Purchase` argument. Therefore, we have to inline `getMarginMultiplier` first: ShoppingCart, v. 04

```
    private double getPrice(Purchase x) {
        double multiplier = getMarginMultiplier(x.getProduct());
        double price = Math.ceil(100*x.getProduct().wholesalePrice*
            multiplier)/100;
        return price;
    }
```

"Introduce Temp"

It is only after we've done the inlining that we can add an additional method `getPrice` that takes a `Product` as its argument. In order to make things easier, we first "Introduce Temporary Variable". In the kernel language defined in chapter 3, all method call results are assigned to local variables. ShoppingCart, v. 05

```
    private double getPrice(Purchase x) {
        Product prod = x.getProduct();
        double multiplier = getMarginMultiplier(prod);
        double price = Math.ceil(100*prod.wholesalePrice*multiplier)/100;
        return price;
    }
```

"Extract Method"

Extracting the price calculation that now only depends on `prod` is easy. ShoppingCart, v. 06

```
    private double getPrice(Purchase x) {
        Product prod = x.getProduct();
        return getPrice(prod);
    }

    private double getPrice(Product prod) {
        double multiplier = getMarginMultiplier(prod);
        double price = Math.ceil(100*prod.wholesalePrice*multiplier)/100;
        return price;
    }
```

"Inline Method"

ShoppingCart, v. 07

Have a look at `summary()` again. We are still invoking the `getPrice` method that we wanted to get rid of. Let's inline this call as well.

```
            double price = getPrice(x.getProduct());
```

"Delete Obsolete Elements"

When we've done that, we can finally delete the "deprecated" `getPrice(Purchase)` and `getMarginMultiplier(Purchase)`. Note that `getPrice` could be a method in in `Purchase` but that would be speculatively general.[1] At the moment, unlike suggested on page 27, there are no discounts that are stored in `Purchase`, so there is no reason to have `getPrice` in `Purchase`.

ShoppingCart, v. 07

Puhh... That's already a bit better than at the beginning. We are now at step 7 of the example and it is worth having a look at the complete implementation of `ShoppingCart` as this is the last refactoring that operates exclusively on `ShoppingCart`.

```
import java.util.ArrayList;
import java.util.Collection;

public class ShoppingCart {
    public String customerName;
    private Collection<Purchase> items = new ArrayList<Purchase>();

    public ShoppingCart(String name) {
        this.customerName = name;
    }

    public String getName() {
        return customerName;
    }

    public void add(Purchase r){
        items.add(r);
    }

    public String summary(){
        double totalAmount = 0;
        String result = "Shopping␣cart␣for␣" + getName() + "\n";
        for (Purchase x : items) {
            double price = getPrice(x.getProduct());

            result += "\t"+x.getProduct().title+"\t"+price+"\n";
            totalAmount += price;
        }
        result += "Amount␣is␣" + totalAmount + "\n";
        return result;
    }

    private double getPrice(Product prod) {
```

[1] I try to allude to the "bad smell" "speculative generality" [16]: Do not generalize your code beyond current needs.

```
34          double multiplier = getMarginMultiplier(prod);
35          double price = Math.ceil(100*prod.wholesalePrice*multiplier)/100;
36          return price;
37      }
38
39      private double getMarginMultiplier(Product prod) {
40          double multiplier;
41          switch(prod.productCategory){
42          case Product.CHEESE:
43              multiplier = Apollo.getDVal("CHEESE");
44              break;
45          case Product.WINE:
46              multiplier = Apollo.getDVal("WINE");
47              break;
48          case Product.BREAD:
49              multiplier = Hablo.cachedConfig(6823);
50              break;
51          default:
52              throw new RuntimeException();
53          }
54          return multiplier;
55      }
56  }
```

"Make Method Instance"

Now have a look at `getMarginMultiplier()` and `getPrice()` above. These are instance methods but they do not access any instance fields nor does their functionality depend on dynamic dispatch – they are not overriden at all. We could very well make them static and keep them in `ShoppingCart`. These two methods do however operate only on the first parameter of type `Product`, a strong indication that they are more appropriately moved there. The first parameter `prod` becomes **`this`**. Let's do it step by step and first move `getMarginMultiplier()` (Only the first **`case`** is listed for brevity, see line numbers) The method is declared **`final`** as it derives from a static method that cannot be overriden either. Making it overridable could be confusing people who might think that different "kinds" of (sub-)classes can have different implementations of this function.

Product, v. 08

```
20      public final double getMarginMultiplier() {
21          double multiplier;
22          switch(this.productCategory){
23          case Product.CHEESE:
24              multiplier = Apollo.getDVal("CHEESE");
25              break;
26          /* other cases omitted */
34              throw new RuntimeException();
35          }
36          return multiplier;
```

We have to rewrite all references to the moved method. Instead of

ShoppingCart, v. 07

```
34          double multiplier = getMarginMultiplier(prod);
```

ShoppingCart, v. 08

We have to write

```
34          double multiplier = prod.getMarginMultiplier();
```

Having done that, it is now possible to move `getPrice()` as well: We remove it from `ShoppingCart` and add it to `Product` replacing all references to `prod` by **`this`**.

Product, v. 09

```
38      public final double getPrice() {
39          double multiplier = this.getMarginMultiplier();
40          double price = Math.ceil(100*this.wholesalePrice*multiplier)/100;
41          return price;
42      }
```

We do of course have to adjust the reference in `ShoppingCart.summary()`. This leaves us with the following method body for `ShoppingCart.summary()`

ShoppingCart, v. 09

```
20      public String summary(){
21          double totalAmount = 0;
22          String result = "Shopping␣cart␣for␣" + getName() + "\n";
23          for (Purchase x : items) {
24              double price = x.getProduct().getPrice();
25
26              result += "\t"+x.getProduct().title+"\t"+price+"\n";
27              totalAmount += price;
28          }
29          result += "Amount␣is␣" + totalAmount + "\n";
30          return result;
31      }
```

It is already much less cluttered than at the beginning but the for-loop is still doing two things that are largely unrelated: adding the price to the output string (line 26) and accumulating the `totalAmount` Because there is *no* dependency between these two activities, we can directly split the loop into two: Unrelated activities should be unbundled as far as possible, so as to be able to move them to different classes and to give them meaningful names as separate procedures.

ShoppingCart, v. 10

```
23          for (Purchase x : items) {
24              double price = x.getProduct().getPrice();
25              result += "\t"+x.getProduct().title+"\t"+price+"\n";
26          }
27          for (Purchase x : items) {
28              double price = x.getProduct().getPrice();
29              totalAmount += price;
30          }
```

“Extract Method”

This yields another opportunity to factor out functionality using “Extract Method”. The new method is called `getTotalAmount()` and accesses – unlike the methods we had to move to `Product` – the fields of `ShoppingCart`.

ShoppingCart, v. 11

```
    public String summary(){
        String result = "Shopping␣cart␣for␣" + getName() + "\n";
        for (Purchase x : items) {
            double price = x.getProduct().getPrice();
            result += "\t"+x.getProduct().title+"\t"+price+"\n";
        }
        double totalAmount = getTotalAmount();
        result += "Amount␣is␣" + totalAmount + "\n";
        return result;
    }

    private double getTotalAmount() {
        double totalAmount = 0;
        for (Purchase x : items) {
            double price = x.getProduct().getPrice();
            totalAmount += price;
        }
        return totalAmount;
    }
```

“Rename Method”

The `summary` method could also be renamed to `getReceipt`, which is a bit more suggestive.

“Replace Representation”

We’ve now done all the trivial refactorings that are more or less directly applicable. The next refactoring is simple, but may be less familiar from books like [16] that focus on high-level abstractions that are not always ideally suited for automatic tools. The low-level refactoring I want to use now is “Replace Representation”. Remember that the type of product that is being sold is encoded in the variable `productCategory`. `productCategory` assumes values in $\{0, 1, 2\}$. It requires that `productCategory` objects can be tested for equality. Because the set $\{0, 1, 2\}$ cannot be specified in Java, the program is using **`int`** as an approximation. What we’re doing now is introduce a new set $\{\texttt{ZERO}, \texttt{ONE}, \texttt{TWO}\}$ and replace all uses of the old values by the new values. It is also necessary to replace all types representing the old set by a type approximating the new set. How this refactoring can be tool supported is discussed in section 4.5.

The values are declared in constants in a new class `Category` inside —Product.java—. The values themselves are object values of that class `Category`. I.e., `Category` is the type we will have to replace **`int`** by in contexts where it is used to approximate $\{0, 1, 2\}$.

We could take any other objects with value semantics that support equality tests. Of course, the decision to choose objects is not completely unintentional. The idea is to

Product, v. 12

turn the `Category` objects into *strategies* later.

```
class Category{
    public static final Category ZERO  = new Category();
    public static final Category ONE   = new Category();
    public static final Category TWO   = new Category();
}
```

The constant declarations in `Product` remain, `productCategory` has to be declared with a different type and the creation routine as well as the constructor have their `category` parameter changed

Product, v. 12

```
8       public static final Category BREAD  = Category.ZERO;
9       public static final Category CHEESE = Category.ONE;
10      public static final Category WINE   = Category.TWO;
13      public Category productCategory;
16      static Product create(Category code, double wholeSalePrice, String
            title){
17          return new Product(code, title, wholeSalePrice);
18      }
19
20      private Product(Category category, String title, double price) {
21          this.productCategory = category;
22          this.title = title;
23          this.wholesalePrice = price;
24      }
```

To make things more legible, we can inline the constants in `Product` and rename the respective constants in `Category`.

Product, v. 13

```
class Category{
    public static final Category BREAD  = new Category();
    public static final Category CHEESE = new Category();
    public static final Category WINE   = new Category();
}
```

On our way from the atomic `Category` to a proper *strategy*, let's differentiate the `BREAD`, `CHEESE`, `WINE` objects by their type. According to the substitution principle, this does not change the program semantics as long as it cannot be determined whether an object is of type `Product` or a subtype thereof. This is the case in Java when `getClass()` and the special field **`class`** are unavailable (as well as more general reflection facilities).

Product, v. 15

```
class Category{
    public static final Category BREAD  = new CatBread();
    public static final Category CHEESE = new CatCheese();
    public static final Category WINE   = new CatWine();
}

```

```
class CatBread  extends Category{}
class CatCheese extends Category{}
class CatWine   extends Category{}
```

“Replace Expression”

Let’s make things a bit more clear by replacing the object identity test in method `getMarginMultiplier()` by a *type* test before we exploit the possibility to use dynamic dispatch on the subclass hierarchy we’ve created.

Product, v. 16

```
    public final double getMarginMultiplier() {
        double multiplier;
        if(this.productCategory instanceof CatCheese)
            multiplier = Apollo.getDVal("CHEESE");
        else if(this.productCategory instanceof CatWine)
            multiplier = Apollo.getDVal("WINE");
        else if(this.productCategory  instanceof CatBread)
            multiplier = Hablo.cachedConfig(6823);
        else
            throw new RuntimeException();
        return multiplier;
    }
```

“Replace Type Tests with Dynamic Dispatch”

It thus becomes obvious that the code in the individual branches can be moved into a dynamically dispatched method in `Category`. It is called `getMarginMultiplier()` for the sake of consistency. The `Category` class has to be made **`abstract`** to be able to leave the method undefined for `Category` itself.

Product, v. 17

```
abstract class Category{
    public static final Category BREAD  = new CatBread();
    public static final Category CHEESE = new CatCheese();
    public static final Category WINE   = new CatWine();

    abstract public double getMarginMultiplier();

}

class CatBread  extends Category{
    public double getMarginMultiplier(){
        return Hablo.cachedConfig(6823);
    }
}
class CatCheese extends Category{
    public double getMarginMultiplier(){
        return Apollo.getDVal("CHEESE");
    }
}
class CatWine   extends Category{
    public double getMarginMultiplier(){
        return Apollo.getDVal("WINE");
    }
}
```

```

public class Product {
    public final double getMarginMultiplier() {
        double multiplier;
        multiplier = productCategory.getMarginMultiplier();
        return multiplier;
    }
}
```

"Inline Method"

Have a look at the getMarginMultiplier() method in Product above. It is merely delegating the call to the corresponding Category. It might be nice to inline it in getPrice() as getPrice() is also a member of Product and need not be shielded from such intricacies.

Product, v. 18

```
    public final double getPrice() {
        double multiplier = this.productCategory.getMarginMultiplier();
        double price = Math.ceil(100*this.wholesalePrice*multiplier)/100;
        return price;
    }
```

"Use Syntactic Sugar"

As a last step, we replace the Category subclass hierarchy by the equivalent synactic extension in Java 1.5 – enumerations.

Product, v. 19

```
enum Category{
    BREAD{  public double getMarginMultiplier(){
                return Hablo.cachedConfig(6823); }},
    CHEESE{ public double getMarginMultiplier(){
                return Apollo.getDVal("CHEESE"); }},
    WINE{   public double getMarginMultiplier(){
                return Apollo.getDVal("WINE");   }};

    public abstract double getMarginMultiplier();

}

public class Product {
    public String title;
    public Category productCategory;
    public double wholesalePrice;

    static Product create(Category code, double wholeSalePrice, String
        title){
        return new Product(code, title, wholeSalePrice);
    }

    private Product(Category category, String title, double price) {
        this.productCategory = category;
        this.title = title;
        this.wholesalePrice = price;
    }

    public final double getPrice() {
        return Math.ceil(100*this.wholesalePrice
                *this.productCategory.getMarginMultiplier())/100;
    }
}
```

The goal of this last step is to keep the definitions of all price calculations together in a single compilation unit. This may render it more inconvenient to add new categories or the work on different categories independently. Keeping the interactions with external systems together can also be valuable however.

3. Formalizing Equivalence For Refactorings

This chapter introduces the framework I am using to prove correctness of refactorings, i.e., whether refactorings retain externally visible behavior or not. This research is using basically the same definition of “externally visible behavior” as the landmark thesis by William Opdyke: *Two program are behaviorally identical if they yield the same output for the same input ceteris paribus.*

Summary of prevented problems

This definition avoids the inadequacies of other formal definitions as discussed in section 1.4 that have been proposed. In *refinement-based formalizations* [11] for instance, visible intermediary states are ignored: If only the input-*values*-output-*values* relation has to be retained as in refinement-based refactorings, the program may do arbitrary visible I/O (e.g., write output to the console or open new windows) without compromising the equivalence of the program. This is not a safe conceptualization. See section 1.4.5 for details. Unlike *model-based refactoring* that does not have any operational concepts, Opdyke’s definition affects the whole of the refactoring, including the imperative formulation, not only the structural part of it and is therefore directly applicable for refactoring tools.[1] This definition is also substantially more useful than other definitions of “equivalence” that do not aim at any kind of behavioral equivalence but merely respect lexical properties of the source code [29]. At the same time, I try to avoid the fatalism of Roberts [39] and his followers who state that the “complexity of the behavior preservation proofs for non-trivial transformations will be intractable”. [10]

Let me quickly summarize the features the notion of equivalence to be used in this text should have over and above the features of definitions that have already been postulated. The formalization of externally visible behavioral equivalence should be. . .

- . . . easy to prove for non-trivial refactorings (but also for trivial ones).

[1]The argument above certainly applies in the case of UML/Alloy refactorings as discussed in section 1.4.4. There are also non-structural models. Operational models are potentially possible but the benefit forgone are severe: Operational models are normally used to derive implementations by refinement (Z,B, etc.). This is largely incompatible with model refactoring and subsequent back-propagation. If refinement is not used *and* there is no mapping from modelling language to implementation language, automatic back-propagation becomes impossible anyway (e.g., AsmL). If operational models are complete, manual back-propagation of model refactorings could at least be verified. Unfortunately, unconstrained model checking is not decidable either. I do not know of any such attempts however. See section 1.4.4 for details.

- . . . at most a conservative approximation of the definition above modulo timing and memory access patterns, which are difficult to analyze formally. It must therefore be precise, i.e., no program that intuitively violates externally equivalent behavior (print “a” instead of “b” on the console) should satisfy the formal definition.

- . . . complete (no part of the program is unaffected by the definition): The definition must lead to fully functional refactoring tools.

- . . . easily transposed to other languages and different semantics.

The syntax and semantics of programs depend on the language that is used. Refactorings cannot be discussed without refering to a concrete implementation language and a concrete programming environment. The notion of “generic refactorings” [26] does not pay tribute to the fact that in spite of all similarites, many refactorings on the source level strongly depend on language-specific features. A language that does not allow named formal method parameters (like the original Perl language) is not useful for investigating the “Rename Parameter” refactoring. Correctness conditions are also affected: Replacing class instantiations by instantiations of (empty) subclasses would be unconditionally possible in Java if it were lacking the special **`class`** field[2]. It is undeciable in general whether a program depends on the *exact* type if it uses the **`class`** field.

Characteristics of the formalism The language I am using is a sequential subset of Java. It could also be called a subset of C# – the covered features are the same. The subset is sequential because most programs are largely sequential to a good approximation. Parallel programs depend on memory models that are not or not precisely defined as well explained in Stärk’s paper [40, section 6].

Language used

A big-step operational semantics is used to capture the meaning of program constructs. The structure that is interpreted by the semantics is the domain of the refactoring, i.e., the same mathematical program representation is used for the semantics and the refactoring. A program representation is always a complete representation of the whole program. This is necessary because refactorings potentially affect the whole program: Renaming a public field must change all accesses – which may be anywhere in the program. Reflection cannot reasonably be part of the language covered here.

Semantics used

Programs are represented as trees and program parts are only meaningful in the context of a whole program. Program parts are always identified by paths in the program tree. A statement for instance is the tuple of a program and the path to the statement.

Programs and program parts

[2] . . . as well as `getClass()` and other reflection capabilities.

$$\begin{tikzcd}
\text{Original source} \arrow[r, "\text{Source-to-source}"] \arrow[d, "\text{Deserialization}"'] & \text{Refactored source} \\
\text{Original AST} \arrow[r, "\text{AST-to-AST}"] & \text{Refactored AST} \arrow[u, "\text{Serialization/Source-patching}"']
\end{tikzcd}$$

Figure 3.1.: Refactoring of the intermediary representation

Overview The rest of this chapter is organized as follows: Section 3.1 contains essential definitions of the symbols that will be used in subsequent sections and chapters, e.g., the program state, the transition relation, the refactoring. A summary of the operational semantics is given in section 3.2. Section 3.3 describes limitations and how they can be overcome. Section 3.4 comes up with a classification of refactorings on which the rest of this text is based.

3.1. Programs

This section covers the program representation first and then defines the formal conceptualization of refactorings.

3.1.1. Program representation

The most basic aspect of program transformation is the domain of the program itself. Needless to say, most programming languages do have their source code stored in text files, i.e., streams of characters. Streams of characters are not very handy and do not really correspond to the structured nature of the subset of Java to be covered here well. A more promising approach is to rely on serialization and deserialization of programs (see figure 3.1). The Smalltalk refactoring browser already relied on this scheme and so do probably a whole lot of other refactoring tools.[3] It has already been recognized in Opdyke's thesis [37] that the preservation of non-semantic data like comments and whitespace is imporant for serialization. Roberts' browser relied on his `Formatter` class. The program databases produced by contemporary compilers with detailed source location information suggest a different kind of implementation: the source code is not fully deserialized and then serialized again. Instead, textual replacement is done by the refactoring tool based on the positions in the program database. A very primitive incarnation of this concept is presented as an example in section 6.1.

Whether textual replacement based on position information in the program representation is performed directly or the whole program representation is deserialized with enough information about whitespace and comments does not matter too much. In

[3]Eclipse's refactoring tool relies on patching the source code: only the parts of a source file are updated that have their AST changed [17]. This means that the AST must contain enough information to map individual source element tokens to the abstract syntax elements.

most practical implementations, a mixed model will have to be used. Files that haven't changed during refactoring for instance do not need to be rewritten. On the other hand, it may often be easier to replace whole method implementations by newly serialized method implementations instead of fiddling around with individual statements and figuring out how they can be replaced. Whatever the choice be, the program refactoring itself is from abstract syntax tree to abstract syntax tree and the additional information used for formatting shall be ignored for the discussion here as well as line-numbers, column-numbers and the like. Eclipse's AST rewrite framework implements just this ideal.

A program consists of interface and class definitions:[4]

$$\mathit{Prog} \equiv \mathit{Name} \hookrightarrow \mathit{Decl} \tag{3.1}$$

A class has a superclass and a number of implemented interfaces while an interface does not have a superclass and the first element of the *Decl* tuple is None.

$$\begin{array}{lll} \mathit{Decl} \equiv & (& \\ & \text{superclass} & : (Name)option, \\ & \text{superifaces} & : (Name)list, \\ & \text{fields} & : Fields, \\ & \text{methods} & : Methods, \\ & \text{staticinitializer} & : StaticInitializer \\ ) & & \end{array} \tag{3.2}$$

Notation: $(\alpha)option$

The data-type $(\alpha)option$ denotes the set $\alpha + \{undef\}$ (cf. Appendix B).

no constructor support

Just as in [44], the program structure does not support constructors. The reason is that constructors are normally used to establish object invariants. Fairly often, this is necessary because unstructured initialization is difficult to reason about. On the other hand, many patterns depend on initialization that is performed outside the constructor (e.g., factory methods) or even outside the class to get good factoring.[5] This research is not mainly about how to establish invariants and how to check them statically and I could ignore constructors for that reason alone. But even when static approximations of the

[4]Partial functions are used directly. This simplifies notation dramatically. There is no specific reason to use lists instead like e.g. in [44]. Inter-class relations are finite if the functions are finite, something we can reasonably assume, just as it is reasonable to assume that lists of classes are finite. Both properties can be derived from the finiteness of the program itself if needed.

[5]I call this the code mobility principle, its the refactoring formulation of the single responsibility principle: Responsibilities should be movable between classes without non-local effect. This gets more important as more constraints are statically verified for the code and the programmer is more and more constrained by these. Only C++ supports this principle with "friends" albeit in a pretty cumbersome manner. A less concise but more clear syntax based on single permissions could help. Ceterum censeo: The same is true for visibility qualifiers like **`private`**, **`protected`**, **`public`**, **`internal`**, default visibility, etc.

criteria to be established for the refactorings are considered, it turns out that constructors are of limited usefulness because they constrain the applicability of a refactoring too much.[6] An example can be found in section 5.4.

Inside the fields of a class, field names are mapped to their type.

$$\mathit{Fields} \equiv \mathit{Name} \hookrightarrow \mathit{TypeTag}$$

Fields have a unique type, so the type can also be thought of as being encoded in the name itself. We write $\mathrm{ctt}(f)$ "compile time type" for the declared type of field f. The same applies to local variables. $\mathrm{ctt}(l)$ is the declared type of a local variable l

Fields, methods

The fields in a class are indexed simply by their proper name for each class. This differs from methods, where the whole signature but excluding the declaring class is used to tell the implementations apart. In the heap however, fields are stored with the declaring class made explicit. Field names inside the heap are thus a tuple of declaring class and the proper field name. Lookup and access to fields happens with this pair $C{::}f$ where C is the declaring class and the proper field name is f.[7] The qualifier $C{::}$ in the text is only used to clarify the declaring class of a field, it is omitted if it is clear from the context.

Inside the methods of a class, method signatures are mapped to their declaration.

$$\mathit{Methods} \equiv \mathit{MethodSignature} \hookrightarrow \mathit{MethodDeclaration}$$

The whole signature is taken because overloading is allowed. Method signatures identify the method uniquely and the return type of a method can be considered part of the name.[8] For a method m, $\mathrm{paramTs}(m)$ yields the list of types of its formal parameters and $\mathrm{pNames}(m)$ yields their names. The parameters are represented as a list because overriding does not depend on formal parameter names in Java. $\mathrm{retT}(m)$ yields the return type of m.

A method declaration (*MethodDeclaration*) is returned from the methods of a class given the corresponding signature (*MethodSignature*). A declaration consists of parameters (this time including names), the return type, and the body. The structure of all program parts is summarized in figure 3.2.

[6]If constructors are assumed to establish some invariants, preconditions and the like, programs that do not make use of constructors for good reasons cannot use the refactorings – which is really sad. If constructors are not used for specifying refactoring preconditions, they can be dropped from the language. This is what happend. It does not mean the results cannot be specialized for languages with constructors – the invariants they might establish are just not needed.

[7]The primary aim of this convention is to establish a direct correspondence between data representation and program structure. This correspondence is used in chapter 5.

[8]Note that the return type is not encoded in the signature because it is not used in method lookups to be considered later. Adding it would render method lookup more complicated but wouldn't add anything to the discussion. The same is true for the declaring class.

$$\begin{array}{lll}
\mathit{MethodSignature} \equiv & (& \\
 & \text{mname} & : \mathit{Name} \\
 & \text{paramTs} & : (\mathit{TypeTag})\mathit{list} \\
) & & \\
\mathit{MethodDeclaration} \equiv & (& \\
 & \text{params} & : (\mathit{ParameterName} \times \mathit{TypeTag})\mathit{list} \\
 & \text{retT} & : \mathit{TypeTag} \\
 & \text{body} & : (\mathit{Statement} + \{\text{inherit-method}, \text{ext-body}\}) \\
) & &
\end{array}$$

The declared type of an entity (field, local variable, method arguments and return value) is represented as a *TypeTag*. Opaque basic types (*BasicType*) like `int` and `bool` are type tags as well as class names (*ClassName*), which are also opaque. Alternatively, a type tag can be an array type. As in Java, only one-dimensional arrays are allowed.

$$\mathit{TypeTag} \equiv \mathit{BasicType} + \mathit{ClassName} + \mathit{TypeTag}[]$$

Valid *TypeTag*s are for example: `int`, `bool`, `int`[], C, D[].

Statements are represented as tuple data types. To avoid having to define a named tuple type for every statement individually, positive integers are used as the names of a statements' component. Concrete syntax is used to label values. Setting a field f in an object referenced by a local variable l to the result of the expression e is written as $l.f{\leftarrow}e$ and its type is

$$(1 : \mathit{LocalVariable}, 2 : \mathit{FieldName}, 3 : \mathit{Expression})$$

Expressions can be treated likewise.

Local variables

Analogously to field and method names, local variables may also be qualified with their declaring method in the text to make clear where they are declared. Unlike fields, local variables are not stored together with their qualification. Qualifications are just added for readability: Variable v in method m in class T is written as $T{::}m{::}v$.[9]

This completes the description of the program representation. Remember that it is underspecified for the implementation of a refactoring tool as it does not correspond to the concrete syntax and comments and whitespace information is also missing as well as position information that would allow source-code patching.

The program representation consists of partial functions and named tuples. Named tuples are also partial functions. Therefore, the program representation is a tree of partial functions (figure 3.4). Statements can be nested, e.g., sequential composition of statements. See figure 3.3 for an example.

[9]The language does not have local scopes in method bodies.

$$Prog = Name \hookrightarrow Decl \tag{3.3}$$

$$\underbrace{Decl}_{D} = \underbrace{(Name)\,option}_{\text{superclass}(D)} \times \underbrace{(Name)\,list}_{\text{superifaces}(D)} \times \underbrace{Fields}_{\text{fields}(D)} \times \underbrace{Methods}_{\text{methods}(D)} \times \underbrace{StaticInitializer}_{\text{staticinitializer}(D)} \tag{3.4}$$

$$Fields = Name \hookrightarrow \overbrace{(\underbrace{FieldName}_{\text{fldname}(f)} \times \underbrace{TypeTag}_{\text{tytag}(f)})}^{f} \tag{3.5}$$

$$Methods = MethodSignature \hookrightarrow \overbrace{(ParameterName \times TypeTag)\,list}^{p=\text{params}(m)=m(\text{params})} \tag{3.6}$$

$$\times \underbrace{TypeTag}_{\text{retT}(m)} \tag{3.7}$$

$$\times \underbrace{(Statement)\,option}_{\text{body}(m)} \tag{3.8}$$

$$MethodSignature = \underbrace{Name}_{\text{mname}(m)} \times \underbrace{(TypeTag)\,list}_{\text{paramTs}(m)} \tag{3.9}$$

$$TypeTag = BasicType + ClassName + TypeTag[] \tag{3.10}$$

$$\tag{3.11}$$

Figure 3.2.: Program structure and accessor names (excluding statements)

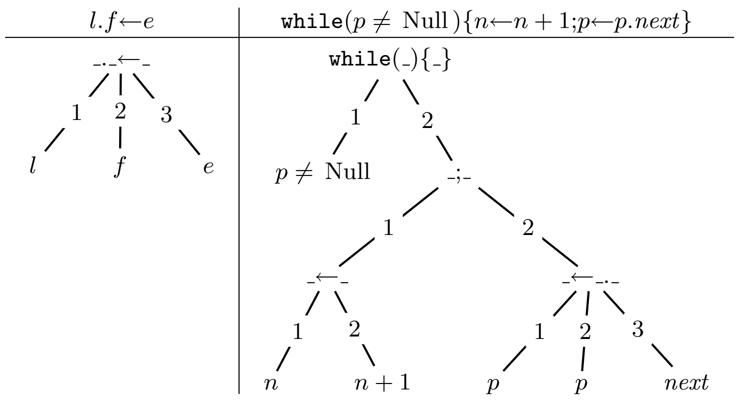


Figure 3.3.: Two examples of statement trees

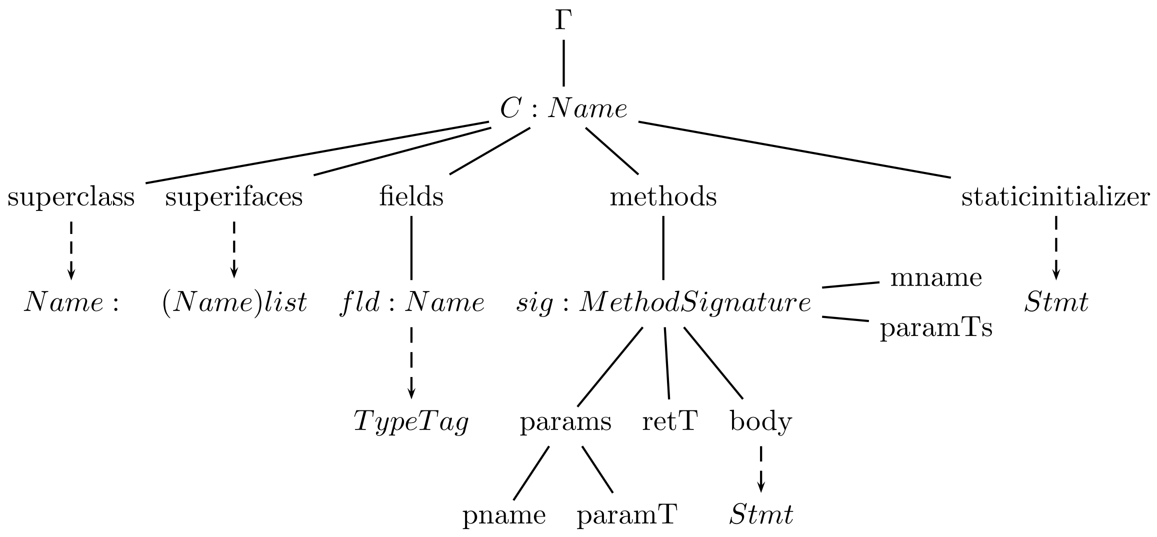


Figure 3.4.: Tree representation of the program

The whole program representation makes it easy to navigate to individual nodes in the tree. Let the program be Γ. $\Gamma(C)$ yields the class declaration of class C. $\Gamma(C)(\text{fields})$ yields the fields in the class and $\Gamma(C)(\text{fields})(f)$ yields the type of the field f in the class. The fact that this is more conventionally written as $\Gamma.\text{fields}.f$ and the fact that refactorings are very often confined to indivividual sections in the source-code suggests a concise notation for defining conditions on programs as well as transformations on them.

3.1.2. Conditions on programs

Path

A path in a program tree is a list $p_1.p_2.\cdots.p_n$ of accessors p_i. If given together with the root of the tree, a path can identify a program part.[10] In the context of a program Γ as the root, the path $p \equiv p_1.p_2.\cdots.p_n$ identifies $\Gamma(p_1)(p_2)\cdots(p_n)$ if defined. $\Gamma[p]$ can be written instead. The notation *root*$[e]$ for any expression e is then defined by the values of the paths in the expression

$$\mathit{root}[e] \equiv e[\mathit{root}[e]/q \text{ for all paths } q \text{ that occur in } e]$$

Example 3.1. Let e be `3.1+7.8`, i.e.,

```
     +
    / \
   1   2
  /     \
3.1     7.8
```

$e[2 \mod 1]$ is then $e[2] \mod e[1] = 7.8 \mod 3.1 = 1.6$.

bles

Free variables in the path lead to paths that do not identify subtrees uniquely. Instead, such expressions are interpreted as sets of subtrees. Example: $\Gamma[C]$ is the set of all class and interface declarations in Γ if C is free

Expressions with free variables

Expressions over path variables are defined as set of all evaluations of the expression in the elements of the cartesian product of the sets the path variables generate. I.e., Let v_1 to v_m be all the free variables that occur in paths in e. Let $v_1 \in V_1$, $v_2 \in V_2$ etc. Let $B = V_1 \times \cdots \times V_m$. $\Gamma[e]$ is then defined as

$$\{\overbrace{\Gamma[e[b_i/v_i \text{ for all } i \in 1..m]]}^{e'} \,|b \in B \wedge e' \text{ defined}\}$$

Example 3.2. $\Gamma[A.\text{fields}.\texttt{f} \times B.\text{fields}.\texttt{g}]$ for the program Γ below yields

$$\{\texttt{int} \times \texttt{long}, \texttt{long} \times \texttt{int}, \texttt{int} \times \texttt{int}, \texttt{long} \times \texttt{long}\}$$

```
class Y{
  int f, g;
}
class Z{
  long f, g;
}
```

$\Gamma[\text{nodecount}(e)]$ denotes the number of values e generates in Γ, i.e., $\Gamma[\text{nodecount}(e)] = |\Gamma[e]|$

Predicates

Predicates are to be interpreted as existential. $\Gamma[P]$ is considered true whenever true $\in \Gamma[P]$.

Star $*$

The star $*$ is used for any path: $\Gamma[*.\texttt{while}(e)\{S\}]$ identifies all while loops in the program Γ that have condition e and loop body S.

3.1.3. Transformations on programs

The simple notation for program transformation is

$$\mathit{root}[\text{selector}_1 := \text{replacement}_1, \ldots, \text{selector}_n := \text{replacement}_n]$$

The idea is that the part of the tree identified by "selector" is replaced by the tree for the expression "replacement". This does not create problems as long as the modified

[10]It actually *does* identify a program part unless there is no program part that corresponds to the path.

parts are disjoint. Then, the expression is equivalent to sequential replacements. This will always be the case for substitutions in this text:

$$\mathit{root}[\text{selector}_1 := \text{replacement}_1] \ldots [\text{selector}_n := \text{replacement}_n]$$

Updates can also be written as sets: $\Gamma[U]$ is $\Gamma[u_1, \cdots, u_m]$ if $U = \{u_i | i \in 1..m\}$.

The path-expression or selector $f_1.f_2.\cdots.f_n$ refers to the subtree $\mathit{root}(f_1)(f_2)\cdots(f_n)$ as stated above. For the selector $f_1.f_2.\cdots.f_n$, the replacement expression "·" refers to $\mathit{root}(f_1)(f_2)\cdots(f_n)$, "··" refers to $\mathit{root}(f_1)(f_2)\cdots(f_{n-1})$. Free variables can again serve as patterns for replacements: $\Gamma[C.\text{fields}.\texttt{foo} := \mathit{undef}]$ deletes all fields `foo` from all classes. $*$ matches any path in the tree. $\Gamma[*.\texttt{while}(e)\{S\} := \texttt{skip}]$ replaces all while-loops in the program by `skip`.

Formally, $\mathit{root}[f_1.f_2.\cdots.f_n := e]$ is the same as

$$\mathit{root}[f_1 \mapsto \mathit{root}(f_1)[f_2 \mapsto \mathit{root}(f_1)(f_2)[f_3 \mapsto \cdots \mathit{root}(f_1)(f_2)\cdots(f_{n-1})[f_n \mapsto \mathit{root}[e]]]\cdots]$$

Replacements may be guarded: $\mathit{root}[\text{if } \mathit{cond} \text{ then } \mathit{updates}]$ is $\mathit{root}[\mathit{updates}]$ only if *cond* is satisfied. *cond* may also contain path expressions. *cond* will then bind some of the free variables in *updates*.

Updates

I.e., Let v_1 to v_m be all the free variables that occur in paths in if *cond* then *updates*. Let $v_1 \in V_1$, $v_2 \in V_2$ etc. Let $B = V_1 \times \cdots \times V_m$. $\Gamma[\text{if } \mathit{cond} \text{ then } \mathit{updates}]$ is then defined as

$$\Gamma[\{\overbrace{\mathit{updates}[b_i/v_i \text{ for all } i \in 1..m]}^{u'} | b \in B \wedge u' \text{ defined} \wedge \mathit{cond}'\}]$$
$$\text{where } \mathit{cond}' \equiv \mathit{cond}[b_i/v_i \text{ for all } i \in 1..m]$$

The notation introduced – both for expressions and transformations – here is similar to XQuery, the "tree parser" syntax in ANTLR as well as other tree matcher generators like the ones of the BURG family (including iburg).

The concept is similar to ECMAScript [14]. This language presents objects as partial functions just in the same way we model the program representation. It has a **`with`** statement that establishes the default root object for other statements just as our notation here. **`with(`**Γ**`){`**U**`}`** roughly corresponds to $\Gamma[U]$.

Refactorings are often concerned with very selective updates that do not fit too well with the syntax used in mainstream functional implementation languages.

The notation is also[11] similar to pointcut designators in AspectJ (see figure 3.1). This is by no means a coincidence. Aspect oriented programming is not much more than local code instrumentation that is specified outside the instrumentation site. It suggests that

[11]This just shows that people come up with the same concepts and ideas over and over again.

postconditions could be added to the application as aspects if a technology or language like AspectJ is used. The lack of visibility that normally causes legitimate concerns is not a problem when only postconditions are formulated as aspects: they do not alter the programs' functioning.

```
// Pointcut designators are a different syntax for pre
   -/postconditions
aspect Foo {
  declare error : set *. && withincode *.get*(..))  :
    "side effect in getter!";
```

Listing 3.1: Designators in AspectJ (from [47])

3.1.4. μ: Refactoring transformations

As explained in the previous section, updates U for the program Γ are written as

$$\Gamma[U]$$

Such updates uniquely identify a transformation function on programs

$$\Gamma \mapsto \Gamma[U]$$

If this transformations is a refactoring, it is named μ. All refactorings are refered to as μ (possibly with some sub- and superscripts) and given as a set of updates on Γ.

3.2. The Notion of equivalence and the operational semantics

Summary. The operational semantics represents a state s as a tuple $s = (\text{xcpt}, \sigma, \gamma, \mathfrak{u})$ of exception state xcpt, local variable and parameter state σ, heap γ and the sequence of input/output operations $\mathfrak{u}$ and an entity that denotes the initial system state and decisions determining the outcome of I/O operations written as $\mathfrak{u}_0$. $\mathfrak{u}_0$ is never written as part of the state because the input decisions are the same for all runs of all versions of all programs observed. The sequence of I/O operations $\mathfrak{u}$ is sometimes refered to as "world". The big-step transition relation is written as $\Gamma \vdash s \xrightarrow{t} s'$ for a program Γ, an initial state s a path t in Γ identifying the executed statement and a terminal state s'. Two programs are considered equivalent if they perform the same sequence of I/O operations on the same $\mathfrak{u}_0$ states for all comparable program parts. At least the whole programs must be comparable.

This section provides an overview over all statements that are valid in the subset of Java I cover in this research. A more detailed explanation is given in appendix C. In the appendix, I also discuss differences between the semantics in the present text and the semantics by Oheimb [44] on which it is based.

The section also derives the notion of equivalence in detail and elaborates on the necessitities and trade-offs.

The statements are described by a big-step operational semantics. The operational semantics is tailored towards its intended use: to prove the externally visible behavioral equivalence of statements. This makes it necessary to model externally visible actions explicitly. Most formal semantics deal only with the state updates "inside" the programming environment and probably with some functions that handle aspects of the language that have an impact on the execution itself, like threading libraries. The semantics are normally geared towards proving correctness properties inside the program. If this is the case, it can be quite reasonably assumed that I/O (or general external) functions do not have any effect on the program state. They can be assumed to just return any value without affecting the program's internal state.

With refactorings, it is different. It *does* matter if `putchar('B')` is called once or twice because if it is called twice, there will be two "B"s on the console instead of one even if both calls return `'B'`! As I explain in the introduction and repeat throughout the book, this leads to the following straightforward definition:

Observably equivalent behavior

Refactorings in this research are considered correct if the program after the refactoring yields the same output, i.e., perform the same I/O operations in the same order, as before the refactoring. It is the definition first formulated by William Opdyke.

For a formalization, this vague definition has to be made more precise: It is certainly not possible to take all the possibilities into account how a certain output can be produced if the framework is to be easily extensible to different kinds of I/O. The treatment of "output" should not depend on the concrete semantics of the I/O routines. Example: The fact that `printf("AB");` and `putchar('A'); putchar('B');` produce the same output on the console in C/C++ under normal conditions is a fact that cannot be derived without knowing about the semantics inherent to `printf` and `putchar`.[12] If this opaque view on I/O operations is accepted, it is clear that externally visibly behavioral equivalence requires two programs to perform the same sequence of I/O operations.

[12]Some of the reasons for this restriction are these: (i) The number of different I/O operations is unreasonably large. (ii) The number of I/O abstractions is enormous: consider disks, tapes, flash memory, windowing, printers, etc. (iii) I/O functions are difficult to specify (iv) and normally remain un- or underspecified. (v) If it matters how I/O operates, differences in behavior may also be possible.

3.2.1. State

The state as modelled in operational semantics should remember the sequence of I/O operations. I/O operations need not be deterministic. The list of I/O operations plus some initial configuration are refered to as $\mathfrak{u} = (a_1^{I/O}, \cdots, a_{|\mathfrak{u}|}^{I/O})$. $A^{I/O}$ denotes all possible I/O operations. The "initial state" of the system that determines the results of subsequent I/O operations is refered to as $\mathfrak{u}_0 \in \mathfrak{U}$. It encapsulates all the possible paths the interaction could lead to, which might be infinite if the program loops.

I find the name "I/O operations" misleading. Even though I/O encompasses all external operations, the name "input/output" suggests that it may be confined to actual input and actual (viewable, readable or otherwise usable output). I may therefore call $\mathfrak{u}$ the "world" instead.

Other aspects of the machine state

The state in the operational semantics does also have to model other more conventional aspects of the machine that can be found in any operational semantics.

In addition to the world, the state (denoted by s, r, etc.) consists of the currently active exception (xcpt), the values of local variables and parameters (σ) and the globals (γ).

The whole machine state for the big-step operational semantics thus consists of: exception, locals, globals and the world. This is a tuple.

$$\begin{array}{rll} \mathit{State} \equiv & (& \\ & \text{xcpt} & : (\mathit{ExceptionLocation})\,\mathit{option}, \\ & \sigma & : \mathit{Locals}, \\ & \gamma & : \mathit{Heap}, \\ & \mathfrak{u} & : \mathfrak{U} \times (A^{I/O})\,\mathit{list}, \\ ) & & \end{array} \tag{3.12}$$

The locals map the names of the local variables to their value:

$$\mathit{Locals} \equiv \mathit{Name} \hookrightarrow \mathit{Value}$$

The *Heap* contains all the global data in the program. This includes static fields as well as dynamically allocated data (data in the object store). Object store data can be objects and arrays. Java supports dynamic type tests so it is important that the *TypeTag* also gets stored in the heap. A *HeapValue* consists of type tag and the proper field values.

$$\begin{array}{rll} \mathit{HeapValue} \equiv & (& \\ & \text{rtt} & : \mathit{TypeTag}, \\ & \text{refvalues} & : \mathit{FieldValues} + \mathit{ArrayValues} \\ ) & & \end{array} \tag{3.13}$$

The accessor refvalues is omitted if it is clear from the context. Field values map field names to their value and array values map non-negative integers within a certain range to the value of the corresponding element. From their respective formal treatment, arrays and objects are identical if the kind of index element is abstracted away as *ObjectIndex*.

$$\begin{aligned} \mathit{ArrayValues} &\equiv \mathbb{N} \hookrightarrow \mathit{Value} \\ \mathit{FieldValues} &\equiv \mathit{FieldName} \hookrightarrow \mathit{Value} \\ \mathit{ObjectIndex} &\equiv \mathbb{N} + \mathit{FieldName} \end{aligned}$$

The heap now maps locations to heap values and the special reference Null to a special value that allow to unify Null and *Location* for rtt.

$$\mathit{Heap} \equiv (\mathit{Location} \hookrightarrow \mathit{HeapValue}) \cup \{\{\mathrm{Null} \mapsto (\mathrm{NullType}, \mathrm{Null})\}\} \tag{3.14}$$

*Location*s can be addresses returned from **new** but also the names of classes. They point to the class object.

Example 3.3. Have a look at the following class definition and the subsequent statements.

```
class A{
  static int x, y;
  static A someobj;
  int f;
}
A.x = 1;
A.y = 2;
A.someobj = new A();
A.someobj.f = 3;
```

The heap will be the following function afterward execution of the statements (**2134132** is an arbitrary valid location):

$$\begin{aligned} \gamma = \{ & \\ & A \quad\quad\quad \mapsto (\mathrm{metaclass}(A), \{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 2, \mathtt{someobj} \mapsto \mathbf{2134132}\}) \\ & \mathbf{2134132} \quad \mapsto (\mathtt{A}, \{\mathtt{f} \mapsto 3\}) \\ \} & \end{aligned}$$

Example 3.4. The following complete program illustrates all aspects of the state: local variables, parameters, the heap, I/O and exceptions.

```
class A{
    A next;
    int n;
}

class Main{
    public static void main(String[] args) throws
       Exception{
        A a = null, b = null;
        // build structure
        for(int i = 0; i < 5; i++){
            b = new A(); b.next = a;
            b.n = i;
            a = b;
            System.out.println("a␣=␣" + a.id + "␣a.n␣=
               ␣" + a.n);
        }

        for(;a.next != null; a = a.next)
            ;
        a.next = b;

        for(; a != null; a = a.next){
            a.n = a.n * 5555;
            if(a.n < 0) throw new Exception();
            out.println(a.n);
        }
    }
}
```

The state s at the end of the program is the following. Locations are again just integer numbers for the sake of simplicity:

$$
\begin{array}{lll}
s.\gamma = \{ & & \\
 & A & \mapsto (\mathrm{metaclass}(A), \{\}) \\
 & \mathbf{1} & \mapsto (\mathtt{A}, \{\mathtt{next} \mapsto 5, \mathtt{n} \mapsto 0\}) \\
 & \mathbf{2} & \mapsto (\mathtt{A}, \{\mathtt{next} \mapsto 1, \mathtt{n} \mapsto 30858025\}) \\
 & \mathbf{3} & \mapsto (\mathtt{A}, \{\mathtt{next} \mapsto 2, \mathtt{n} \mapsto 61716050\}) \\
 & \mathbf{4} & \mapsto (\mathtt{A}, \{\mathtt{next} \mapsto 3, \mathtt{n} \mapsto 92574075\}) \\
 & \mathbf{5} & \mapsto (\mathtt{A}, \{\mathtt{next} \mapsto 4, \mathtt{n} \mapsto -1529451860\}) \\
 & \mathbf{33} & \mapsto (\mathtt{RuntimeException}, \{\}) \\
\} & & \\
s.\mathrm{xcpt} = & \mathbf{33} & \\
s.\sigma = \{ & & \\
 & \mathsf{a} & \mapsto \mathbf{5} \\
 & \mathsf{b} & \mapsto \mathbf{5} \\
\} & & \\
s.\mathfrak{u} = [& & \\
 & & \mathtt{System.out.println}(0) \\
 & & \mathtt{System.out.println}(22220) \\
 & & \mathtt{System.out.println}(16665) \\
 & & \mathtt{System.out.println}(11110) \\
 & & \mathtt{System.out.println}(5555) \\
 & & \mathtt{System.out.println}(0) \\
 & & \mathtt{System.out.println}(123432100) \\
 & & \mathtt{System.out.println}(92574075) \\
 & & \mathtt{System.out.println}(61716050) \\
 & & \mathtt{System.out.println}(30858025) \\
 & & \mathtt{System.out.println}(0) \\
] & &
\end{array}
$$

$\mathfrak{u}_0$ is omitted.

Example 3.5. Now consider the following program that does input as well as output and prints the character read from the console.

```
System.out.print((char)System.in.read());
```

Assume I enter “f” on the console. The resulting state is then

$$
\begin{array}{lll}
s.\gamma = & \{\} & \\
s.\mathrm{xcpt} = & \mathrm{None} & \\
s.\sigma = & \{\} & \\
s.\mathfrak{u} = [& & \\
 & & \mathtt{System.in.read}() \Rightarrow \mathtt{'f'} \\
 & & \mathtt{System.out.print}(\mathtt{'f'}) \\
] & &
\end{array}
$$

Imagine I decide to enter “f” on the console only if there hasn’t been any I/O activity before in the program. One possible representation of $\mathfrak{u}_0$ could be a function from I/O and method calls to return values: $s.\,\mathfrak{u}_0 = \{([], \texttt{System.in.read}()) \mapsto \texttt{'f'}\}$. The I/O operations have to interpret $\mathfrak{u}_0$ correctly. Refactorings that are unrelated to input-output such as the one in this thesis will be correct if they do not make any assumptions about its representation. Therefore, I do not want to commit myself to any particular representation of $\mathfrak{u}$ and $\mathfrak{u}_0$.

3.2.2. The transition relation

The big-step transition relation is written as $\Gamma \vdash s \xrightarrow{t} s'$ where t is a path in Γ. $t = C.\,\mathrm{methods}\,.g.\,\mathrm{body}\,.2$, for instance, identifies the second subtree of the statement that constitutes the implementation of method g in class C in the program. The notation $\Gamma \vdash s \xrightarrow{t} s'$ means that the execution of the statement identified by t leads from some initial state s to some terminal state t'. Instead of “the statement identified by t in Γ”, I just write *statement* t even though t is still only meaningful in the context of a program.

As customary for operational semantics, the transition relation is specified for each kind of statement individually. The kinds of statements are the following.

`skip` does nothing

$t_1;t_2$ executes t_1 and then t_2

`while`$(e)\{t\}$ executes t if and as long e is true

`if`$(e)\{t_1\}$`else`$\{t_2\}$ executes t_1 if e is true and t_2 otherwise

$l\leftarrow$`new` C allocates an object on the heap and stores its location in the local variable l

`throw` e throws the exception identified by e

`try`$\{t_1\}$`finally`$\{t_2\}$ executes t_1 and then t_2 even if t_1 causes an exception

`try`$\{t_1\}$`catch`$(C\ l)\{t_2\}$ executes t_1 and then t_2 if t_1 causes an exception of type C with l bound to the location of the exception caused

$l\leftarrow e$ sets the local variable l to the value of e

$l\leftarrow(T)e$ sets the local variable l to the value of e or causes an exception if the runtime type of the object identified by e is not a subtype of T

$l.f\leftarrow e$ Sets the value of field f in l to e

$l_1\leftarrow l_2.f$ Retrieves the value of f in l_2 and stores the result in l_1.

$l[i_e]\ \leftarrow e$ sets element i_e of l to e

$l_1\leftarrow l_2[i_e]$ sets l_1 to element i_e of l_2

`init_class` C initializes the class C

$l_1 \leftarrow l_2.T{::}m(e)$ invokes an implementation of method m in T or the next superclass that implements it on object l_2 with parameter vector e and stores the result of the invocation in l_1. The method is statically bound.

$l_1 \leftarrow l_2.m(e)$ invokes the most specific implementation of method m for l_2 on l_2 with parameter vector e and stores the result of the invocation in l_1. The method is dynamically bound.

Evaluation of expressions

The evaluation of expressions $[\![e]\!]^s_\Gamma$ is defined as usual and can basically be reduced to the formula

$$[\![f(e_1, \cdots, e_n)]\!]^s_\Gamma = [\![f]\!]^s_\Gamma([\![e_1]\!]^s_\Gamma, \cdots, [\![e_n]\!]^s_\Gamma)$$

$[\![f]\!]^s_\Gamma$ is the function that corresponds to function symbol f. Values like integers and reals are idealized in the operational semantics and it therefore holds that the operator symbol corresponds to the proper operator over the corresponding set. Example: $[\![\mathtt{x}\cdot\mathtt{y}]\!]^s_\Gamma = s.\sigma(\mathtt{x})\cdot s.\sigma(\mathtt{y})$.

The formalization of I/O statements

I/O is performed by the program using special I/O "instructions". These I/O instructions are abstracted as ext-body. The execution of an ext-body-statement does not change the state space of the program except for the sequence of I/O operations, i.e., if the initial state is s_0, the terminal state will be $s_0[\mathfrak{u} := \mathfrak{u}']$ for some new world $\mathfrak{u}'$. Moreover, the behavior of such a statement will only depend on the local variables and the objects reachable through them and static fields. The objects reachable from a set of root values σ in a heap γ is denoted by $\gamma^{\sigma+}$. The possible updated sequence of I/O statements $\mathfrak{u}'$ is then determined by the relation *Ext* alone: $\mathit{Ext}(\sigma, \gamma^{\sigma+}, \mathfrak{u}, \mathfrak{u}')$

As its name implies, the ext-body statement only occurs as the body of a method. There is no specific reason for this decision except that it makes it easier to think about the effect of external methods as the set of local variables used in a method is not limited except at the beginning of a routine where only the parameters are set.

As can be eaily seen from the semantics, only external ext-body changes the $\mathfrak{u}$ part of the state. All other instructions leave it invariant.

Typegraphic conventions for variable names

I do not use the same font for all concrete variables. A variable that is represented by the *letter* x can either be written as X (or `x`, which is the same) or as x. The difference is merely that `x` represents a *concrete* variable that has the name `x` and x can represent any concrete variable irrespective of what its name is, i.e., concrete variable x can be called `x` in the source code but it might just as well be called `y`, `z` or anything else.

Java features not covered here

This informal description illustrates the subset of features Java that are investigated for this thesis. Omitted are features that would complicate the semantics but wouldn't add to the validity of the reasoning done here. These include: expressions with side effects (assignment statements and method calls), object access inside expressions (field reads), abrupt termination with **`return`**, **`break`** or **`continue`**. Methods are assumed to return the value of the special variable **`result`**.

Not covering abrupt termination is problematic for some investigations. It is unproblematic for this research because atomic statements that affect the state space directly are more likely to break equivalence than composite statements that affect control flow. Switching to a language with completely unrestricted control flow like the one presented in [2] for instance would be easy. The operational semantics for the statements is summarized in the tables 3.2 to 3.5. They use the two abbreviations for raising new exceptions $raise(s, E)$ of type E in state s and allocating objects $alloc(l, s, X)$ of type X in state s, storing the result in l, a local variable.

$$\begin{aligned} raise(s,E) &= s[\text{xcpt} := loc, \gamma := \gamma[loc \mapsto \text{init\_obj}(\Gamma, E)]] \text{ where } s.\,\gamma(loc) = \text{None} \\ alloc(l,s,X) &= s[\sigma.l := loc, \gamma := \gamma[loc \mapsto X]] \text{ where } s.\,\gamma(loc) = \text{None} \end{aligned}$$

Figure 3.5.: Abbreviations for exception handling

Statement	Poststate	Necessary conditions ($\wedge\ s_0.\,\text{xcpt} = \text{None}$)
`skip`	s_0	–
t_1;t_2	s_2	$\Gamma \vdash s_0 \xrightarrow{t_1} s_1 \wedge \Gamma \vdash s_1 \xrightarrow{t_2} s_2$
`while`(e){t}	s_0	$\llbracket e \rrbracket^{s_0}_{\Gamma} = \text{false}$
`while`(e){t}	s_2	$\llbracket e \rrbracket^{s_0}_{\Gamma} = \text{true} \wedge \Gamma \vdash s_0 \xrightarrow{t} s_1 \wedge \Gamma \vdash s_1 \xrightarrow{\texttt{while}(e)\{t\}} s_2$
`if`(e){t_1}`else`{t_2}	s_2	$\llbracket e \rrbracket^{s_0}_{\Gamma} = \text{true} \wedge \Gamma \vdash s_0 \xrightarrow{t_1} s_2$
`if`(e){t_1}`else`{t_2}	s_2	$\llbracket e \rrbracket^{s_0}_{\Gamma} = \text{false} \wedge \Gamma \vdash s_0 \xrightarrow{t_2} s_2$

Table 3.2.: Basic statements

Statement	Poststate	Necessary conditions ($\wedge\ s_0.\,\text{xcpt} = \text{None}$ unless stated)
t	s_0	$s.\,\text{xcpt} \neq \text{None}$
`throw` e	$s_0[\text{xcpt} := x]$	$x = \llbracket e \rrbracket^{s_0}_{\Gamma} \neq \text{Null}$
`throw` e	$raise(s_0, \texttt{NullPointer})$	$x = \llbracket e \rrbracket^{s_0}_{\Gamma} = \text{Null}$
`try`{t_1}`finally`{t_2}	$s_2[\text{xcpt} := s_1.\,\text{xcpt}]$	$\Gamma \vdash s_0 \xrightarrow{t_1} s_1 \wedge \Gamma \vdash s_1[\text{xcpt} := \text{None}] \xrightarrow{t_2} s_2 \wedge s_2.\,\text{xcpt} = \text{None}$
`try`{t_1}`finally`{t_2}	s_2	$\Gamma \vdash s_0 \xrightarrow{t_1} s_1 \wedge \Gamma \vdash s_1[\text{xcpt} := \text{None}] \xrightarrow{t_2} s_2 \wedge s_2.\,\text{xcpt} \neq \text{None}$
`try`{t_1}`catch`(C l){t_2}	s_2	$\Gamma \vdash s_0 \xrightarrow{t_1} s_2 \wedge s_2.\,\text{xcpt} = \text{None}$
`try`{t_1}`catch`(C l){t_2}	s_2	$\Gamma \vdash s_0 \xrightarrow{t_1} s_1 \wedge \text{rtt}_{s_1}(s_1.\,\text{xcpt}) \preceq_{\Gamma} C \wedge \Gamma \vdash s_1[\text{xcpt} := \text{None}, \sigma.l := x] \xrightarrow{t_2} s_2$
`try`{t_1}`catch`(C l){t_2}	s_1	$\Gamma \vdash s_0 \xrightarrow{t_1} s_1 \wedge \text{rtt}_{s_1}(s_1.\,\text{xcpt}) \not\preceq_{\Gamma} C$

Table 3.3.: Exception statements

Statement	Poststate	Necessary conditions ($\wedge\, s_0.\,\mathrm{xcpt} = \mathrm{None}$)
$l{\leftarrow}\mathtt{new}\ C$	$alloc(l, s, \mathrm{init\_obj}(\Gamma, C))$	–
$l{\leftarrow}\mathtt{new}\ C[e]$	$alloc(l, s, \mathrm{init\_obj}(\Gamma, (C, N)))$	$N = \llbracket e \rrbracket_{\Gamma}^{s_0} \wedge N \geq 0$
$l{\leftarrow}\mathtt{new}\ C[e]$	$raise(s_0, \mathtt{NegArrSize})$	$N = \llbracket e \rrbracket_{\Gamma}^{s_0} \wedge N < 0$
$l{\leftarrow}e$	$s_0[\sigma.l := \llbracket e \rrbracket_{\Gamma}^{s_0}]$	–
$l{\leftarrow}(T)e$	$s_0[\sigma.l := x]$	$x = \llbracket e \rrbracket_{\Gamma}^{s_0} \wedge X = \mathrm{rtt}_{s_0}(x) \wedge X {\preceq_\Gamma} T$
$l{\leftarrow}(T)e$	$raise(s_0, \mathtt{ClassCast})$	$x = \llbracket e \rrbracket_{\Gamma}^{s_0} \wedge X = \mathrm{rtt}(\gamma(x)) \wedge X \not\preceq_\Gamma T$
$l.f{\leftarrow}e$	$s_0[\gamma\,.x.f := \llbracket e \rrbracket_{\Gamma}^{s_0}]$	$x = \llbracket l \rrbracket_{\Gamma}^{s_0} \wedge x \neq \mathrm{Null}$
$l.f{\leftarrow}e$	$raise(s_0, \mathtt{NullPointer})$	$x = \llbracket l \rrbracket_{\Gamma}^{s_0} \wedge x = \mathrm{Null}$
$l_1{\leftarrow}l_2.f$	$s_0[\sigma.l_1 := \gamma\,.x.f]$	$x = \llbracket l_2 \rrbracket_{\Gamma}^{s_0} \neq \mathrm{Null}$
$l_1{\leftarrow}l_2.f$	$raise(s_0\mathtt{NullPointer})$	$x = \llbracket l_2 \rrbracket_{\Gamma}^{s_0} = \mathrm{Null}$
$l[i_e]\ {\leftarrow}e$	$raise(s_0, \mathtt{NullPointer})$	$x = \llbracket l \rrbracket_{\Gamma}^{s_0} = \mathrm{Null}$
$l[i_e]\ {\leftarrow}e$	$raise(s_0, \mathtt{IndOutBound})$	$x = \llbracket l \rrbracket_{\Gamma}^{s_0} \neq \mathrm{Null} \wedge ((T, N), a) = s_0.\,\gamma(x) \wedge i = \llbracket i_e \rrbracket_{\Gamma}^{s_0} \notin [0, N)$
$l[i_e]\ {\leftarrow}e$	$raise(s_0, \mathtt{ArrStore})$	$x = \llbracket l \rrbracket_{\Gamma}^{s_0} \neq \mathrm{Null} \wedge ((T, N), a) = s_0.\,\gamma(x) \wedge i = \llbracket i_e \rrbracket_{\Gamma}^{s_0} \in [0, N) \wedge v = \llbracket e \rrbracket_{\Gamma}^{s_0} \wedge \mathrm{rtt}_{s_0}(v) \not\preceq_\Gamma T$
$l[i_e]\ {\leftarrow}e$	$s_0[\gamma\,.x.i := v]$	$x = \llbracket l \rrbracket_{\Gamma}^{s_0} \neq \mathrm{Null} \wedge ((T, N), a) = s_0.\,\gamma(x) \wedge i = \llbracket i_e \rrbracket_{\Gamma}^{s_0} \in [0, N) \wedge v = \llbracket e \rrbracket_{\Gamma}^{s_0} \wedge \mathrm{rtt}_{s_0}(v) {\preceq_\Gamma} T$
$l_1{\leftarrow}l_2[i_e]$	$raise(s_0, \mathtt{NullPointer})$	$x = \llbracket l_2 \rrbracket_{\Gamma}^{s_0} = \mathrm{Null}$
$l_1{\leftarrow}l_2[i_e]$	$raise(s_0, \mathtt{IndOutBound})$	$x = \llbracket l_2 \rrbracket_{\Gamma}^{s_0} \neq \mathrm{Null} \wedge ((T, N), a) = s_0.\,\gamma(x) \wedge i = \llbracket i_e \rrbracket_{\Gamma}^{s_0} \notin [i, N)$
$l_1{\leftarrow}l_2[i_e]$	$s_0[\sigma.l_1 := a(i)]$	$x = \llbracket l_2 \rrbracket_{\Gamma}^{s_0} \wedge x \neq \mathrm{Null} \wedge ((T, N), a) = \gamma(x) \wedge i = \llbracket i_e \rrbracket_{\Gamma}^{s_0} \in [0, N)$

Table 3.4.: Statements for object manipulation

Statement	Poststate	Necessary conditions ($\wedge\, s_0.\,\mathrm{xcpt} = \mathrm{None}$)
$\mathtt{init\_class}\ C$	$s_0[\mathrm{xcpt} := x, \gamma := s_2.\,\gamma])$	$\gamma(C) = \mathrm{None} \wedge \gamma' = s_0.\,\gamma[C \mapsto \mathrm{init\_obj}(\Gamma, \mathrm{metaclass}(C))] \wedge \Gamma \vdash s_0[\sigma := \{\mathtt{this} \mapsto C\}] \xrightarrow{\mathrm{staticinitializer}(\Gamma(C))} s_2$
$\mathtt{init\_class}\ C$	s_0	$s_0.\,\gamma(C) \neq \mathrm{None}$
$l_1{\leftarrow}l_2.T{::}m(e)$	$raise(s_0, \mathtt{NullPointer})$	$x = \llbracket l_2 \rrbracket_{\Gamma}^{s_0} = \mathrm{Null}$
$l_1{\leftarrow}l_2.T{::}m(e)$	$s_1[\sigma := \sigma[l_1 \mapsto \sigma'(\mathtt{result})]]$	$x = \llbracket l_2 \rrbracket_{\Gamma}^{s_0} \neq \mathrm{Null} \wedge \Gamma \vdash s_0[\sigma := \{\mathtt{this} \mapsto x, \mathrm{pNames}(\Gamma, T, m) \mapsto e\}] \xrightarrow{\mathrm{body}(\Gamma, T, m)} s_1$
$l_1{\leftarrow}l_2.m(e)$	$raise(s_0, \mathtt{NullPointer})$	$x = \llbracket l_2 \rrbracket_{\Gamma}^{s_0} = \mathrm{Null}$
$l_1{\leftarrow}l_2.m(e)$	$s_1[\sigma := s_0.\sigma[l_1 \mapsto s_1.\sigma(\mathtt{result})]]$	$x = \llbracket l_2 \rrbracket_{\Gamma}^{s_0} \neq \mathrm{Null} \wedge T = \mathrm{rtt}(x) \wedge \Gamma \vdash s_0[\sigma := \{\mathtt{this} \mapsto x, \mathrm{pNames}(\Gamma, T, m) \mapsto e\}] \xrightarrow{\mathrm{body}(\Gamma, T, m)} s_1$
ext-body$_m$	$s_0[\mathfrak{u} := \mathfrak{u}']$	method m has ext-body$_m$ as its body and $Ext(s_0.\sigma, s_0.\,\gamma^{s_0.\sigma+}, s_0.\,\mathfrak{u}, \mathfrak{u}')$

Table 3.5.: Invocation statements

3.2.3. State and execution correspondence

This section starts with two examples that make it clear that formalizing refactorings needs two additional concepts: state correspondence and execution correspondence. These concepts help define equivalence. As their names imply, state correspondence relates the internal states in the original and the refactored program and execution correspondence captures the intuitive[13] requirement that it is not arbitrary which parts of the programs do what in the original program and the refactored program. Indeed, there is one state correspondence that is particularly obvious and necessary: the correspondence of the I/O statements performed. The I/O statements have to be the same, whereas other parts of the state can be in any relation to each other.

Consider the following two equivalent programs:

```
for(int i = 0; i < 10; i++)  | for(int i = 1; i < 1000; i *= 2)
    out.println(1 << i);     |     out.println(i);
```

They produce the same output yet they have a different internal state. Refactorings may change the internal state as well – just consider "Extract Method". The two states are not unrelated however. Both states can be used to derive the same output relevant values for the invocation of I/O routines. This concept can be captured in a binary relation that associates corresponding states. This correspondence is called state correspondence and it is denoted by β. It needs to be specified in addition to the *program correspondence* μ for every refactoring as shown in figure 3.6.

State correspondence β

The need for β

Why does it not suffice to state how programs are transformed and let the state correspondence, β be inferred from the code transform itself? The reason is that also parts of programs may be comparable, not only whole programs. Their initial state is not independent of previous program-dependent behavior. Consider the following comparable and equivalent code snippets. The output is the same, provided $2^{\mathtt{i}} = \mathtt{j}$ holds *at the beginning of the execution.*

```
j = j*2;         | i = i+1;
out.println(j);  | out.println(1 << i);
```

The example illustrates another point. Imagine the left hand side code is the original code and the right hand side is the refactored result. What happens if $\mathtt{j} = 0$ at the beginning? According to what I said above, $2^{\mathtt{i}} = 0$ should hold. Unfortunately, this this equation does not have a solution. Should such a correspondence be forbidden? Probably not because at the example above would not be possible in that case. I even reckon that it would render the formalism much less useful. *Initial* data correspondence *is therefore a precondition for equivalence, i.e., something that can be assumed. Terminal data correspondence is a postcondition of the transformation, i.e., it must be ensured*

[13]The requirement is not only intuitively justified as shown below (cf. section 4.3)

by the transformation. This rule could be weakened for the execution of the whole program, but *that* seems overly general. Moreover, it is clear that a data correspondence that is *never* satisfied[14] is not very helpful, even though it could help prove whatever refactoring is desired. Initial data correspondence at execution start is a precondition of the program.

$$\begin{tikzcd}
\Gamma \arrow[rr, "\mu : Prog \to Prog"] & & \Gamma' \\
\Gamma \vdash s \xrightarrow{t} s' & \beta_{[t,t']} \subset State^2 & \Gamma' \vdash r \xrightarrow{t'} r'
\end{tikzcd}$$

Figure 3.6.: Program correspondence μ (the transform) and state correspondence β indexed by the respective positions in the source-code

To understand why *execution correspondence* is needed, consider the following equivalent programs.

```
            out.println(j);  |             out.println(j1);
j = j+1; out.println(j);  | j2 = j1+1; out.println(j2);
j = j+1; out.println(j);  | j3 = j2+1; out.println(j3);
j = j+1; out.println(j);  | j4 = j3+1; out.println(j4);
```

It is obvious that the programs produce the same output. The right hand side is the *single-assignment* form of the left hand side. There is no fixed correspondence between the variables. The correspondence depends on what statement is being executed. At the beginning, $\texttt{j1} = \texttt{j}$ but at the the end, $\texttt{j1} = \texttt{j} - 3$. In other words, the correspondence depends on the statement that is being executed. Moreover, the statements may not even be symmetric as in the example above – to make things more complicated, I could have wrapped the original into one (or two) for-loops. Unfortunately, big-step semantics do not have a single valid program counter. Paths to the respective parts of the programs provide a sufficient approximation to this low-level concept however. Data correspondence can therefore only be asserted if the statements that are executed in the original and the refactored program correspond to each other.

The combined data-execution correspondence is thus a quarternary relation over paths to statements being executed and the respective state spaces. Correspondence in two programs Γ and Γ' of two states s and r and two program positions t and t' is written as $\beta_{[t,t']}(s, r)$.

It is worth noting that this is just another way of recognizing that the "program counter" is effectively also part of the machine state albeit one that is hidden behind the big-step

[14]e.g., $|\texttt{i}| + 1 = -|\texttt{j}|$

operational semantics.[15] The viable option would be a “mixed” operational semantics that combines abstract block structures and a “flat” program counter (i.e., without stack frames) [2]. The thesis would probably be less accessible as such semantics are not (yet!) widely used or taught.

The arguments serve to legitimate the representation I have chosen for the rest of this thesis. They are by no means complete and this chapter does not provide a *derivation* of a “complete” formalism in whatever sense. Stack-frames for instance also play a role in some cases. The abstractions presented here however are sufficient for what I want to illustrate.

3.2.4. Defining EXTERNAL EQUIVALENCE

This section formalizes the discussion in the previous sections and defines equivalence as a *formula* that is to prove for each refactoring. The formal definition is presented first and then explained and related to the discussion.

Two programs Γ and Γ' are considered *externally equivalent* (cf. equation (3.15)) if for all states s, r and s' and all statement paths t and t' the following property holds: if there is an execution of t in Γ with initial state s that results in s', and s and r are corresponding states for the statements t in Γ and t' in Γ', then there is a transition from r to r' of t' in Γ' and s' and r' correspond to each other for the successor statements. All statements have successors. The main routine has a sentinel successor statement.

Initial data correspondence is a program precondition

If m and m' are the respective main routines, there *must* be a possible initial state w in the original and a possible initial state z in the refactored program such that $\beta_{[m,m']}(w,z)$.

$$\begin{aligned} \mathit{ExtEq}_\beta(\Gamma,\Gamma') \equiv{} & \forall t,t',s,s',r : (\Gamma \vdash s \xrightarrow{t} s') \wedge \beta_{[t,t']}(s,r) \\ & \Rightarrow \exists r' \forall t_+ \in \mathrm{succ}_\Gamma(t), t'_+ \in \mathrm{succ}_{\Gamma'}(t') : (\Gamma' \vdash r \xrightarrow{t'} r') \wedge \beta_{[t_+,t'_+]}(s',r') \end{aligned} \tag{3.15}$$

This definition of external equivalence is too general and too cumbersome to apply for practical refactorings. Special cases of the definition that are powerful enough are defined below.

A program transformation μ is a *refactoring* with data correspondence β if it retains external equivalence under the condition that certain preconditions pre and postconditions post are satisfied. Pre- and postconditions are whole program properties: These

Definition *refactoring*

[15] A small step semantics would not have been an alternative: Big-step semantics allow to consider the cumulative effect of a whole block. This is important for refactorings because most blocks will remain unaffected even if they are part of statements that may have been updated by the refactoring.

properties are independent of properties that must hold for statements because of data correspondence β.

$$\mu \text{ is a } \textit{refactoring} \text{ with correspondence } \beta \text{ under pre-/postconditions pre and post} \\ \equiv \text{pre}(\Gamma) \wedge \text{post}(\Gamma') \Rightarrow \mathit{ExtEq}_{\beta}(\Gamma, \mu(\Gamma)) \quad (3.16)$$

$\mathfrak{u}$-equivalence β is only a valid data correspondence if it is the identity on the sequence of I/O operations, i.e.,

$$\beta(s, r) \Rightarrow \mathfrak{u}(s) = \mathfrak{u}(r)$$

It is this condition that renders the definition faithful to the required intuition.

Equation (3.15) is perhaps not so suprisingly quite similar to the definition of a simulation.[16] After all, simulations are a standard means to prove equivalence between transition systems. If the subscripts of β are ignored and we assume t' is uniquely determined by a function we call μ for consistency like the refactoring itself, β is a simulation between the transition systems $\Gamma \vdash .\dot{\rightarrow}.$ and $\Gamma' \vdash .\xrightarrow{\mu(.)}..$ β's correspondence to the *coupling relation* in *refinement* then becomes apparent.

Note that the definition equation (3.16) is asymmetric. It makes perfect sense to demand that the definition of a refactoring is symmetric, making it resemble a *bisimulation.* This has favourable consequences as discussed later but is of little general relevance if refactorings qua (unidirectional) transformations are concerned. Definition equation (3.16) and its variants are the main ones used in the text.

$$\mu \text{ is a } \textit{symmetric} \text{ refactoring with correspondence } \beta \equiv \\ \text{pre}(\Gamma) \wedge \text{post}(\mu(\Gamma)) \Rightarrow \mathit{ExtEq}_{\beta}(\Gamma, \mu(\Gamma)) \wedge \mathit{ExtEq}_{\beta^{-1}}(\mu(\Gamma), \Gamma) \quad (3.17)$$

[16]For the readers who are surprised by my understanding of what a *simulation* encompasses and those who are not completely familiar with the notion itself, here is the general definition I assume: let $\mathcal{S} = (S, A, \underset{\mathcal{S}}{\longrightarrow})$ and $\mathcal{R} = (R, A, \underset{\mathcal{R}}{\longrightarrow})$ be *transition systems* with states S and R, actions A and transition relations $\underset{\mathcal{S}}{\longrightarrow}$ and $\underset{\mathcal{R}}{\longrightarrow}$. $b \subseteq S \times R$ is then a simulation iff whenever $s \xrightarrow[\mathcal{S}]{a} s' \wedge b(s, r)$, there is some r' such that $r \xrightarrow[\mathcal{R}]{a} r' \wedge b(s', r')$. b is a bisimulation if both b and b^{-1} are simulations.

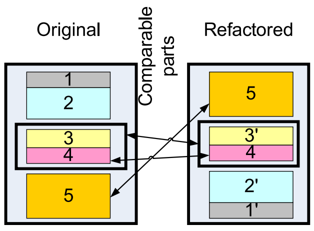


Figure 3.7.: Comparable parts in the refactored programs

Two important variants

More often than not, the definition in equation (3.15) is overly general. As the informal legitimation of equation (3.15) in the previous sections showed, only certain parts of the definition are actually needed for each sufficiently simple refactoring. The following paragraphs summarize the variants that are actually needed in the refactorings discussed in the rest of the text.

Variant 1: β is the identity Many refactorings do not change the state space of the program. “Simplify Conditional” or “Strength Reduction” are examples. Different code is used to achieve the same effect, including state updates. Another more important example is reordering independent statements. Example:

`a++;`	`b++;`
`b++;`	`a++;`

Comparable statements

β is in this case the identity for all statements t and t' for which $\beta_{[t,t']}$ is not empty. Such statements are called “comparable”. They are the statements that correspond to each other in the original and the refactored program and are not affected by the transformation. Equation (3.15) has to be proven directly for all program parts that are not comparable while an inductive method can be leveraged for all comparable parts. This suggests that all regions unaffected by a refactoring are considered comparable. Figure 3.7 illustrates the matter.

$$ExtEq_\beta(\Gamma,\Gamma') \equiv \forall s, \text{comparable } t \text{ and } t', s' : (\Gamma \vdash s \xrightarrow{t} s') \Rightarrow (\Gamma' \vdash s \xrightarrow{t'} s') \tag{3.18}$$

The crucial point now is that equivalence can be temporarily invalidated, namely for the statements that are not comparable. Consider the example above again. The statements are now labelled.

$$t \left\{ \begin{array}{l} t_1 \left\{ \texttt{a++} \right. \\ t_2 \left\{ \texttt{b++} \right. \end{array} \right. \qquad t' \left\{ \begin{array}{l} t'_1 \left\{ \texttt{b++} \right. \\ t'_2 \left\{ \texttt{a++} \right. \end{array} \right.$$

Statements t_1 and t'_1 are obviously not comparable. Given an identical initial state, the first program will have a updated, the second, refactored program, will not. Instead, b will be incremented by one. The second statement will not start with β-compatible states. After its execution however, correspondence is reestablished. t and t' are thus comparable: If t and t' start in compatible states – in this case this means equivalent states, the states will be compatible again after the execution of t even if it is temporarily violated during the execution of t. The possibility to violate data correspondence temporarily greatly increases the flexibility you have to formalize a refactoring. It is exactly the kind of flexibility you need because refactorings normally leave most statements unaffected but change a few, that are known and have specific properties that can be used for the definition of the data correspondence.

Comparable statements are like "visible states" JML assumes during which class invariants have to hold.

Variant 2: β is invariant For most refactorings, in particular all the refactorings in this text, β is a kind of *invariant* that must be reestablished by every relevant statement (i.e., one that is comparable to its refactored counterpart) and may also be assumed by it. Statements t and t' are comparable if $\beta_{[t,t']}$ is not empty as above. The "comparability" criterion in that case replaces the correspondence between different statements in the original and the refactored program.

Most important variant!

$$\boxed{\begin{aligned} \mathit{ExtEq}_\beta(\Gamma,\Gamma') \equiv \forall s, \text{comparable } t \text{ and } t', s', r : (\Gamma \vdash s \xrightarrow{t} s') \wedge \beta(s,r) \\ \Rightarrow \exists r' : (\Gamma' \vdash r \xrightarrow{t'} r') \wedge \beta(s',r') \quad (3.19) \end{aligned}}$$

It is important to understand that for a refactoring μ, there is no single correct β. Different β can be defined. Some of them may be weak enough to be invariants, others may depend on the actual statement being executed. The reordering statements example is again good enough as an illustration: If only t and t' are considered comparable, β is very simply the identity on comparable statements: $\beta = \text{id}$. If all statements are considered comparable, β has to encode the temporary violation of the invariant. $\beta_{[t,t']}$ and $\beta_{[t_1,t'_1]}$ are still id, but after the execution of the first statement, the state spaces are different

$$\beta_{[t_2,t'_2]}(s, s[\sigma := \sigma[\mathtt{a} := \mathtt{a} - 1, \mathtt{b} := \mathtt{b} + 1]])$$

For more complex refactorings there are more choices you can make with more or fewer comparable statements. This choice influences the complexity of a correctness;

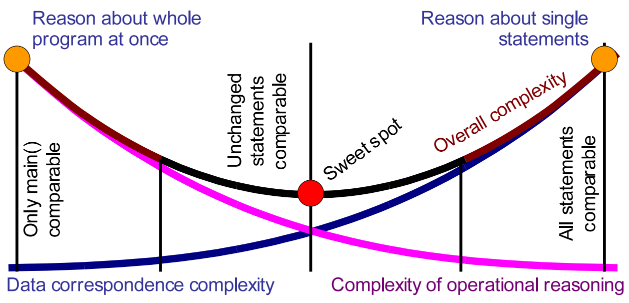

Reason about whole program at once
Reason about single statements
Only main() comparable
Unchanged statements comparable
Sweet spot
Overall complexity
All statements comparable
Data correspondence complexity
Complexity of operational reasoning

Figure 3.8.: The spectrum of possible data correspondences and associated complexity

Comparability *Every* refactoring can be explained with a detailed β that allows comparability of *all* statements, including those that are subject to the refactoring. β "knows" the refactoring. Such β are difficult to specify and duplicate the work done for formalizing the transformation itself. Specification and verification of refactorings are only practical if the definitions can be kept simple. Leaving β undefined everywhere except the main routines makes it very simple but not very descriptive and renders the proof unncessarily complicated. *β should therefore capture the essence of the refactoring while leaving the description of the transformation to μ as far as possible.*

Unreasonably well behaved β

For usual refactorings, there is only one β that faithfully reflects intuition and results in simple proofs. It is the β that considers only those statements comparable that are unchanged by the refactoring, even if their components are updated. Cf. figure 3.8

It is clear that comparability cannot serve as a foundation for the notion of equivalence if programs do not terminate *in general*. This is why termination should be required separatly for individual program relevant parts if they are also comparable. For such parts, equivalence of termination must be shown (i.e., if one program part terminates in the first program, the corresponding program part in the second program part must also terminate. In practice, every *unexamined* program part may also do I/O, which yields a sensible notion of equivalence at least for those parts.

3.3. Extensions to the basic model

3.3.1. Syntactic approximation with preconditions

Refactorings as discussed by Fowler [16] are meant to preserve equivalence. This is their correctness criterion and unit tests normally serve as a specification as noted above – the most prominent advocate of this idea is Roberts in his PhD thesis [39], see section 1.4.2 for details.

Unit tests as specifications naturally lead to “refactorings” that do not retain the semantics for all inputs. Instead, the inputs that are contained in the tests are actually the execution “pre”-conditions in the meaning of a condition that must hold for the program to yield the expected results just as it is used in software verification. The formalized refactorings in this text do not incorporate the notion of such execution preconditions, which may be essential to the applicability of refactorings. This is not a problem if semantic conditions are tested dynamically, during program execution because every excecution will necessarily satisfy the preconditions. It is a problem however if conditions are verified statically using syntactic approximation – a possibility I cordially embrace.

The problem occurs as well for static verification, but it is more naturally taken into account there. It is unreasonable to expect that useful preconditions can be “extracted” from tests in either case as it involves generalization.

Here is a simple idea: A specified execution precondition is propagated to all parts of the program using a data flow analysis (or abstract interpretation). These data abstractions cannot possibly be overly complex but may include value abstractions of individual data structures at each program point.

The special status of dataflow analyses for static approximations to postconditions

As I said above, value propagation does not have any benefit for the *dynamic* tests that are injected into the program by the refactoring tool, because the possible inputs are conformant with the precondition. If syntactic approximations are implemented however, the precondition become important when the syntactic approximation can be formulated in the same framework as the propagation algorithm – i.e., some kind of data flow analysis. If the initial state of the analysis is reduced to the precondition, static approximations take into account the presence of preconditions. This is of course applicable to parts of programs as well as whole programs.

Example 3.6. Imagine a “program”[17] takes an input x as its argument. x is guaranteed to be in $\{1, 2, 3\}$. For the refactoring “Move Field”, the pointer to the object to which the field is moved must not be null at program points when the moved field is accessed. Consider the following program

```
    ...
    z.target = null;
    switch(x){
    case 1: x.target = Some non-null expression; break;
    case 2: x.target = Some non-null expression; break;
    case 3: x.target = Some non-null expression; break;
    }
    t = z.f
    ...
```

[17]It is just the entity outside of which I do not want to analyze code.

Applying “Move Field” to this program – moving `f` from ctt(`z`) to ctt(`z.target`) will require the postcondition `z.target` $\neq$ Null at line 8. A non null analysis could only guarantee this claim if the additional knowledge about `x` $\in \{1, 2, 3\}$ is taken into account.

This technique could – depending on the speed of the analysis – outweigh the flexibility lost initially acquired by formalized refactorings. Keep in mind however that formalized refactorings are most useful when it is not (yet) possible to test code. Writing white box tests does have many other functions beyond checking refactorings that are discussed in the introduction.

3.3.2. Reverse transformations

As noted in [16, chap. 14 by Don Roberts and John Brant], it is particularly important that the effect of a refactoring can be reversed. Simplicity of transformation should not be an argument for investigating a transformation instead of its reverse. Moreover, refactorings may turn out to be bad and must be undone long after the transformation has initially happened without affecting the changes made later. In principle, the backward transformation is easy to formulate if you have the forward transformation: The postcondition becomes the precondition and the precondition becomes the postcondition.

This does not work in general, but it does if the refactoring is symmetric as in equation (3.17). The original refactoring is μ_1 with data correspondence β_1 and pre- and postconditions pre_1 and post_1. The refactoring then satisifes equation (3.17)

$$\text{pre}_1(\Gamma) \wedge \text{post}_1(\Gamma') \Rightarrow \mathit{ExtEq}_{\beta_1}(\Gamma, \mu_1(\Gamma')) \wedge \mathit{ExtEq}_{\beta_1^{-1}}(\mu_1(\Gamma'), \Gamma)$$

The inverse refactoring is $\mu_2 = \mu_1^{-1}$ with data correspondence $\beta_2 = \beta_1^{-1}$ and pre-/postconditions $\text{pre}_2 = \text{post}_1$ and $\text{post}_2 = \text{pre}_1$. As can be trivially verified, equation (3.17) is satisfied for the new refactoring if μ_1 is injective (and μ_2 is therefore defined). If this is not the case, finding a μ_2 such that $\mu_1(\mu_2(\Gamma')) = \Gamma'$ for all $\Gamma' \in \{\mu_1(\Gamma) | \Gamma \in \mathit{Prog} \wedge \text{pre}_1(\Gamma)\}$ may be an acceptable solution.

Are postconditions of the forward refactoring good preconditions for the backward refactoring?

Consider “Move Field”. (section 5.4) A precondition for the reverse transformation would be that every access to the field would happen with the field access statements the forward transformation produces. This is quite acceptable as the program can probably be brought into this shape by reordering instructions and forward substitution. This is certainly an *essential applicability condition* for the reverse transformation. What about aliasing conditions that also play a role? It is hard to defend that they are essential for the applicability of the refactoring. These should be framed as postconditions in the reverse refactoring and preconditions in the original refactoring.

Translating between pre and postconditions is easily possible if β is a function – which is indeed the case for “Move Field”. Let the state[18] in the untransformed program be s

[18] or tuple of states

and in the transformed program be r as in equation (3.15). We call the precondition P and the corresponding postcondition P'. At least when $\beta(s,r)$ is satisfied, the pre- and postconditions have to correspond, i.e., $\beta(s,r) \Rightarrow (P(s) \iff P'(r))$. If the postcondition P' is given, $P(s) = P'(\beta(s))$ trivially satisfies the equation. The opposite is of course true if β^{-1} is a function.[19] This leads to the following conclusion: As long as β is a function in its first argument, the reverse transformation can leverage the postconditions of the forward transformation to generate its own postconditions.

I treat β as a function

Are preconditions of the forward refactoring good postconditions for the backward refactoring?

It is worth noting that most refactorings do not have postconditions, for these refactorings, the opposite problem is most imminent and the opposite condition must hold.

3.3.3. Coping with variant control flow and externally visible state in practice

For most refactoring, talking about the "world" or "the sequence of I/O operations" seems rather heavyweight and superflous. Often, a simple informal argument is sufficient to *support* the claim that the refactorings do not change the externally observable behavior. As long as the methods with external effect (i.e., I/O routines) are called in the same order and number, the programs are equivalent. Control flow does only change if (a) exceptions or (b) the evaluation of basic expressions change.[20]

The externally visible semantics need not be taken into account. For refactorings that do actually alter the control flow of a program in non-trivial manners, it might be worthwhile to explicitly reason about visible state and its history ("the world"). These refactorings are quite rare in practice and include:[21] "Replace Exception by Check", "Replace Typecase by Dynamic Dispatch", "Replace Error Code with Exception", etc.

Introducing the "world" abstraction does not actually alter what is possible to examine, it merely introduces a formal entity that represents what is being reasoned about therefore providing tangible argumentation.

Example 3.7. "Replace Exception by Check" is a refactoring that benefits marginally from the world abstraction.

[19]If the precondition is given and β^{-1} is a function, the postconditions can be derived, which are the preconditions of the reverse transformation.

[20]The approach is reminiscent of the notion of "preservation" in [29]. [29] does not go any further to examine what the semantic consequences of "access", "update", "call" preservation are – they just name it.

[21]They may be rare just because control flow alterations do not benefit as much from automation as other refactorings.

```
class C {                                   class C {
  void m(D o, Params params){                 void m(D o, Params params){
    try{                                        if(o.noExc(params)){
      o.f(params);                  ⇒             o.f(params);
    }catch(MyException e){                      }else{
      S();                                        S();
    }                                           }
  }                                           }
}                                           }
```

$\mathfrak{u}$ is altered by both the call `o.f(params)` and `S()`. The statements implementing these two routines remain the same. $\mathfrak{u}$ is therefore updated in the same way before and after refactoring if the context in which they are invoked is also the same.

For the proof of “Replace Exception by Check”, it is necessary that `o.noExc(params)` does not update any state local variables that is relevant for the remainder of the program’s execution.

For practical purposes, it is sufficient that the changes leave the externally visible reactions to these changes identical. It is a very weak condition that could for instance be reduced to observational purity, to identity of the reachable state or any other notion that is handy to check.

3.3.4. Data-vs-instruction induction refactorings

For refactorings that mainly affect the data space of the program, giving only β could allow to derive a provably correct refactoring μ. The more conventional direction would be to derive β from μ even if care is taken to leave β unspecified where the actual transformation happens. As an example, let’s consider the “Move Field” refactoring. β is a function in its first argument: β. β undoes the effect of the program transformation on the state space. Conversly, μ undoes the effect of β:

$$\Gamma \vdash s \xrightarrow{t} s' \Rightarrow \mu(\Gamma) \vdash \beta(s) \xrightarrow{\mu(t)} \beta(s')$$

The idea is now to identify load and store statements with every data access path.[22] β is presented as a table of data correspondences with the invariant data paths stripped away. For “Move Field”, it looks like this:

Before	After
$\gamma(loc)(Src{::}f)$	$\gamma(\gamma(loc)(Src{::}target))(Target{::}f)$

[22]This translation is straigforward: $\gamma(\gamma(x)(f))(d)$ for instance would generate the statements $tmp{\leftarrow}x.f;tmp.d{\leftarrow}\_$ and $tmp{\leftarrow}x.f;\_{\leftarrow}tmp.d$

Refactorings that can be handled in this very reduced framework are those that are concerned with data access alone: “Move Field”, “Rename Field”, “Replace Array with Object”, “Rename Local Variable/Parameter”. Others like “Split Temporary Variable” need more information: It is not only a matter of replacing one access by another but of replacing one access by two others. Still, the replacement of one access by another is an important element in this procedure.

The fundamental difference between transformations altering the access path to a value in the object graph and other conventional refactorings is that refactorings are normally concerned with bringing code into a certain shape that is not determined by the change in the state space alone.

Access paths mappings are a framework that allows the general substitution of one sequence of access statements (i.e., field/variable load and store) by another sequence. It can also guide the user of a refactoring tool using this formalism to the locations in the code where modifications have to be made to keep two copies of the same value in sync.

Refactorings of this kind are covered in detail in chapter 5.

3.4. A proof-pragmatic classification of refactorings

Different refactorings differ in the program parts and abstractions they modifiy. It might be useful to classify the refactorings according to these dimensions to identify other refactorings and topics that could be worth investigating. I present such a classification here. Clearly, refactorings can modify the data structure or the program representation. Changes in the data structure are always reflected in the program structure but not vice versa. Refactorings can retain or they can alter the control flow. Refactorings can have a reverse transformation that is easily applicable or they may not.

A classification of refactorings could focus on the programmatic aspects of refactoring.[23] A more interesting classification focuses on the way the program behaves and how it differs after the refactoring. This is relevant for the proof strategy.

The classification is therefore instructive to capture the different approaches that are needed for the correctness proofs. It is also *necessary* because I want to have at least one refactoring in each class of modifications that I have shown correct. There are three relevant axes to be considered:

[23]Fowler [16] also implies a classification of refactorings by the organisation of the catalogue into chapters. This implicit classification is organized along the lines of syntactic elements and intentions that are less relevant for the theoretical framework here.

Control flow preservation If control flow does not change, externally visible program behavior remains trivially the same as long as external method calls are not changed and their argument values remain the same. If control flow *does* change however, it is necessary to capture the externally visible effects of code explicitly in the state.[24] See section 3.3.3

There are not many refactorings in [16] that do change the control flow of the program. They are limited to

- "Replace Exception by Check",
- "Replace Typecase by Dynamic Dispatch",
- "Replace Error Code with Exception"

Locality of code changes Refactorings can either be purely local or they can be scattered across the whole program. Most refactorings are purely local. Purely local refactorings assume a stringent structure of the program that is being transformed – if they didn't, they couldn't transform it. Because of the stringent structural assumptions, semantic postconditions are mostly essential applicability conditions that are too difficult to check syntactically. Most refactorings in [16] involve only local modifications.

Global data variance (access/create-path preservation) Some refactorings operate on the object representation: They move fields up, down and across the hierarchy, alter the representation of fields, introduce them and remove them.

I want to keep the refactorings as simple as possible, to be able to prove correctness easily, even if this means that they have to be composed or applied repeatedly.

- If the access paths are not kept the same, the effect cannot always be local: In the absence of visibility restrictions, it is impossible to guarantee that access paths are confined to certain parts of the program. Global data changes should be isolated from changes to local data by data mapping. This avoids having to update unrelated user code.

Global data variance ⇒ Non-local modifications

- If every access path is retained, it is impossible that the refactoring is changing an unbounded number of program parts – if the refactoring is atomic. This is so because data invariant changes are in fact *independent* of each other and the refactoring could be split into multiple simpler refactorings.

Global data invariance ⇒ local modifications

- If the control flow is not retained, modification should be local. If control flow changes, modifications to the local state σ are made. This cannot be due to access path changes because they should be isolated by data translation. Changes in control flow must therefore be due to other local changes. Local changes can be made in isolation however and if they are not, the refactoring is not atomic. Likewise, control flow changes should not involve access path modifications because

Control flow changes ⇒ local modifications

Control flow changes ⇒ no global state changes

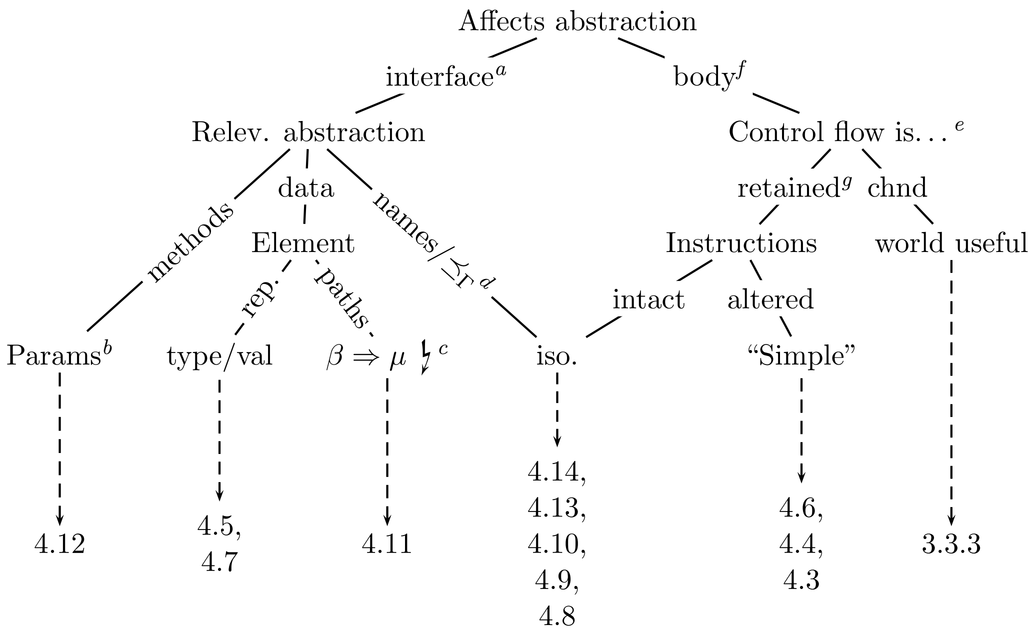
[24]It can still be argued about correctnenss, but not in a mathematically rigorous way.

access path changes are not local.

Some refactorings do not fit into this simplified classification. They do not affect control flow or global state, but they do affect the program non-locally. The simplest example is “Rename Method”. These refactorings are isomorphisms on the relevant program parts. The classification above is lacking them for exactly this reason: If they were more than isomorphisms, we would have been able to anticipate them with the classification above that is based on the necessities for proofs. Renaming however does not require a sophisticated proof for showing program equivalence. Every isomorphism can be eliminated by adding an extra level of indirection to the operational semantics.[25] This leads to the classification tree in figure 3.9.

All refactorings of abstraction interfaces may be subject to non-local transformations. Fortunately, methods are the only kind of abstraction in Java that is unrelated to the heap state *per se*. Another, related issue is class hierarchies: altering only the class hierarchy only should not change the way the application works. However, it may modify the way methods are looked up – even though the results do not change. They are classified as isomorphisms and “simple” refactorings.

[25]Example: Use indices instead of parameter names, etc.



[a]Implies dispersed changes. Conventional wisdom suggests that there are no usable *atomic* refactorings that modify both the interface to an abstraction and the control flow on individual method bodies.

[b]These can be made local by "Extract Method" followed by "Move Static Method" and "Inline Method." See "Make Method Static" for an example of such a refactoring.

[c]This is the only difficult class of refactorings. Unlike method calls for instance, these refactorings cannot be decomposed into a few refactorings that concentrate the whole complexity because the path replacement is only subject to the object structure the program establishes.

[d]The method resolution procedure may change while the method to be invoked does not usually change in normal refactorings. I cannot imagine any usable *atomic* refactorings that change the class hierarchy and *at the same time* change the result of method lookup.

[e]An orthogonal classification is whether the data mapping is path dependent or not.

[f]Implies locally confined updates

[g]Most refactorings discussed and formalized in other texts fall into this category. They are the simplest but also the most useful refactorings. Some of them are discussed in chapter 4.

Figure 3.9.: Classification according to proof strategy and references to discussed refactorings chapter 4

4. Simple Refactorings Proved Correct

This chapter discusses some refactorings that are important yet “simple” in the sense that they do either not involve complex behavioral correctness conditions or behavioral conditions that are formulated synactically, drawing from existing compiler optimization analyses. These refactorings include all those that are needed in the introductory example in chapter 2.

This section is also meant to provide some examples of simpler refactorings and ought to give a feeling how the framework provided in this text can guide the analysis of other refactorings as needed for a refactoring tool.

The goal is to show that the refactorings are often very simple, something that is not clear when reading other texts on refactoring. “Rename Method” is treated quite formally to illustrate the proof strategy. Later sections are less formal.

For the definition of β, it is convenient to definitionally extend it pointwise from correspondences of the components of the state, which I also call β.

$$\beta(s, r) = \beta(s.\,\mathrm{xcpt}, r.\,\mathrm{xcpt}) \wedge \beta(s.\sigma, r.\sigma) \wedge \beta(s.\,\gamma, r.\,\gamma) \wedge \beta(s.\,\mathfrak{u}, r.\,\mathfrak{u})$$

β is the identity for components that are left unmentioned. The same notation is used in chapter 5.

Contents

4.1. “Rename Method” . . . 76
4.2. “Replace Type Code with Subclass” . . . 82
4.3. “Extract Method” . . . 90
4.4. “Inline Method” . . . 95
4.5. “Replace Representation” . . . 97
4.6. “Replace Expression” . . . 101
4.7. “Replace Type Tests with Dynamic Dispatch” . . . 103
4.8. “Use Reducible Language Features (Syntactic Sugar)” . . . 104
4.9. “Make Method Static” . . . 107
4.10. “Make (Extracted) Method Instance-Bound” . . . 108
4.11. “Move Local to Object” and “Field to Local” . . . 110
4.12. “Rename/Reorder Parameter” . . . 113
4.13. “Rename Field” . . . 114
4.14. “Delete Obsolete Element” . . . 114
4.15. Conclusion . . . 115

4.1. “Rename Method”

The “Rename Method” refactoring renames a method and all its occurences – just as the name implies. This refactoring is widespread, conceptually simple and it is an isomorphism.

As with all the refactorings, “Rename Method” is chiefly described by a transformation function μ. Before I continue defining μ for this first simple case and proving by induction that equation (3.16) is satisfied, let me first quickly introduce an easy notational scheme: The refactoring’s name is written as a superscript, the arguments to the refactoring are written as sub-scripts. $\mu^{\text{Refactoring name}}_{\text{arguments}}$ The same notation is used for data correspondence β. For the discussion of a single refactoring, I am just using μ and β instead of the qualified symbols.

For method renaming, the declaring class or interface, the old signature and the new signature of the method must be specified: $\mu^{\text{Rename Method}}_{class,old\_sig,new\_sig}$ Two signatures are supposed to differ only by their name: $new\_sig = (new\_name, tys)$ and $old\_sig = (old\_name, tys)$

Before continuing, let me give a simple example of the “Rename Method” refactoring. Method **void** `B::m()` is to be renamed to **void** `B::n()`. `B` implements both **void** `I::m()` and **void** `J::m()`. They will both have to be renamed as well as the methods implementing and overriding them.[1]

[1]The situation is more complicated in .NET. The corresponding C# program, `A::m` can implement both `I::m` and `J::m` just as in Java, but it does not have to. In fact, interface method implementations and class methods are separate. The following is therefore allowed:

```
interface I {
    void m();
}
interface J {
    void m();
}

class A : I, J {
    void I.m() {
        con.WriteLine("A::I.m");
    }
    void J.m() {
        con.WriteLine("A::J.m");
    }
    public virtual void m() {
        con.WriteLine("A::m");
    }
}
```

The Java-like construct does also exist of course:

```
class B : I, J {
    public void m() {
```

```
interface I{                 interface I{
    void m();}                   void n();}
interface J{                 interface J{
    void m();}                   void n();}

class A implements I,J{      class A implements I,J{
  public void m(){             public void n(){
    out.println("A::m"); }}      out.println("A::m"); }}
class B extends A{           class B extends A{
  public void m(){             public void n(){
  out.println("B::m"); }}        out.println("B::m"); }}
class C extends A{           class C extends A{
  /* inherit A::m */}          /* inherit A::n */}

class X implements J{        class X implements J{
  public void m(){             public void n(){
  out.println("X::m"); }}        out.println("X::m"); }}
```

This example shows is that the difficulty of this refactoring is merely to identify all the methods that are related to the method that is being renamed. I call the set of these methods $\mathcal{S}$. It contains both the methods a method implements/overrides as well as the methods that implement them.

$$\mathcal{S} \equiv \bigcup \{ \mathit{override}_\Gamma(T, m) | (T, m) \in \mathit{overriden}_\Gamma(\mathit{class}, \mathit{old\_sig}) \}$$

The *overriden* function returns those class/method-signature pairs that are overriden or implemented by its parameter while the *override* function returns those that override the given method including abstract methods and methods redeclared in related interface definitions.[2]

```
            con.WriteLine("B::m");
        }
    }
```

[2]Here is a formal definition employing the *isa* relation: $\mathit{isa}1_\Gamma(A,B) \equiv \Gamma[B = A.\text{superclass} \vee B \in A.\text{superifaces}]$ and consequently, $\mathit{isa}_\Gamma = \mathit{isa}1^*_\Gamma$

$$\mathit{overriden}_\Gamma(T, m) \equiv \{(B, m) | T \mathit{isa}_\Gamma B \wedge m \in \Gamma[B.\text{methods}]\}$$

The other auxiliary function is almost identical.

$$\mathit{override}_\Gamma(B, m) \equiv \{(T, m) | T \mathit{isa}_\Gamma B \wedge m \in \Gamma[T.\text{methods}]\}$$

Transformation 4.1.1. Having defined the set of "affected" definitions $\mathcal{S}$, it is easy to formulate the transformation concisely.

$$\begin{aligned}\mu^{\text{Rename Method}}_{class,old\_sig,new\_sig}(\Gamma) = \Gamma[&\mathsf{if}\ (T,m) \in \mathcal{S}\ \mathsf{then}\\ &T.\,\mathrm{methods}\,.m := undef, T.\,\mathrm{methods}\,.new\_sig := \cdot.m,\\ &*.l_1{\leftarrow}l_2.T{::}m(e) := l_1{\leftarrow}l_2.T{::}new\_sig(e),\\ &*.l_1{\leftarrow}l_2.m(e) := l_1{\leftarrow}l_2.new\_sig(e)\\ &]\end{aligned}$$

Data correspondence 4.1. "Rename Method" leaves data unaffected. This is reflected in β for which $\beta = \mathrm{id}$. I.e., equation (3.18) can be used to prove correctness. "Rename Method" is an isomorphism. Two statements are therefore comparable if they are identified by the same path (up to isomorphism), i.e., in some method, statement 34 in the original program still corresponds to statement 34 in the refactored program. Statements in method declarations that are renamed correspond to each other of course. The statement path correspondence is called μ as in chapter 3, i.e., statement t corresponds to statement $\mu(t)$ and t, $\mu(t)$ are always comparable.

4.1.1. Proof

The next step is to check, *by induction on the derivation of* $\Gamma \vdash s \xrightarrow{t} s'$, whether the transformation acutally satisfies the equivalence criterion equation (3.18), which I repeat for clarity

$$\forall s, \text{comparable } t \text{ and } t', s' : (\Gamma \vdash s \xrightarrow{t} s') \Rightarrow (\Gamma' \vdash s \xrightarrow{t'} s')$$

For "Rename Method", all statements are comparable because the data correspondence is not even temporarily violated.

The proof is conducted in a strictly step-wise manner that can seem trivial from time to time. I do believe however that this is the only effective way to prevent misunderstandings and valuable for the first example of a formal treatment in this text.

Lemma 4.1. For the proofs, note first of all that the result of simple expressions are not affected by the transformation. This is trivial, because $\beta = \mathrm{id}$

$$[\![e]\!]^{s}_{\Gamma} = [\![e]\!]^{\beta(s)}_{\mu(\Gamma)} \tag{4.1}$$

With this lemma in mind, let's continue to the proof of the individual kinds of statements. For the proof in this section, the world $\mathfrak{u}$ part of the state is irrelevant as are $\mathfrak{u}$-updates because of equation (4.1).

Case Skip: `skip` For `skip`, the result state does not depend on Γ. Moreover $\mu(\texttt{skip}) = \texttt{skip}$. The implication is trivial.

Similar: Write Local $l \leftarrow e$

Exception Propagation The result state is independent of Γ as well as the statement t. Equation (3.18) is satisfied in that case.

Case Chain $t_1;t_2$ According to the induction hypothesis, both $\mu(\Gamma) \vdash s_0 \xrightarrow{\mu(t_1)} s_1$ and $\mu(\Gamma) \vdash s_1 \xrightarrow{\mu(t_1)} s_2$ hold. Because of the unique derivation of $t_1;t_2$,

$$\mu(\Gamma) \vdash s_0 \xrightarrow{\mu(t_1);\mu(t_2)} s_2$$

holds. This is, according to the definition of μ, $\mu(\Gamma) \vdash s_0 \equiv (\text{None}, \sigma, \gamma, \mathtt{u}) \xrightarrow{\mu(t_1;t_2)} s_2$. qed.

The same approach works for all other statements for which μ is just used to translate embedded statements, i.e., $\texttt{while}(e)\{t\}$, $\texttt{try}\{t_1\}\texttt{finally}\{t_2\}$, $\texttt{try}\{t_1\}\texttt{catch}(T\ l)\{T_2\}$.

Case While: $\texttt{while}(e)\{t\}$ There are two possible derivations for while loops. The case when e is similar to $t_1;t_2$, so I consider $[\![e]\!]^s_\Gamma = \text{true}$ here.

According the the induction hypothesis, $\mu(\Gamma) \vdash s_0 \xrightarrow{\mu(t)} s_1$ and $\mu(\Gamma) \vdash s_1 \xrightarrow{\mu(\texttt{while}(e)\{t\})} s_2$ hold. According the definition of μ, it is equivalent to $\mu(\Gamma) \vdash s_1 \xrightarrow{\texttt{while}(e)\{\mu(t)\}} s_2$. Combining the (cf. table 3.2), we get $\mu(\Gamma) \vdash s_0 \xrightarrow{\texttt{while}(e)\{\mu(t)\}} s_2$, which is $\mu(\Gamma) \vdash s_0 \xrightarrow{\mu(\texttt{while}(e)\{t\})} s_2$

Case New $l \leftarrow \texttt{new}\ C$ **and** $l \leftarrow \texttt{new}\ C[e]$ The proof does not really depend on the intricacies of array allocation, so I am proving the case for $l \leftarrow \texttt{new}\ C$. It has to be shown that

$$\Gamma \vdash (\text{None}, \sigma, \gamma) \xrightarrow{l \leftarrow \texttt{new}\ C} (\text{None}, \sigma[l \mapsto loc], \gamma[loc \mapsto \text{init\_obj}(\Gamma, C)])$$

implies

$$\mu(\Gamma) \vdash (\text{None}, \sigma, \gamma) \xrightarrow{\mu(l \leftarrow \texttt{new}\ C)} (\text{None}, \sigma[l \mapsto loc], \gamma[loc \mapsto \text{init\_obj}(\Gamma, C)])$$

Γ is arbitrary and $l \leftarrow \texttt{new}\ C = \mu(l \leftarrow \texttt{new}\ C)$:

$$\mu(\Gamma) \vdash (\text{None}, \sigma, \gamma) \xrightarrow{\mu(l \leftarrow \texttt{new}\ C)} (\text{None}, \sigma[l \mapsto loc], \gamma[loc \mapsto \text{init\_obj}(\mu(\Gamma), C)])$$

This is satisfied according to the definition of init_obj because init_obj depends at most on fields($\Gamma(ty)$), which is left unchanged by μ: $\gamma[loc \mapsto \text{init\_obj}(\mu(\Gamma), C)] = \gamma[loc \mapsto \text{init\_obj}(\Gamma, C)]$.

Case Throw $\texttt{throw}\ e$ **Similar:** Cast $l \leftarrow (T)e$

Subcase $[\![e]\!]^s_\Gamma \neq$ Null . See “Case Skip”

Subcase $[\![e]\!]^s_\Gamma =$ Null . See “Case New”

Case Finally $\texttt{try}\{t_1\}\texttt{finally}\{t_2\}$ See “Case Chain”

Case Catch $\texttt{try}\{t_1\}\texttt{catch}(C\ l)\{t_2\}$ See “Case Chain”

Case Write Local See “Case Skip”

Case Write Field, Read Field, Write Array, Read Array See “Case Skip” and “Case New”

Case Initialize Class

Subcase $\gamma(C) \neq$ None . See “Case Skip”

Subcase $\gamma(C) =$ None . We can assume $\gamma(C) = \text{None}$, $\gamma' = \gamma[C \mapsto \text{init\_obj}(\Gamma, \text{metaclass}(C))]$ and

$$\Gamma \vdash (\text{None}, \{\texttt{this} \mapsto C\}, \gamma_1, \mathfrak{u}_1) \xrightarrow{\text{staticinitializer}(\Gamma(C))} (x, \sigma, \gamma_2, \mathfrak{u}_2)$$

According the the induction hypothesis,

$$\mu(\Gamma) \vdash (\text{None}, \{\texttt{this} \mapsto C\}, \gamma_1, \mathfrak{u}_1) \xrightarrow{\mu(\text{staticinitializer}(\Gamma(C)))} (x, \sigma, \gamma_2, \mathfrak{u}_2)$$

holds. According the definition of μ,

$$\mu(\text{staticinitializer}(\Gamma(C))) = \text{staticinitializer}(\mu(\Gamma)(C))$$

Therefore,

$$\mu(\Gamma) \vdash (\text{None}, \{\texttt{this} \mapsto C\}, \gamma_1, \mathfrak{u}_1) \xrightarrow{\text{staticinitializer}(\mu(\Gamma)(C))} (x, \sigma, \gamma_2, \mathfrak{u}_2)$$

Moreover, because μ does not affect fields and init_obj depends at most on fields.

$$\gamma[C \mapsto \text{init\_obj}(\Gamma, \text{metaclass}(C))] = \gamma[C \mapsto \text{init\_obj}(\mu(\Gamma), \text{metaclass}(C))]$$

Therefore, $\mu(\Gamma) \vdash (\text{None}, \sigma, \gamma, \mathfrak{u}) \xrightarrow{\mu(\texttt{init\_class}\ C)} (x, \sigma, \gamma_2, \mathfrak{u}_2)$ holds according to the derivation of $\texttt{init\_class}\ C$.

Case Invoke Special/Virtual $l_1 \leftarrow l_2.T{::}m(e)$ **and** $l_1 \leftarrow l_2.m(e)$ This is very similar to “Case Class Initialization”. I treat the case “Invoke Special” only. “Invoke Virtual” is – with $\text{rtt}(\gamma(x))$ instead of T – basically the same. If $(T, m) \notin \mathcal{S}$, the cases become trivial. Assume $m = \textit{old\_sig}$.

Subcase $x =$ Null . See “Case New”

Subcase $x \neq$ Null Assume $x = [\![l_2]\!]^s_\Gamma$, $x \neq$ Null and

$$\Gamma \vdash (\text{None}, \{\texttt{this} \mapsto x, \text{pNames}(\Gamma, T, m) \mapsto e\}, \gamma, \mathfrak{u}) \xrightarrow{\text{body}(\Gamma,T,m)} (\text{xcpt}, \sigma', \gamma', \mathfrak{u}')$$

According the the induction hypothesis,

$$\mu(\Gamma) \vdash (\text{None}, \{\texttt{this} \mapsto x, \text{pNames}(\Gamma, T, m) \mapsto e\}, \gamma, \mathfrak{u}) \\ \xrightarrow{\mu(\text{body}(\Gamma,T,m))} (\text{xcpt}, \sigma', \gamma', \mathfrak{u}')$$

This is

$$\mu(\Gamma) \vdash (\text{None}, \{\texttt{this} \mapsto x, \text{pNames}(\mu(\Gamma), T, \mathit{new\_sig}) \mapsto e\}, \gamma, \mathfrak{u}) \\ \xrightarrow{\text{body}(\mu(\Gamma),T,\mathit{new\_sig})} (\text{xcpt}, \sigma', \gamma', \mathfrak{u}')$$

From which we can deduce

$$\mu(\Gamma) \vdash (\text{None}, \sigma, \gamma, \mathfrak{u}) \xrightarrow{\mu(l_1 \leftarrow l_2.T::m(e))} (\text{xcpt}, \sigma[l_1 \mapsto \sigma'(\texttt{result})], \gamma', \mathfrak{u}')$$

because $\mu(l_1 \leftarrow l_2.T::m(e)) = \mu(l_1 \leftarrow l_2.T::\mathit{new\_sig}(e))$ (equation (4.1) is used tacitly as before)

The critical step is to show the required equivalence between $\mu(\text{body}(\Gamma, T, m))$ and $\text{body}(\mu(\Gamma), T, \mathit{new\_sig})$. What it means of course is that the lookup of the changed program results in the body of the original program, but transformed. This critical identity is what I am going to show next.

First of all, note that $m = \mathit{old\_sig}$. I am showing that

$$\mu(\text{body}(\Gamma, T, m)) = \text{body}(\mu(\Gamma), T, \mathit{new\_sig})$$

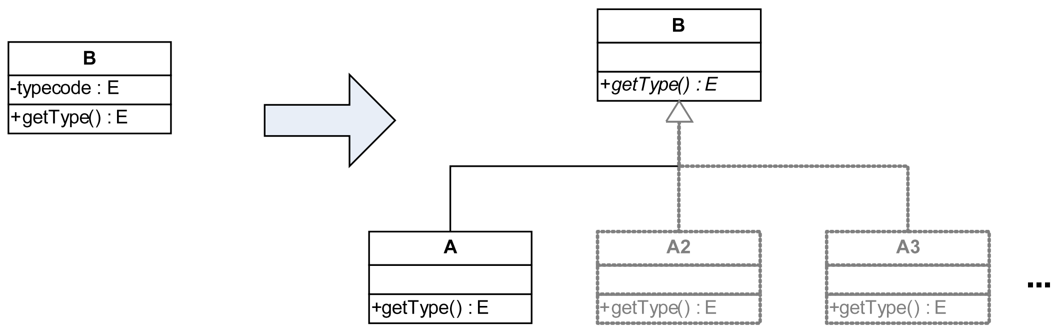
To check that equality assume that class B in the transformed program contains $\text{body}(\mu(\Gamma), T, m')$. It is then easy to verify that class B in the original program contains $\text{body}(\Gamma, T, m)$, i.e.,

$$\begin{aligned} \text{body}(\mu(\Gamma), T, \mathit{new\_sig}) &= \Gamma'[B.\text{methods}.\mathit{new\_sig}] \\ \text{body}(\Gamma, T, m) &= \Gamma[B.\text{methods}.m] \end{aligned} \tag{4.2}$$

Moreover, $(B, m) \in \mathcal{S}$. Now recall the relevant part of the transformation:

$$\Gamma[B.\text{methods}.\mathit{new\_sig} := \Gamma.B.\text{methods}.m]$$

The left-hand side of the assignment is just what is looked up in the transformed program, which yields the new equation when combined with both lines in equation (4.2).



4.2. “Replace Type Code with Subclass”

4.2.1. “Replace Type Code” explained

A field in the class determines the behavior of objects in a discrete number of ways. This field is constant and it identifies the “kind” of the object. Examples could include the “type” identifier of ASTs,[3] the kinds of commands in the command pattern (e.g., for undo/redo operations), the format used to store images in memory, etc. For all the possible values of the field, a subclass is created and the type-code is removed. The class cannot have subclasses before the refactoring. If it has, “Replace Type Code with Strategy” [16, p. 227] has to be used.

4.2.2. Applicability

The Replace Type with Subclass refactoring [16, p. 223] is a “scaffolding move” to enable more complete refactorings that replace switch statements (“bad smell switch statement”).

This refactoring makes obvious what is imminent in other refactorings as well:[4] The level of abstraction does not suggest itself unambigously.

For “Replace Type Code with Subclass”, some questions are: Should the subclasses already exist? How is the typecode initialized? Is the type code allowed to change?

A very generic solution could be subject to unrealistic restrictions that invalidate the attempted generality and make the resulting investigation more cumbersome than necessary. As development by application of transformation patterns like refactorings becomes more common, it will have to be left up to the programmer to choose the appropriate level of flexibility.

There is still always a spectrum of possibilities. Extremely flexible refactorings can be applied to practically all programs. They do not ensure many invariants. Very

[3]The enumeration `System.Compiler.NodeType` is an example discussed in chapter 6
[4]Including section 5.4 “Move Field”

strict refactorings can be applied to a narrow range of code patterns but guarantee useful invariants that simplify the postconditions. It is thus at least necessary that any program that fulfills minimal requirements can be transformed into a program that can be subject to the refactoring as presented in this text where the entities in the code retain their function. For the “Move Field” refactoring for instance, this criterion means that we must not introduce a fresh class that serves as a container for the moved field.

It is better to split refactorings into a few structurally simple *auxiliary* refactorings that have weak requirements and some refactorings that perform heavy-weight structural modifications. The auxiliary refactorings must be used to make the program conform with the syntactic[5] requirements of structural refactorings. You could well demand a more flexible refactoring, but it seems for this refactoring that a simple solution is more rewarding to discuss.

4.2.3. The refactoring

“Replace Type Code with Subclasses” is a structural refactoring. As such, it expects a specific code pattern. As explained above, we can easily and pragmatically assume that the type code is a private field that does not change after its initial assignment (“**`final`**”). This is a *precondition* of the refactoring. The refactoring could introduce all the type-classes at once and it would still be local.

Alternatively, we could allow stepwise transformation as suggested by Fowler [16]. Stepwise transformation is easier to describe and more elementary but it is more difficult to state concrete syntactic criteria that identify the appropriate situations in which the refactoring should be applied. I consider this a problem of identifying refactoring opportunities that is beyond the scope of this text. The refactoring will therefore be described as a stepwise procedure.

Here is a code example that roughly shows the pattern and how it is transformed by the refactoring.

[5]Such requirements can also be semantic, but they have to be easy to check. If they aren’t, they should be formulated as postconditions.

```
public class B {
    static B create(
      E type,
      Rest rest){
        result = new B();
        result.type = type;
    }

    private E type;
    final E getType(){
        result = type;
    }
}
```

$\Rightarrow$

```
class A extends B{
    E getType(){
      result = AE; }
}

public class B {
    static B create(
      BV type,
      Rest rest)
    {
      if(type != AE){
        result = new B();
        result.typecode
          = type;
      }else{
        result = new A();
        result.typecode
          = AE;
      }
    }

    private E type;
    E getType(){
      result = type; }
}
```

4.2.4. Expected structure

Let's call the class that contains the typecode encoded in its field B. B is the variant class. B's typecode field is *typecode*. Its type is E, which can be an integer without losing generality. It is set in B's static *create* (factory) method that takes arbitrary parameters. The *create* method is expected to have the following structure before the refactoring:

```
1: $S_1$
2: if($cond$){
3:   $S_2$ //must assign result, result.$typecode \neq A_E$
4: }else{
5:   result = new $B$();
6:   result.$typecode$ = $A_E$;
7: }
8: $S_3$ // Must not alter result, result.$typecode$
```

As you can see, the pattern specifies semantic properties. It is easier to formulate complex conditions semantically first and then relate them to known analyses (for instance compiler analyses).[6]

$B.typecode$ is written exactly once: In the $B.create$. For the refactoring, we want to turn the B objects with typecode value A_E into A objects. The typecode is only read through the getter method $getType$. Formulating the transformation $\mu^{\text{Replace typecode with subclasses}}_{B,typecode,getType,A,A_E}$ is quite challenging and not helping the understanding of the refactoring. For the sake of conciseness and understandability, it is worth trying to find a different formulation for the transformation that can also be used for specifying the preconditions.

4.2.5. Formalizing “Replace Typecode by Subclass”

After this informal exposition of the preconditions, I give a formal definition of the same conditions using the abbreviations in chapter 3 to illustrate that they are suitable for practical specification of preconditions and transformations.

Preconditions (essential applicability conditions) To keep the precondition more readable, I identify the name of the creation procedure *create* with its path in the program $\mathrm{metaclass}(B).\mathrm{methods}.(create, sig)$.

First the purely syntactic ones

$$
\begin{aligned}
&\Gamma[\quad \textit{The merly syntactic ones first}\\
&\quad B.\mathrm{fields}.typecode = E \quad \textit{“B's typecode field is typecode : E”}\\
&\wedge\ B.\mathrm{methods}.(getType, []) = ([], E, \texttt{result} \leftarrow \texttt{this}.typecode) \quad \textit{getType's simple definition}\\
&\wedge\ create = (p, B, b) \quad \textit{Declaration of create}\\
&\wedge\ b = S_1;\texttt{if}(cond)\{S_2\}\texttt{else}\{S_r\};S_3 \quad \textit{Implementation of create}\\
&\wedge\ S_r = \texttt{result} \leftarrow \texttt{new}\ B;\texttt{result}.typecode \leftarrow A_E\\
&\wedge\ \mathrm{nodecount}(*.\_ \leftarrow \texttt{new}\ B) = 1\\
&\quad]
\end{aligned}
$$

[6] Why I am not using postconditions instead to express these conditions? Here is a recapitulation of the reason from the introduction: Local properties are often unrelated to architectural decisions. Invariants and specifications are most important for architectural decisions that are non-local and non-trivial, i.e., at the join points of different components like non-private instance variables and method or class interfaces. The semantic conditions above are non-trivial, but they are local and certainly not linked to any architectural decision. The programmer will be able to bring his program into this shape.

$$\text{\textit{S_1 and S_3 do not allocate B objects.}}$$
$$\wedge\, \forall \Gamma', \gamma, \gamma' : \Gamma' \vdash (\_,\_,\gamma',\_) \xrightarrow{S_1} (\_,\_,\gamma',\_) \Rightarrow \forall \mathit{loc} \notin \operatorname{dom}\gamma \Rightarrow \operatorname{rtt}(\gamma'(\mathit{loc})) \not\preceq_\Gamma B$$

$$\text{\textit{S_2 sets \texttt{result} and \texttt{result}.typecode to something else than A_E.}}$$
$$\wedge\, \forall \Gamma', s, s', t : \Gamma' \vdash s \xrightarrow{S_2} s' \equiv (\mathrm{None}, \sigma', \gamma', \mathfrak{u}') \Rightarrow t = \gamma'(\sigma'(\texttt{result}))(\mathit{typecode}) \wedge t \neq A_E$$

$$\text{\textit{S_3 leaves \texttt{result}, \texttt{result}.typecode unchanged}}$$
$$\begin{aligned}&\wedge\, (\forall \Gamma', s, s', r : \Gamma' \vdash s \equiv (\_, \sigma, \gamma, \_) \xrightarrow{S_3} s' \equiv (\mathrm{None}, \sigma', \gamma', \mathfrak{u}')\\ &\quad \Rightarrow r = \sigma(\texttt{result}) = \sigma'(\texttt{result}) \wedge\ \gamma(r)(\mathit{typecode}) = \gamma'(r)(\mathit{typecode}))\end{aligned}$$

$$\text{\textit{The rest of the program leaves \texttt{result}.typecode unchanged}}$$
$$\begin{aligned}&\wedge\, (\forall \Gamma', t \not\supseteq \mathit{create}, s', r : \Gamma' \vdash s \equiv (\_, \sigma, \gamma, \_) \xrightarrow{S_3} s' \equiv (\mathrm{None}, \sigma', \gamma', \mathfrak{u}')\\ &\quad \Rightarrow \forall r : \gamma(r)(\mathit{typecode}) = \gamma'(r)(\mathit{typecode}))\end{aligned}$$

Syntactic approximations for the semantic preconditions Semantic conditions as preconditions are unacceptable. I list here some possibilities to approximate them syntactically. The legitimacy of the restrictions I present could seem drastic and indiscriminate. Keep in mind however that *create* is already fairly constrained anyway. I deemed the additional responsibilities the syntactic approximations create acceptable.

Transformation Let's introduce abbreviation for *create*'s body:

$$b(S) \equiv S_1;\texttt{if}(\mathit{cond})\{S_2\}\texttt{else}\{S;\texttt{result}.\mathit{typecode}{\leftarrow}A_E\};S_3$$

The transformation is

$$\begin{aligned}&\Gamma[\text{if } \operatorname{metaclass}(B).\operatorname{methods}.(\mathit{create}, \mathit{sig}) = (p, B, b(\texttt{result} \leftarrow \texttt{new } B)) \text{ then}\\ &\quad \operatorname{metaclass}(B).\operatorname{methods}.(\mathit{create}, \mathit{sig}) := (p, B, b(\texttt{result} \leftarrow \texttt{new } A))\\ &\quad A := \mathit{emptyCls}[\operatorname{methods}.(\mathit{getType}, []) := ([], E, \texttt{result} \leftarrow A_E)]\\ &\ ]\end{aligned}$$

Semantic goal	Approximation
Guarantees that S_1 and S_3 do not allocate B objects.	No allocations of objects of type B or one of its subtypes outside the **`if`** statement. In Java, this could mean making the constructor private and checking that it is not used outside *create*.[7]
Guarantee that S_2 sets these values such that they do not interfere with the criteria for subclass B.	Definite assignment of `result`, `result`.*typecode* in S_2 (definite assignment to `result`.*typecode* guarantees `result` $\neq$ Null) *typecode* must not be A_E. This can be checked with simple copy propagation and only allowing assignments with known values.
Correctness of the *typecode* field.	Strict no aliasing for `result` before S_3 (e.g., no right-hand side occurences and no use as parameter) and no assignment to `result`.*typecode* in S_3 and the program outside *create*.

4.2.6. State correspondence

State correspondence is able to express directly the intuition behind the refactoring pattern: All objects of type B with typecode A_E must become A objects in the transformed version. The exception state and the local variables do not change at all, so external behavior will be retained. For individual heap values, it is easy to formulate $\beta^{\text{Replace Typecode with Subclass}}$ as a function that is bijective:

$$\beta(T, v) = \begin{cases} (T, v) & \text{if } v(B{::}\mathit{typecode}) \neq A_E \ (v(B{::}\mathit{typecode}) \text{ may be undefined}) \\ (A, v) & \text{if } v(B{::}\mathit{typecode}) = A_E \end{cases} \tag{4.3}$$

4.2.7. Equivalence

Let’s check the crucial invariant before progressing to the actual proof. If $\mathit{typecode} = A_E$, the type must be *equal to* B, not a subclass thereof, i.e.,

Lemma 4.2. For all S in Γ, including the body of *create* but excluding its constituents

$$\Gamma \vdash (\_, \_, \gamma, \_) \xrightarrow{S} (\_, \_, \gamma', \_) \Rightarrow \forall loc \notin \operatorname{dom}\gamma : \gamma'(loc)(B{::}\mathit{typecode}) = A_E \Rightarrow \operatorname{rtt}(\gamma'(loc)) = B$$

This will turn out to be the only non-trivial requirement, so I invest some care. First of all, the fact that $v(B{::}\mathit{typecode})$ is defined implies $v \preceq_\Gamma B$. It also helps to observe that the runtime type of a location in the heap does not change, i.e., for every S in Γ outside *create*, it holds that

$$\Gamma \vdash (\_,\_,\gamma,\_) \xrightarrow{S} (\_,\_,\gamma',\_) \Rightarrow \mathrm{rtt}(\gamma(loc)) = \mathrm{rtt}(\gamma'(loc))$$

This can be seen by careful examination of the operational semantics: For all rules and all locations *loc*, if there is a transition from heap γ to γ', the type of all defined locations remains the same.[8] This does only hold for statements that are outside *create*.

B objects are only allocated within *create*'s body b and their *typecode* field is only set within that method. If it is guaranteed that after its execution, the desired invariant holds, then the invariant holds for all programs. This argument could again be formalized inductively with a case distinction on whether *create* is called or not for method invocations.

It remains to show that *create establishes* the required invariant for new objects:

$$\Gamma \vdash (\_,\_,\gamma,\_) \xrightarrow{b} (\_,\_,\gamma',\_) \Rightarrow \forall loc \notin \mathrm{dom}\,\gamma : \gamma'(loc)(B::typecode) = A_E \Rightarrow \mathrm{rtt}(\gamma'(loc)) = B \tag{4.4}$$

For *create*'s body b, we introduce the following *abbreviations*

$$b = \underbrace{\overbrace{S_1}^{t_{1a}}\; ; \overbrace{\texttt{if}(cond)\{S_2\}\texttt{else}\{\,\texttt{result}\leftarrow\texttt{new}\ B; \texttt{result}\,.typecode\leftarrow A_E\}}^{t_{1b}}}_{t_1} ; \underbrace{S_3}_{t_2}$$

Let's now consider the shape of an arbitrary execution. At the top level, b is a chaining of different statements. A derivation tree must therefore have the following shape:

$$\dfrac{\dfrac{\Gamma \vdash s_0 \xrightarrow{t_{1a}} s_{1a} \qquad \Gamma \vdash s_{1a} \xrightarrow{t_{1b}} s_1}{\Gamma \vdash s_0 \xrightarrow{t_{1a};t_{1b}} s_1} \qquad \Gamma \vdash s_1 \xrightarrow{t_2} s_2}{\Gamma \vdash s_0 \xrightarrow{t_1;t_2} s_2}$$

Such a derivation must always exist: Expression evaluation is invariant because there is no way to determine whether a value is of exact type B or a subtype thereof. That's why the β defined above gives that guarantee. A formal proof works by induction on the structure of expressions.

Because the value of expressions remains the same, control flow does not change and the derivation does always exist.

If there was a way to tell A objects and B objects apart, for instance using Java's **.class** field, the derivation wouldn't be guaranteed. S_3 could loop for example.

```
while(result.class.getName().equals("A")) /* loop */;
```

[8]Formally, this is an inductive argument of course.

That simply means that additional constraints are necessary in practice like forbidding access to `getClass()`, `.class` in Java and `GetType()` in .NET. Another possibility would be to check that reflection is not used in conjunction with class B and its subclasses as postconditions.[9]

From the applicability conditions, we know that

$$\forall \gamma, \gamma' : \Gamma \vdash (\_, \_, \gamma, \_) \xrightarrow{t_{1a}} (\_, \_, \gamma', \_) \Rightarrow \forall loc \notin \operatorname{dom} \gamma \Rightarrow \operatorname{rtt}(\gamma'(loc)) \not\preceq_\Gamma B$$

That is, the left-hand side of equation (4.4), $\gamma'(loc)(B::typecode) = A_E$ is never satisfied.

For the **then** part (i.e., S_2) of the **if** statement, it is the same, because it is guaranteed – according to the semantic preconditions – that *typecode* is never A_E. The **else** part is fully determined, so it is easy to check that the invariant is indeed retained. t_2 does not invalidate it either because there are no allocations of B objects. The condition that `result` is not altered is only required for the intended semantics of the program, not for the correctness.

With this strong lemma at hand, it is easy to establish the equivalence theorem equation (3.16). To see why, notice that, in the proof sketches above, B can be replaced by A. The only place where code and propositions are linked is the body of *create*, more precisely, it is the `result` $\leftarrow$`new` B instruction. After the transformation it becomes `result` $\leftarrow$`new` A – matching the modified propositions we have proven already. Leading to the following lemma

Lemma 4.3. For all S in $\Gamma' = \mu(\Gamma)$, including the body of *create* but excluding its constituents

$$\Gamma \vdash (\_, \_, \gamma, \_) \xrightarrow{S} (\_, \_, \gamma', \_) \Rightarrow \forall loc \notin \operatorname{dom} \gamma : \gamma'(loc)(B::typecode) = A_E \Rightarrow \operatorname{rtt}(\gamma'(loc)) = A \tag{4.5}$$

This guarantees that newly allocated objects v with $v(B::typecode) = A_E$ have type A. Now let's go back and assume $(\Gamma \vdash s \xrightarrow{t} s') \wedge \beta(s, r)$ as in the equivalence definition equation (3.16). $r = \beta(s)$ means everything is the same except that heap objects in s of type B with their *typecode* field set to A_E are of type A in r. If we could show that B objects and A objects behave the same if $typecode = A_E$, $\Gamma \vdash r \xrightarrow{t} r'$ would be guaranteed immediately. This is indeed the case. The critical aspect is invocations to *getType*, which can be easily shown to behave identically to the implementation in B.[10]

[9] The problem with this approach is that reflection is very often used in conjunction with generic `Object`s or interfaces instead of values with more precise type information that could help eliminate some of the tests easily.

[10] This may raise the question why we did override *getType* in the first place. Needless to say, there is no clear answer. Imagine we override *getType*: After all typecodes are replaced with subclasses, we probably wanted to get rid of the *typecode* field that has now become superfluous if B is an abstract class anyway. The required transformation is anther refactoring, a variation of what is done here that should be included in a comprehensive refactoring catalogue. The refactoring couldn't draw from the extensive list of assumptions we rely on here. It would probably be cumbersome to apply and specify and mostly used in the context of “Replace Typecode with Subclass” anyway.

To finish the proof, equation (4.5) is used. It shows that newly allocated values are β compatible, i.e., $\beta(s', r')$.

4.2.8. Conclusion

The required correspondence is satisfied without any additional postconditions – even though I must admit that the catalogue of requirements for the transformation itself is quite extensive. Some semantic preconditions could be translated to postconditions instead of being approximated. The difficulty is still that many assumptions have to be made about the structure of the program to allow transformation in the first place. Some of these assumptions on the program structure could be replaced by more complex, purely semantic conditions. It would be interesting to see how this could simplify the treatment of the refactoring. The problem with this approach is that the necessary postconditions are difficult to check dynamically (or to verify them statically). The solution I've chosen here was more pragmatic. Only the adoption by a refactoring tool can show where such a formalization needs to be generalized.

Example: the body of *create* must not allocate B objects and set invalid typecodes.

4.3. "Extract Method"

This refactoring is used to factor out inline code. It is used to break long methods into smaller pieces. "Extract Method" is a local refactoring. It is well supported by existing refactoring browsers and alters the data space only locally. The variables that are set in the extracted blocks must be returned by the extracted method and must be set at the invocation site. The example in figure 4.1 is taken from chapter 2 and illustrates a limitation of the Java language: the only direct way variables at the invocation site can be set to variables in in the invokee is through return values.[11] In this case, the return value of the extracted method `mc()` corresponds to the variable `m` in the original program.

Relevance The formalization of this refactoring is reasonably interesting because it is the prime representative of *local state space modifications*, i.e., modifications that extend to a limited period of the program's execution inside a confined region of the program code. In our framework, local state space modifications can be handled because β may differ because of the path leading to a statement ($\beta_{p_0.p_1...p_n}$). This is vital because invocation frames are abstracted away by the big-step semantics we're using. Refactorings could be construed where the state space correspondence cannot be described in terms of the location in the program text (i.e., class and method name) and

[11]Out and in/out parameters are possible in .NET. The refactoring tool for C# that comes with Visual Studio exploits these capabilities.

```
String s(){
  double t = 0;
  String r = gN()+"\n";
  for (P x : a) {
    double m;
    switch(x.gP().c){
    case P.A:
      m = ...;
      break;
    // etc...
    default:
      fail();
    }
    double p =
      x.gP().s*m;

    r += "\t"+x.gP().t
      +"\t"+p+"\n";
    t += p;
  }
  r += t+"\n";
  return result;
}
```

$\Longrightarrow$

```
String s(){
  double t = 0;
  String r = gN()+"\n";
  for (P x : a) {
    double m = mc();
    double p =
      x.gP().s*m;

    r += "\t"+x.gP().t
      +"\t"+p+"\n";
    t += p;
  }
  r += t+"\n";
  return result;
}
static double mc(P x){
  double m;
  switch(x.gP().c){
  case P.A:
    m = ...;
    break;
  // etc...
  default:
    fail();
  }
  return m;
}
```

Figure 4.1.: Extract method (adapted from chapter 2)

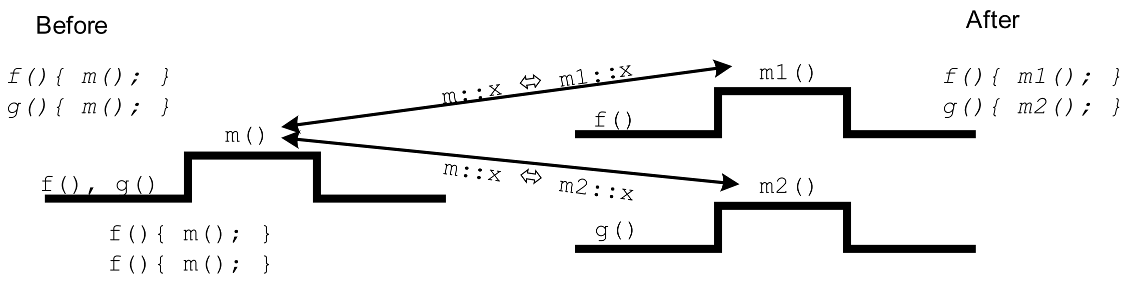


Figure 4.2.: Stack dependent state space correspondence

the technique used here would fail.[12] β would have to take into account the runtime stack of the program. This is not the case for practical refactorings that have been proposed.

For the formalization, I assume that there is exactly one variable that is used by the source method after the invocation. Reduction to no variable is trivial, extension to more than one value that is passed from the extracted method must be handled by turning local variables into objects, i.e., apply "Turn Locals into Object", a micro-refactoring briefly discussed in section 4.11. The refactoring as presented here may result in unassigned variables being passed to the method.[13] This is not syntactically valid in proper Java but allowed by the operational semantics. Such variables could also be declared inside the invoked method. Even equivalence can be proven easily because it is known that their values are *undef*.

Preconditions The code to be extracted is in method f of class F. It is to be extracted into a new *static* method with name t in class T. t can be turned into an instance method in a second step by the micro-refactoring As the extracted method is always static, it can be turned into an instance method by "Make Method Instance-Bound" (section 4.10). in a second step. Additional contraints have to hold if the method is accessing non-public fields: $F = T$, $T \preceq_\Gamma F$ or the classes have to be in the same package (depending on whether the fields are "**`private`**", default visibility or "**`protected`**").

[12]Examples are absolutely unrealistic and so is the following: Imagine I want to split a method `m` with two invocation sites – in one step – so that a different method is called from each site. For the sake of clarity, assume the two invocations are in different methods `f` and `g`. It is obvious that β-correspondence for any local variable x cannot be formulated if no information is available about the invocation site: If `m` was called from `f`, $\beta(C::m::x) = C::m1::x$ and $\beta(C::m::x) = C::m2::x$ if called from `g` (see figure 4.2).

If you think this is an artefact from qualifying the local variables – which is just a way of hiding the paths that qualify β, just consider the case when you want to rename the variables as well. As I contended before, the fact that it is a composite refactoring does not give it a special status – whether or not a refactoring is composite depends on the refactorings you consider primitive.

[13]This is so because I do not distinguish between

For the refactoring, I assume that **`this`** as well as **`result`** is properly renamed. This is the only source of complication for the refactoring.

The method $F{::}f$ consists of the statements S_1, S and S_3.[14] *S is the statement to extract.* It takes formal parameters P_f and returns a value of type R_f. P_f^r are the types of the parameters. I.e.,

Method body containing statement to be extracted is $S_1;S;S_3$. Parameters are P_f, return type is R_f.

$$\Gamma[F.\mathrm{methods}.(f, P_f^r) = (P_f, R_f, S_1;S;S_3)]$$

Statement to extract is S

E_r denotes the set of local variables and parameters S is (potentially) using. E_w is the set of locals S is writing and are retained beyond S. I.e., E_r will be the parameters of the newly extracted method and E_w contains the variable the will be returned as a result. In this simplified setting, this set must consists of a single element: $\{r\} = E_w$. Note that variables that are local to a block of statements are not in E_w, independent of whether they are written or not. A more general incarnation would allow variables in E_w that are written but whose value is never read after the execution of S.

E_r are the input variables, E_w are the output variables of S

The heavly constrained structure of the enclosing method now helps define E_r and E_w. I am using the following abbreviations: $\mathit{read}(I) = \bigcup\Gamma[\mathsf{if}\ t = I.\_{\leftarrow}e\ \mathsf{then}\ \mathrm{FV}(e)]$ denotes the set of local variables that are read in statement I. $\mathit{written}(I) = \Gamma[\mathsf{if}\ t = I.l{\leftarrow}\_\ \mathsf{then}\ l]$ is the set of local variables written in I.

E_w are the variables read in S_3 and written in S. E_r are the variables read in S and written in S_1:

$$\begin{aligned} E_r &= \mathit{read}(S) \cap \mathit{written}(S_1) \\ E_w &= \mathit{read}(S_3) \cap \mathit{written}(S) \end{aligned}$$

These definitions do not take into account the possibility of write-before-write accesses: A variable could be written by S_3 before being read in S_3 but *after* being written in S. Alternatively, it could be written in S before being read in S but after being written in S_1.

Definition of E_r and E_w simple, but too conservative

Transformation The body $S_1;S;S_3$ of f has to be replaced by the invocation to the newly created method t in metaclass T, i.e., $r{\leftarrow}T.T{::}t(E_r)$. The newly created method takes all the variables read by S as formal arguments. We refer to the formal argument list as well as the name and types list by E_r to economize on symbol names.

[14] A more flexible approach is taken with the inverse refactoring in the next section on page 95. The formal treatment remains the same.

The special variables and **this** and **result** have to be renamed because they are already used by the extracted method. The renaming is done by piecewise application of

$$\text{ren}_{\text{“Extract Method”}} = V \mapsto \begin{cases} \text{metaclass}\, T{::}t{::}X' & \text{if } V = F{::}f{::}X \\ & X' = X \text{ if } V \notin \{\texttt{this}, \texttt{result}\} \\ & X' = \text{unique id otherwise} \\ V & \text{otherwise} \end{cases}$$

Using ren, the transformation can be expressed as follows. As always, static methods are declared as instance methods of the corresponding class objects.

$$\begin{aligned} &\Gamma[\\ &\quad F.\,\text{methods}\,.(f, P_f^r).\,\text{body} := S_1; r \leftarrow T.T{::}t(E_r); S_3 \\ &\quad \text{metaclass}(T).\,\text{methods}\,.(t, \text{ren}\, E_r) := (\text{ctt}\, r, \text{ren}\, E_r, \text{ren}\, S; \mathit{result} \leftarrow \text{ren}\, r)] \end{aligned}$$

Data correspondence β is the identity outside S and maps between variables in $F{::}f$ and metaclass $T{::}t$, i.e., $\beta(F{::}f{::}x) = \text{metaclass}\, T{::}t{::}\,\text{ren}\, x$ inside S for all $x \in (E_r \cup E_w)$ and $\beta = \text{id}$ otherwise. It is absolutely vital to confine β to $(E_r \cup E_w)$ because only the variables that are read and written and consequently returned are set at all in the extracted method. There may well be variables that survive the call of the extracted method without being used there.

$$\beta_{[q,\mu(q)]}(X) = \begin{cases} \text{ren}(X) & \text{if } q \supseteq S \quad \text{and } X \in (E_r \cup E_w) \\ X & \text{otherwise} \end{cases}$$

β leaves the heap and the exception state unaffected.

Equivalence This refactoring is a local transformation and it is sufficient to show that

$$(\Gamma \vdash s \xrightarrow{S} s') \wedge \beta(s, r) \Rightarrow (\mu(\Gamma) \vdash r \xrightarrow{r \leftarrow T.T::t(E_r)} r') \wedge \beta(s', r')$$

The assignments due to the parameter passing restore the relevant part of the environment in $T{::}t$ to that in $F{::}f$. The aspect of the proof that is not entirely trivial is that the environments are only equivalent with respect to the variables that are read (i.e., $E_r \cup E_w$) but not the variables that are not read. Luckily, this has already been taken care of by the definition of β.

4.4. “Inline Method”

“Inline Method” is a refactoring that is most well known as a compiler optimization – see [34] for instance. For composite refactorings however, it plays an important role when redistributing the code incrementally to different old and new methods as illustrated in chapter 2 and figure 4.4. “Inline Method” is not only theoretically the inverse of “Extract Method”. What this means is that “Inline Method” could be carefully described and “Extract Method” could be covered as the derived reverse refactoring only.

“Inline Method” is also a local refactoring and does not have any postconditions necessary for correctness preserving execution.

It is possible to inline method calls whose implementing method is not known by case distinction on the runtime type of the receiver. This is the idea of dynamic dispatch. In this section, I’ll cover only the most simple case where the implementing method is known – the possible reasons of which I am leaving completely unspecified.

Preconditions Let the method call we want to inline be identified by path i. For the sake of explicitness, i is an invoke special in Γ, i.e., $\Gamma[i = r \leftarrow y.Q{::}m(E)]$. The case for invoke virtual is identical (with $\mathrm{rtt}(y)$ instead of Q) as for all proofs that do not specifically depend on the runtime type of the object possibly differing between executions of the statement, i.e., if dynamic dispatch can be resolved statically or not is irrelevant for that matter.

Definitions r, y, Q, m, E

In this case, the body the lookup resolves to must be constant: $b = \mathrm{body}(\Gamma, Q, m)$ for all possible Q in case of virtual invocation. I.e., either a single, special Q in case of invoke-special or all sub-classes and implementations of $\mathrm{ctt}(y)$. The implementing method is $T{::}t$ and the method that contains the invocation statement i is $F{::}f$. Note that it is immaterial if $Q{::}m$ is static or not – in case of a static method, Q is some class object, i.e., $Q = \mathrm{metaclass}(\_)$).

Definition b

Definitions $T{::}t$ and $F{::}f$

$$\begin{array}{l}\Gamma[i = r \leftarrow y.Q{::}m(E)],\ Q \text{ variable}\\ \quad b = \mathrm{body}(\Gamma, Q, m) \text{ for all possible } Q\end{array}$$

The symbols are illustrated in figure 4.3.

```
class F{
  ...
  _ f(_){
    ...
    r←y.Q::m(E) ≡ i
    ...
  }
  ...
}

class T{
  ...
  ctt(r)  t(E) {
    b
  }
  ...
}
```

⇒

```
class F{
  ...
  _ f(_){
    ...
    ren b
    ...
  }
  ...
}
```

Figure 4.3.: Variable names for “Inline Method”

```
private double f(P x){
  return g(x);
}
private double g(P x) {
    Product p = x.m();
    return this.q(p);
}
```

⇒

```
private double f(P x){
    Product p = x.m();
    return this.q(p);
}
```

Figure 4.4.: Simple inline method for eliminating unneeded indirections

Transformation The transformation itself is specified as if variable names were qualified by their enclosing method and class. It actually depends on the subtree inside which the variable is to be renamed.

$$\text{ren}_{\text{“Inline Method”}} = V \mapsto \begin{cases} F{::}f{::}X' & \text{if } V = T{::}t{::}X \\ & X' = y \text{ if } X = \texttt{this} \\ & X' = r \text{ if } X = \texttt{result} \\ & X' = \text{generated unique identifier otherwise} \\ V & \text{otherwise} \end{cases}$$

The expansion is then easy to formulate in terms of ren.

$$\Gamma[i := \text{ren}\, b]$$

Correspondence Inline method is interesting because it illustrates that the mapping between corresponding program parts, which is denoted by μ, just as the transformation, may be neither bijective nor injective for a simple refactoring. This means that β cannot be parameterized by only the path to the statement in the original program but *necessarily* has to inlclude the corresponding part in the transformed program. I have given an example of an “Inline Method” refactoring where the data mapping necessarily depends on the runtime stack in figure 4.2. There are no practical refactorings of this sort that cannot be *easily* decomposed to refactorings with simpler characteristics – including “Inline Method” that is used to “unwind” the stack to the necessary level.

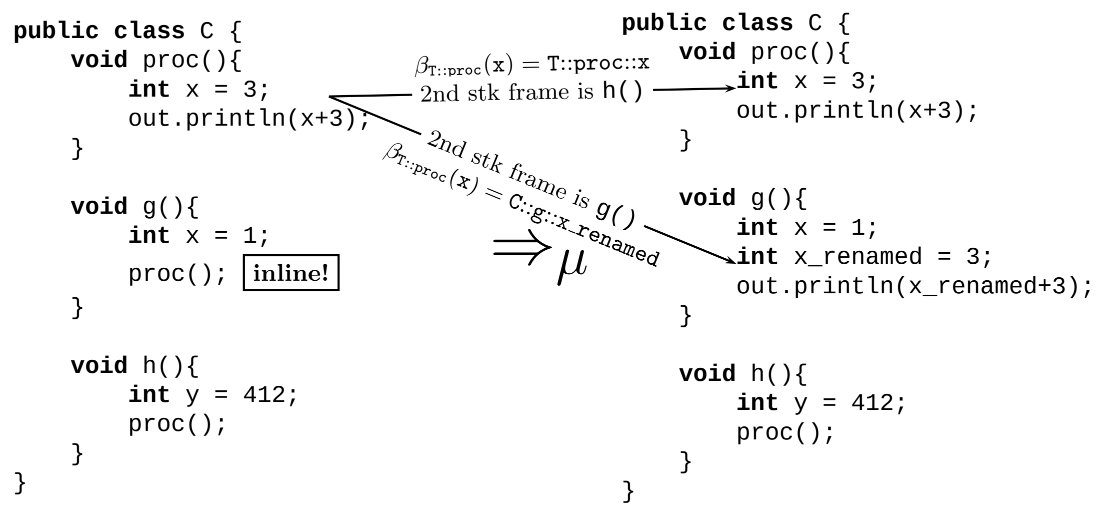
“Inline Method” is a local refactoring, so the data correspondence is id almost everywhere. Just as for “Extract Method”, β is id at the refactoring site.

$$\beta_{[q,q']} = \begin{cases} \text{ren} & \text{if } q \supset b \text{ and } q' \supset i \\ \text{id} & \text{if } q = q' \end{cases}$$

Correctness The proof for “Inline Method” is analogous to the one in “Extract Method”: No runtime checks are needed.

4.5. “Replace Representation”

“Replace Representation” could be formulated in a very general manner. This is not what I want to attempt in this mini-section. Instead, I want to provide a pragmatic solution that is often sufficient and is well suited as a scaffolding step as illustrated in chapter 2 for example for “Replace Type Tests with Dynamic Dispatch”. “Replace



```
public class C {
    void proc(){
        int x = 3;
        out.println(x+3);
    }

    void g(){
        int x = 1;
        proc();
    }

    void h(){
        int y = 412;
        proc();
    }
}
```

```
public class C {
    void proc(){
        int x = 3;
        out.println(x+3);
    }

    void g(){
        int x = 1;
        int x_renamed = 3;
        out.println(x_renamed+3);
    }

    void h(){
        int y = 412;
        proc();
    }
}
```

inline!
$\beta_{\texttt{T::proc}}(\texttt{x}) = \texttt{T::proc::x}$
2nd stk frame is h()
2nd stk frame is g()
$\beta_{\texttt{T::proc}}(x) = \texttt{C::g::x\_renamed}$
$\Rightarrow \mu$

Figure 4.5.: “Inline Method” leads to non-injective PC mapping

Representation” as discussed here is about replacing the *value* of a single variable, not the *logic* abstraction this value may encapsulate as in the case of a linked list vs. an array based list for instance.

The idea is again to confine the application of the refactoring. Just as “Rename Field”, the present refactoring is a data refactoring and it is not in general beneficial to avoid changing all occurences of the data abstraction. It is however possible to shield other data abstractions from the change. This is done by introducing translation functions *inside* the program in a global class. These translation functions must correspond to the specified correspondences that are given as input to the refactoring tool. The translation functions can easily be synthesized so as to be correct by construction.

Every write access uses the forward translation function, every read access uses the reverse translation function. In the example in figure 4.6 as well as the formalization below, representation of field `T::f` is changed from type `X` and corresponding set $X \subset [\![\texttt{X}]\!]$ to type `Y` and set $Y \subset [\![\texttt{Y}]\!]$. The translation function $M : X \to Y$ is encoded as `Globals::forwardM` its inverse M^{-1} is encoded as `Globals::backwardM`

For the sake of completeness, I want to present a formal definition and a brief discussion of correctness for this section as well even though it is a bit less rewarding than the ones treated before.

Preconditions (essential applicability) The only precondition is that there is a field f of type `X` in class T.[15] All other conditions are more aptly formulated as postcondi-

[15]I write `X` for the name in the program to distinguish it from X, which is the subset it is approximating even though `X` is also variable and need not be called `X`.

```
class T{
  X f;
}

// example
T t = new T();

// write access
t.f = $E_1$;
// read access
X tt = $E_2$.f;
```

```
class T{
  Y f;
}

public class Globals{
  // assume $X$ is finite
  static Y forwardM(X t){
    if(t == $C_1$)
      return $M(C_1)$;
    ...
    else assert(false);
  }

  static X backwardM(Y t){
    if(t == $M(C_1)$)
      return $C_1$;
    ...
    else assert(false);
  }
}
// example
T t = new T();

// write access
t.f = Globals.forwardM($E_1$);
// read access
X tt = Globals.backwardM($E_2$.f);
```

Figure 4.6.: Pattern for “Replace Representation”

tions.

$$\Gamma[T.\operatorname{fields}.f = \mathtt{X}]$$

The presence of the translation methods, which I call *forwardM* and *backwardM*, just as in the example, is not assumed. They are produced as part of the transformation.

There is another correctness constraint I introduce here to keep the data correspondence simple: compatibility of initial values. The default value of field f must correspond before and after the transformation. $M(z_{\mathtt{X}}) = z_{\mathtt{Y}}$ if the default value of X is $z_{\mathtt{X}}$ and the default value of Y is called $z_{\mathtt{Y}}$. To avoid this quite significant restriction, the data correspondence can be weakened in much the same way as in chapter 5.

Data correspondence "Replace Representation" only changes individual statements. All original statements are comparable, $\beta_{[t,t]}$ is defined for all paths t in Γ. For those statements, β trivially maps the value of field f. Let (Q, w) be any heap value of type T, i.e., $Q \preceq_{\Gamma} T$. Field $f \in X$ is then mapped to the corresponding value in Y: $\beta(Q, w) = (Q, w[T{::}f \mapsto M(w(T{::}f))])$

Additional correctness conditions Whatever value v field f is set to in the original program, v must be in X. When v is read, it must also be in X. This can be translated to postconditions using the β correspondence. To make sure that f is set to a valid value, an assertion has to be added before the assignment. To make sure that the value that is read from f is valid, an assertion has to be added after the field access.

If you think the second check is not necessary, you're right. β guarantees that f contains only valid values anyway. Yet, I want to keep it because it gives me the opportunity to reiterate two concepts of the formalism in a simple setting. Firstly, the read assertion is in the resulting program, but the condition it verifies is formulated in terms of the original program: "x must be in X" must be translated to "x must be in Y". The condition is translated to the abstractions of the refactored program. Secondly, it could be argued that the resulting program aborts when the condition is not met at execution time. That's absolutely true in practice, but it does not have anything to do with the formal treatment. Pre- and postconditions are properties of the original program Γ and the resulting program $\mu(\Gamma)$. Assertion failure at runtime means that the postconditions were not satisfied, i.e., the conditions that are assumed for every correctness proof are not given. There is no `assert`(E) statement in the language!

Transformation The first step is certainly to change the type of f:

$$\Gamma[T.\operatorname{fields}.f := Y]$$

Then we have to produce the translation functions *forwardM* and *backwardM* are unique names that not present in the program. Depending on the exact implementation, it would also be necessary to produce additional helpers that test whether a value is contained in the sets X and Y or not. I omit them for clarity.

I put the translation functions into T as static methods to avoid having to come up with a separate class for them and I call the parameters `t`.

$$\begin{aligned}\Gamma[&\mathrm{metaclass}(T).\,\mathrm{methods}\,.(\mathit{forwardM}, X) := ([(t, X)], Y, \mathit{encode}(M)),\\ &\mathrm{metaclass}(T).\,\mathrm{methods}\,.(\mathit{backwardM}, Y) := ([(t, Y)], X, \mathit{encode}(M^{-1}))]\end{aligned}$$

I am using the helper function *encode* to produce a program representation of a finite function. One possible definition could be

$$\mathit{encode}(M) = \begin{cases} \texttt{if}(t{=}{=}x)\{\,\texttt{result}\leftarrow y\}\texttt{else}\{\mathit{encode}(M - \{x \mapsto y\})\} & \text{if } \{x \mapsto y\} \in M \\ \texttt{assert}(\,\mathrm{false}\,) & \text{if } M = \{\} \end{cases}$$

The code faithfully represents the given function. This has to be proven by case distinction on the element passed to the function.

It is also necessary to modify read and write accesses to the field: I do need additional temporary variables. I write them as temp_1 and temp_2 even though they have to be understood as injective functions from program point to unique name.

$$\begin{aligned}&\Gamma[\mathsf{if}\ t = *.e_1.T{::}f\leftarrow e_2\ \mathsf{then}\\ &\quad t := \texttt{assert}(e_2 \in X);\mathit{temp}_1\leftarrow T.\,\mathrm{metaclass}(T){::}\mathit{forwardM}(e_2);e_1.T{::}f\leftarrow \mathit{temp}_1\\ &\quad][\mathsf{if}\ t = *.l\leftarrow e.T{::}f\ \mathsf{then}]\\ &\quad t := \mathit{temp}_2\leftarrow e.T{::}f;\texttt{assert}(\mathit{temp}_2 \in Y);l\leftarrow T.\,\mathrm{metaclass}(T){::}\mathit{backwardM}(\mathit{temp}_2)]\end{aligned}$$

Correctness The correctness of "Replace Representation" is almost guaranteed by the definitions. The only crucial statements are object allocation where precondition guarantees that the poststate is compatible and field updates with expressions. The assumed validity of the assertion guarantees the property that make sure that β correspondence is retained, i.e., that the newly set Y value has a corresponding value in X.

4.6. "Replace Expression"

Replace expression whenever one expression (without side-effects as usual in the subset covered here) E is equivalent to a different expression F for the given preconditions of the application. Assertions can be used to test equivalence of F and E. Even if checked dynamically, these assertions are not a performance issue because evaluation of expressions is a constant time operation in our language.

The refactoring becomes more useful if more general expressions or *calculations* with side effects are allowed as well. “Replace Expression” then becomes “Replace Algorithm”. Testing equivalence for different algorithms modularly such that it does not affect the outcome of subsequent I/O operations is not something I would want to attempt as part of this Thesis. Here is a exposition of the simplest version where an expression e is replaced by e':

Conditions The expression to be replaced is the right hand side e of an assignment to a local variable l that is identified by i, i.e.,

$$\Gamma[i = l{\leftarrow}e]$$

The other non-syntactic condition, namely that e and e' are identical is less trivial. I can definitely not expect e and e' to be equivalent.[16] All I need is equivalence for all values at which I evaluate e.

This is exactly what can be tested by an assertion. Assertions have to be conservative approximations of conditions, not vice versa. The condition has to be at least as weak. weak as the condition that is tested by the assertion. Only then I can be sure that the program will run correctly.

Data correspondence If the resulting expressions are indeed identical, nothing should change in the state space of the program, i.e., $\beta[t, t] = \mathrm{id}$

Transformation The expressions are compared just before the assignment:

$$\Gamma[i := \texttt{assert}(e = e');l{\leftarrow}e']$$

Correctness Correctness for this refactoring requires postconditions and yet it is so simple that I definitely want to discuss it in great detail.

β is identity. The relevant correctness notion is equation (3.18). All statements t are comparable, so it has to be shown that $\forall t, s, s' : \Gamma \vdash s \xrightarrow{t} s' \Rightarrow \mu(\Gamma) \vdash s \xrightarrow{t} s'$. Most statements are unaffected by the transformation, so equivalence is trivial for them. Let's consider the case $t = i$ then.

Assume $\Gamma \vdash s \xrightarrow{l\leftarrow e} s'$, i.e., $s' = s[\sigma.l := [\![e]\!]^s_\Gamma]$. The goal is to show $\Gamma \vdash s \xrightarrow{\texttt{assert}(e=e');l\leftarrow e'} s''$ for $s' = s''$. The derivation always exists with result $s'' = s[\sigma.l := [\![e']\!]^s_{\mu(\Gamma)}]$. The

[16]See the chapter 1 for an example.

```
class B{ ... }
class Ti extends B{
}
if(x instanceof Ti)
  t = m(..., t);
```

```
class B{
  method m_indirect(..., t){
    return t;
  }
}
class Ti extends B{
  method m_indirect(..., t){
    return m(...);
  }
}

t = x.m_indirect(...);
```

Figure 4.7.: Pattern for “Replace Type Tests with Dynamic Dispatch”

prefixed assertion (without operational interpretation!) tells us that $[\![e' = e]\!]^s_{\mu(\Gamma)}$ and consequently $[\![e']\!]^s_{\mu(\Gamma)} = [\![e]\!]^s_{\mu(\Gamma)}$. Expression evaluation is unaffected by μ as could be shown by induction on the structure of expressions. Therefore, $[\![e']\!]^s_{\mu(\Gamma)} = [\![e]\!]^s_{\Gamma}$ and $s' = s''$.

4.7. “Replace Type Tests with Dynamic Dispatch”

“Replace Type Tests with Dynamic Dispatch” is another very simple refactoring if a modest interpretation is adopted that is very apt for composition (i.e., it looks a bit silly if it isn’t composed). Consider the simple case where the types $T_1, \ldots, T_N$ you test against have a single common superclass B. For every (general) test you introduce a function that does not do anything for the base class and executes the then-part of the test for the class that is tested against.

What could happen when translating a series of if-statements is that you end up calling lots of different functions that have been generated according to the pattern above:

```
t = x.m1_indirect(..., t);
t = x.m2_indirect(..., t);
... // and so on
```

In that case, it may be time to merge these different functions. Two virtual functions with identical scope can be merged if they share the same implementation for all classes for which they are both implemented. This is just the case for the functions produced by the refactoring above.

Preconditions The formalization adheres to the pattern in figure 4.7. There is a base class B and its subclass T_i: $T_i \preceq_\Gamma B$.

The type test to be replaced is identified by q:

$$\Gamma[q = \texttt{if}(x\, \texttt{instanceof}\, T_i)\{t \leftarrow v.m(e)\}\texttt{else}\{\texttt{skip}\}]$$

For the transformation to be syntactically valid, it is necessary that $\mathrm{ctt}(x) = B$. The new method that is introduced is called *indirect* and is fresh in the program.

Data correspondence A complex data correspondence is avoided. The statements in the newly introduced method are not considered comparable. The dynamic dispatch and the delegation to the original method have to be treated in one step: $\beta = \mathrm{id}$.[17]

Transformation This is just a matter of replacing q and introducing a new method that does the dispatch. I write P for the names as well as the static types of (e, v, t).

$$\begin{aligned}\Gamma[B.\,\mathrm{methods}\,.(\mathit{indirect}, P) &:= (\mathrm{ctt}(t), P, \texttt{result} \leftarrow t)\\ T_i.\,\mathrm{methods}\,.(\mathit{indirect}, P) &:= (\mathrm{ctt}(t), P, t \leftarrow v.m(e))\\ q &:= t \leftarrow x.\mathit{indirect}(e, v, t)]\end{aligned}$$

Correctness The correctness proof is only interesting for q. It then works by case destinction on the value of $x\, \texttt{instanceof}\, T_i$. The body of *indirect* has to be unfolded completely to show equivalence.

4.8. "Use Reducible Language Features (Syntactic Sugar)"

Reducible language features and syntactic sugar in particular is increasingly popular in Java as well as the mainstream .NET languages (C#, Visual Basic, J#). It is introduced to capture parts of patterns that have become widely established and are built into the framework. These refactorings are trivial but nonetheless important because they usually provide the last step of "refactoring to patterns": Once the pattern is "diffused" into the program, syntactic sugar makes it even more visible. Moreover, syntactic sugar often provides some kind of syntactic checking beyond the type constraints of the primitive constituents of a pattern. The section provides a number of examples for synactic sugar that show that the scope of today's syntactic sugar constructs is quite broad and not confined to its most banal incarnations like operator overloading, "indexers" and arrow syntax (`->`) in C++.

[17] Remember that when I write β instead of $\beta_{[t_1,t_2]}$, it means that β is defined for all symmetric program parts, i.e., $\beta_{[t,t]} = \beta$. In this case, the then-part of the if statement is not present in both the original and the transformed program. It thus has to be examined together with the if statement.

Example 4.1. Enumerations in Java are more than *simple* syntactic sugar because they are also types. Syntactic sugar does not normally introduce additional binables into the program. Still, the translation pattern of enumerations is fix as defined by the language definition and it is fair to say that the enumeration pattern as understood in Java is not a first class descendant of algebraic datatypes in functional languages and conventional enum types in Pascal and C but of the typesafe enum pattern.

Example 4.2. Foreach loops in both Java and .NET are examples of syntactic sugar. The translation is comparatively simple. The Java for-each loop

```
for (T it : col)
  ...
```

gets translated to about this code

```
for (Iterator i = col.iterator(); i.hasNext(); ) {
    T it = (T) i.next();
    ...
}
```

Example 4.3. Generics in Java are implemented using erasure (for example described in Appel’s compiler text [1]). I.e., there is an equivalent Java program without generics that will generate exactly the same bytecode as the corresponding program with generics. Generics in Java are certainly more than mere syntactic sugar, but this transformation could certainly be useful for the Java 1.4 code that needs to be upgraded.

Example 4.4. LINQ in .NET is a language feature for the .NET that is to date described for C# 3.0[18] and Visual Basic 9.0.[19] This feature allows you to specify declarative query expressions that operate on in-memory collections, databases and XML documents. The following example is adapted from the LINQ Project Overview document [7]

```
var names = new[]{ "Peter", "Adam", "Werner",
                   "Jenny", "Hermann", "Martin",
                   "Arsenii", "Joseph" };

var expr = from s in names
           where s.Length == 5
           orderby s
           select s.ToUpper();

foreach(string item in expr)
    Console.WriteLine(item);
```

[18]http://msdn.microsoft.com/vcsharp/future/
[19]http://msdn.microsoft.com/vbasic/future/

LINQ is not a meta-programming facility. The code in the example above does not contain executable code for filtering the collection. Instead, the query expression is translated to function calls that interpret the collection in the .NET libraries. The translation is specified in the C# 3.0 overview document [31]. A query expression with a single from clause, no orderby clause, and a select clause for instance:

```
from x in e select v
```

is translated to

```
( e ) . Select ( x => v )
```

4.8.1. Formalization

Syntactic sugar are language features whose semantics derives from the semantics of their reductions. For-each loops for instance do not have a semantic production that could interpret them directly. They are translated to the kernel language first and then evaluated. Equivalence and data correspondence are therefore not an issue – these are semantic concepts. Still, the translation and the preconditions can formulated in the manner I present other refactorings in this chapter. Let's consider a very simple case of syntactic sugar with local scope.

The syntactic sugar I want to translate has a label and subtrees as all other statements. The label is S and the subtrees are W. For a simple for-each loop (without the declaration that is required in Java), $S(x, a, b) = \texttt{for}(x{:}a)\{b\}$. $R(W)$ is the reduction of that statement. In the example, this could be

$$R(x, a, b) = i{\leftarrow}a.iterator();f{\leftarrow}i.hasNext();\texttt{while}(f)\{x{\leftarrow}i.next();b;f{\leftarrow}i.hasNext()\}$$

where f, i are fresh variables.

Now suppose at program point q, I want to use syntactic sugar.

Precondition The statement at q has to be a reduction of some syntacic construct:

$$\Gamma[q = R(W)]$$

Transformation The resulting program uses the syntacic sugar instead of the reduced syntax.

$$\Gamma[q := S(W)]$$

```
 1 class C{
 2   int x = 3;
 3   static void main(){
 4     (new C()).run();
 5   }
 6   void run(){
 7     doA();
 8     doB();
 9     out.println(x);
10   }
11   int doA(){
12     x += 1;
13   }
14   int doB(){
15     x += 3;
16     doA();
17   }
18 }
```

Original

```
class C{
  int x = 3;
  static void main(){
    (new C()).run();
  }
  void run(){
    doA(this);
    doB();
    out.println(x);
  }
  static int doA(C it){
    it.x += 1;
  }
  int doB(){
    x += 3;
    doA(this);
  }
}
```

Possibility 1: Replacing all occurences

```
 1 class C{
 2   int x = 3;
 3   static void main(){
 4     (new C()).run();
 5   }
 6   void run(){
 7     doA();
 8     doB();
 9     out.println(x);
10   }
11   void doA(){
12     doA_static(this);
13   }
14   void doA_static(C it){
15     it.x += 1;
16   }
17   void doB(){
18     x += 3;
19     doA();
20   }
21 }
```

Possibility 2: Using mediator

Figure 4.8.: Possibilities to make methods static

4.9. “Make Method Static”

A method can be made static if all invocation sites are known that definitely resolve to this method at runtime and it is known that there are no other invocations that could resolve to this method. In that case, calls as well as the implementation can be replaced.

Consider the following closed and complete “original” program in figure 4.8. The calls to `doA` on lines 7 and 16 can be easily resolved as body$(\Gamma, \mathtt{C}', \mathtt{doA})$ is definitely constant for all $C' \preceq_\Gamma C$. Moreover it is clear that there are no other calls to adjust because the whole program is known. We can transform the program by changing the declaration of `doA` and replacing all calls. This is a dispersed refactoring on parameters.

Non-local dispersed changes in the source program are difficult to handle. One goal of this chapter is to get rid of as many of them as possible. They can be avoided by introducing a mediator that translates the original function call to a call to the new static function `doA_static`. The calls to `doA` can then be eliminated one by one using “Inline Method”.

Take another look at `doA_static`. We could just as well get the result by applying “Extract Method” to the body of `doA` and calling the extracted method `doA_static`. There is nothing peculiar about this example. It represents the general situation: *“Make*

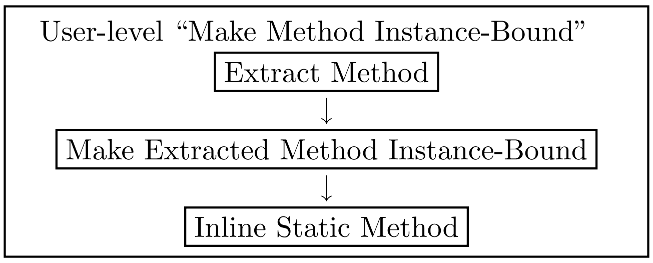

User-level "Make Method Instance-Bound"
Extract Method
Make Extracted Method Instance-Bound
Inline Static Method

Figure 4.9.: Decomposition of "Make Method Instance-Bound"

Method Static" is the chaining of "Extract Method" followed by a number of "Inline Method". Again, this is a simple fact, but it has wide applicability because many other refactorings that are summarized as "Change Signature" can be decomposed in the same manner.

No formal treatment necessary: Refactoring is fully compositional

4.10. "Make (Extracted) Method Instance-Bound"

Section 4.9 elaborates on how "Extract Method" followed by "Inline Method" can be used to make a method static. "Make Method Instance-Bound" is a different refactoring only because "Extract Method" yields a static method in this text.

The "Make Method Instance-Bound" is a refactoring that augments the chain "Extract Method" $\Rightarrow$ "Inline Method". Before inlining, the method with the single call site "Extract Method" has introduced is made instance-bound. This is the purpose of "Make Method Instance Bound". Splitting the desired refactoring into multiple steps simplifies reasoning for each of them.

The difficult part of the proof is to show that only one call site is affected. This is simplest if the name of the method is unique, which is what we will assume for the moment. After all, the method can still be renamed afterwards using "Rename Method" that takes care of all the conditions that need to be observed to retain equivalence of all impl invocations in possible derivation trees. To keep things simple, assume that it is parameter N on which the method shall be dispatched (the parameters are v_1 to v_N). Let the method be `T::m`. In the method's body, all occurences of V_N have to be renamed to **`this`** which is what is passed as the implicit instance parameter. It is necessary to rename **`this`** as well as the **`this`** parameter in static methods points to the respective class object. Figure 4.10 illustrates the transformation mechanics.

The formalization is basically identical to that of "Extract Method" and "Inline Method" and so is the correctness proof. The pattern is summarized in figure 4.10. I again use ren as the variable rename function that is now particularly simple:

$$\operatorname{ren} v = \begin{cases} \texttt{this} & \text{if } v = v_N \\ \texttt{T} & \text{if } v = \texttt{this} \end{cases}$$

```
method T::m($v_1$, ..., $v_N$){            method T::m($v_1$, ..., $v_{N-1}$){
  body                                       body[($v_N$ ↦this,this↦T]
}                                   ⟹      }

single invocation site                     single invocation site
l = T.m($A_1$, ... $A_N$)                  l = $A_N$.T::m($A_1$, ... $A_{N-1}$)
```

Figure 4.10.: “Make Extracted Method Instance-Bound”

Precondition I refer to the formal parameters of the extracted method as well as their types as P and I assume that the single invocation site is identified by q, i.e.,

$$\begin{aligned}\Gamma[\mathrm{metaclass}(T).\,\mathrm{methods}\,.(m,P) &= (P,R,b)\\ q &= l{\leftarrow}T.\,\mathrm{metaclass}(T)::m(V)]\end{aligned}$$

Transformation The transformation consists of renaming the variables in method body b that is added to the newly created instance method. I also leave the original static method in place to avoid having to argue that there it is never called.

$$\begin{aligned}\Gamma[T.\,\mathrm{methods}\,.(m,P_{1..(N-1)}) &:= (P_{1..(N-1)},R,\mathrm{ren}\,b)\\ q &:= l{\leftarrow}V_N.T::m(V_{1..(N-1)})]\end{aligned}$$

Statements in the old static and the new instance method correspond to each other and are comparable:

$$\mu(t) = \begin{cases} T.\,\mathrm{methods}\,.(m,P_{1..(N-1)}).p & \text{if } t = \mathrm{metaclass}(T).\,\mathrm{methods}\,.(m,P).p\\ t & \text{otherwise}\end{cases}$$

Data correspondence β is defined for all corresponding statements and maps local variables in the method. All other values remain unaffected.

$$\beta_{[t,\mu(t)]}(\sigma) = \begin{cases} \mathrm{ren}\,\sigma & \text{if } t \supset \mathrm{metaclass}(T).\,\mathrm{methods}\,.(m,P)\\ \sigma & \text{otherwise}\end{cases}$$

Correctness Only one statement is structurally changed and has to be examined: q. It has to be shown that each invocation resolves to the corresponding method. A case distinction is necessary for all kinds of statements depending on whether it is inside the method or outside. Both cases are trivial.

```
method m{
  X x;
  ...
  ...
}
```

$\Rightarrow$

```
method m{
  X x;
  ...
  req: o ≠ Null
  // ''temp_f'' is fresh
  F temp_f = o.f;
  o.f = x;
  x = o.f;
  o.f = temp_f;
  ...
}
```

Figure 4.11.: Kernel version of “Move Local to Object”

4.11. “Move Local to Object” and “Field to Local”

Both refactorings in this section are *complex.* Their applicability is based on semantic conditions that heavily depend on the syntactic formulation. The syntactic requirements in turn are difficult to describe concisely. I therefore omit a formal treatment in favour of a more conceptual overview that describes possible implementation strategies.

4.11.1. “Move Local to Object”

The refactoring discussed in this paragraph is one of the transformations whose scope is highly debatable. I referred to this refactoring while discussing encapsulation of individual parameters and will be considering “Move Local to Field” in a way that aims at satisfying this simple case *only.* Minimal deliberation shows that the refactoring is quite sufficient if refactorings conventionally used as compiler optimizations are allowed. They include transformations for reordering statements, for copy propagation, register promotion, etc. The basic idea is to allow only trivial assignments that copy a value to an object and then back as shown in figure 4.11. A temporary variable is used to make sure the object’s field value is retained. This temporary variable that backs up the object’s original value can be removed if it turns out that the object does not escape or the escaping object has the field reassigned before it is read. This is most easily tested locally but interprocedural tests could also be envisioned.

If this refactoring does not look terribly useful in isolation, it is probably because it isn’t! It is a refactoring that is meant to be used as part of “Introduce Parameter Object”, a refactoring in [16].

The “Move Local to Field” refactoring can be used like this: “Introduce Parameter Object” changes the method interface by replacing the parameters by an object that contains the respective parameter values. Clearly, “Extract Method” can be used to avoid having to reason about non-local properties and protect the rest of the source code from modifications. This is not enough however because the code still expects scalar parameters, not an object that is decomposed.

The solution is to introduce another intermediary routine that takes the object apart that was created by the first routine. The only challenge for this middle routine is to prove that it retains the input values and passes them unchanged to the original implementation. This is depicted in figure 4.12.

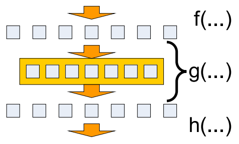


Figure 4.12.: “Introduce Parameter Object” intermediaries

The individual steps in the transformation are depicted in figure 4.13. It also shows that the method can also be extended to an arbitrary subset of the parameters by simply introducing an object that aggregate fewer parameters.

4.11.2. “Move Parameter Object’s Field to Local”

“Move Field to Local” tries to use a local variable instead of an instance variable. Fields are sometimes considered “the new globals”: they can obscure the responsibilities of methods and objects and make understanding programs difficult. It is easy to determine when fields have become unnecessary by backward slicing the program from all external method calls. In compiler optimization, it is sometimes refered to as “aggressive global dead code elimination” and has received extensive formal treatment.

Transformations to keep a field value in a local variable are well established because they can reduce load on the data bus and are an effective compiler optimization called “register promotion”. For the present refactorings, it is best to rely on this existing technique. Then the steps in figure 4.14 are possible for any field `f` that is to be moved to a local variable. In figure 4.14, `x` stands for any object instance abstraction as determined by some points-to analysis. Every assignment to the field `x.f` is replaced by an assignment to a temporary variable `temp_x_f` that corresponds to the value in field `f` of that abstract object followed by a separate assignement of `temp_x_f` to `x.f`. Every access of `x.f` with assignment to `l` is replaced by an assignment of `x.f` to `temp_x_f` followed by an assignment of `temp_x_f` to `l`. The ideal is that register promotion of `x.f` in `temp_x_f` completely eliminates read accesses of `x.f`.

A clever dead code elimination could then render the assignments to `x.f` redundant. As a last step, the field can be removed from the class definition if desired.

```
class P{ A a; B b; }
```

```
f(A x,B y){
  ...
}
```

$\Rightarrow^{a}$

```
f(A x,B y){
  g(x,y);
}

g(A x,B y){
  ...
}
```

$\Rightarrow^{b}$

```
f(A x,B y){
  g(x,y);
}

g(A x,B y){
  P t = new P();
  t.x = x;
  t.y = y;

  x = t.x;
  y = t.y;
  ...;
}
```

$\Rightarrow^{c}$

```
f(A x,B y){
  g(x,y);
}

g(A x,B y){
  P t = new P();
  t.x = x;
  t.y = y;
  h(t);
}

h(P t){
  x = t.x;
  y = t.y;
  ...;
}
```

[a] "Extract Method 1" isolates the callers from any changes. They may later wish to use "Inline Method", possibly after simplifications to the method structure. These refactorings transform the program while structurally retaining the method body returned by body(Γ, T, m). Except for "Push Up/Pull Down Method", such refactorings are not described in [16]. They are probably considered too simple.

[b] "Move Local to Object" The step is new and explained in this section.

[c] "Extract Method 2" This is how the desired object is introduced as a formal parameter.

Figure 4.13.: Stepwise transformation to get desired "Introduce Parameter Object"

```
...
x.f = E;
...
l = x.f;
...
```

$\xrightarrow{\text{this refactoring}}$

```
...
temp_x_f = E;
x.f = temp_x_f;
...
temp_x_f = x.f;
l = temp_x_f;
...
```

$\xrightarrow{\text{promotion plus local dead code elim.}}$

```
...
temp_x_f = E;
...
l = temp_x_f;
...
```

$\xrightarrow{\text{remove field if possible}}$

Figure 4.14.: "Move Field to Local"

```
                                                f(..., O o){
                                                  temp_o_f = o.f;
f(..., O o){              f(..., O o){              g(..., o, temp_o_f)
  ...               ==>     temp_o_f = o.f;   ==>   }
}                           ...[promote o.f]
                          }                       g(..., O o, F f){
                                                    ...'
                                                  }

// Ext. call site         // Ext. call site       // Ext. call site
f(..., o);                f(..., o);              f(..., o);
```

Figure 4.15.: “Move Field to Local” for parameters

Usage

The construction is introduced to extract a field from a parameter object and pass it separately. Here is how.

The construction again relies on “Extract/Inline Method”. It is shown in figure 4.15. The last parameter `o` in method `f` is the object from which we want to extract the field `f` to pass separately. As always, the signature of `f` must not change because this would mean non-local code updates. Such updates are condensend in “Inline Method” and other refactorings that unify identical methods as described above.

`f` should now use a local variable instead of `o.f`. We will turn this local variable into a parameter using “Extract Method”. We do this by just assigning `o.f` to `temp_o_f` and relying on promotion of `o.f` in `temp_o_f` for the rest of the method body. Needless to say, I assume that the field `f` is used unconditionally such that exceptions etc. are not an issue for equivalence. Promotion is subject to dependency and alias analysis of course. If `o.f` is afterwards still read inside the body (the body of `g` in figure 4.14), well, bad luck. It means that the field cannot be eliminated.

4.12. “Rename/Reorder Parameter”

This is the only refactoring on method interfaces that is applicable and still not completely banal – unlike “Remove Parameter” and “Add Parameter”, that are applicable only if the parameter is not used. Let the parameters be v_1 to v_N. If parameters v_i and v_j are to be swapped in procedure $C::m$, β is $\beta_t(v_i, v_j)$ and $\beta_t(v_j, v_i)$ if $C::m \subset t$ and id if the left-hand side is not in $\{i, j\}$ or the program counters is not in $C::m$. Parameters can be used almost everywhere, so μ can have an effect on almost all instructions. Every parameter access has to be translated, including the parameter accesses in ordinary expressions. Parameters are not qualified with their enclosing method because of dynamic dispatching. μ is thus defined as $\Gamma[\textsf{if } C::m \subset t \textsf{ then } t := t[v_i \mapsto v_j, v_j \mapsto v_i]]$. As this

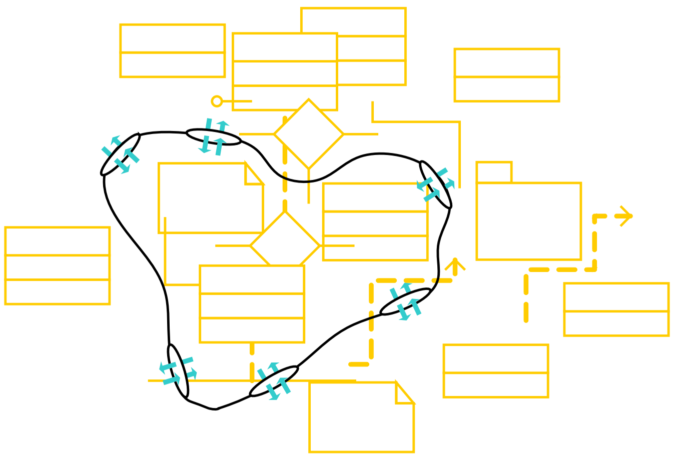


Figure 4.16.: Isolating changes with data translation

translation function shows, the refactoring is just a matter of renaming local variables. See sections 4.3, 4.4, 4.9, 4.10 for how this is formalized.

4.13. “Rename Field”

“Rename Field” has the characteristics of the refactoring in chapter 5 and it is a special case thereof. Just like the more general refactoring in chapter 5, the refactoring cannot easily be made local: Either you have one field name or you have a different field name. I.e., the problem is that you cannot isolate the refactoring by mapping the changed abstractions to the original abstractions without losing the gist of the refactoring. Here is one possibility: You could introduce an additional field with the new name. You then identify the region to which you want to apply the “Rename” and then copy the old field to the new field whenever entering that region and vice versa when leaving the region. This does not work well when non-local data is concerned as in the case of objects and their fields.

4.14. “Delete Obsolete Element”

Is removing unused code elements a refactoring? Many programs are cluttered with unreachable code, functions that are never invoked, classes that are never instantiated. So at least I can say it is a very important transformation

Getting rid of these code elements safely and quickly can reduce the “restistance to change” of existing code considerably. For private members and locally scoped variables, warnings are normally generated by the compiler to indicate that a program abstraction is not used. For public and protected class members, the compiler cannot possibly issue a warning because the abstraction is visible and usable from outside the compilation unit. A refactoring tool that helps eliminate unused abstractions must have a whole program view, as usual for refactorings.

A whole-program view is inevitably tied to the kind of model choosen for the operational semantics: The whole program's tree representation Γ. This means that all investigations *automatically* assume the presence of the whole program. While this is a trivial fact, it is particularly important to keep in mind for "Delete Obsolete Elements".

Deleting an obsolete method is the most delicate of the "Delete Obsolete Elements" family of refactorings. The correctness proof is becomes trivial when the following

4.15. Conclusion

This chapter has shown that many complex refactorings can be aptly decomposed into simpler refactorings. The primive refactorings that remain are confined in their scope or the number of modifications that are necessary. For such refactorings, the notations and the proof method presented in chapter 3 is useful – even when only used as a conceptual framework.

Some local refactoring however are still difficult to describe and investigate. They involve aliasing properties of objects that are referenced inside a method. Even though these properties are difficult to describe syntactically, it may not be worthwhile to include them as postcondition assertions because they do not express architectural constraints. It is better to rely on existing analyses such as the ones used in optimizing compilers. For this kind of refactorings, a more expressive spezialized notation would be useful to describe the transformations and the preconditions.

5. Refactoring Access Paths: Moving Data Between Objects

This chapter is about a specific class of refactorings that are concerned with how data is accessed in the program, i.e., about access paths in the object graph of the program. It starts with an example and then continues with a more general discussion that leads to the conclusion that *moving* fields in the access path – replacing one access path by another – is the most important such refactoring and the most theoretically rewarding as well.

Summary. Replacing one path in the object graph like `x.f` by a different one like `x.t.f`, where `t` and `f` are field names, is a refactoring if `x.t.f` after the replacement identifies the same value as `x.f` before the replacement. This can either be the case because the structure of the program is such that `x.t.f` and `x.f` have the same value or because the transformation replaces updates to `x.f` by updates to `x.t.f`. The path `x.f` becomes invalid in this case. The first refactoring is called "Replace", the second is called "Move". Establishing the redundant structure expected by "Replace" is the "Copy" refactoring. "Copy" is special because it does not alter existing structures in the program. It merely adds new ones.

All three access path refactorings "Replace", "Copy" and "Move" have architectural significance. The paths I allow are *simple* and *fixed* paths that specify the fields used to access a value (e.g., of the form $p_1.\cdots.p_n$). When *replacing* access paths in a program, such simple paths prove insufficient but provide tangible results. A thorough investigation that would pay tribute to the importance of "Replace" in program redesign would have to allow arbitrary paths. This is done by generalizing actual instance variables to *field abstractions*. It is omitted for the lack of time, space and conceptual clarity. In this research, "Replace" is acknowledged to be important but not derived separately. The same is true for "Copy". This is because "Move" contains all elements of both the "Copy" and the "Replace" refactorings: "Move" can simulate "Copy" and "Replace". The opposite is also true: "Move" can be split into "Copy" followed by "Replace". It is simpler however to minimize the basis and consider only "Move" a primitive refactoring because all of the three refactorings have to tackle similar problems (aliasing, value identity, etc.). That's why "Move" is formally investigated. "Copy" and "Replace" aren't. "Move" can be further decomposed according to the structure of the simple paths. The result is a number of "Move Field" transformations. For this base transformation, semantic postconditions are established and syntactic approximations are discussed.

5.1. About access paths

Here is an example of an access path and a picture of one possible object graph in figure 5.1:[1]

```
x.next.parent.spouse
```

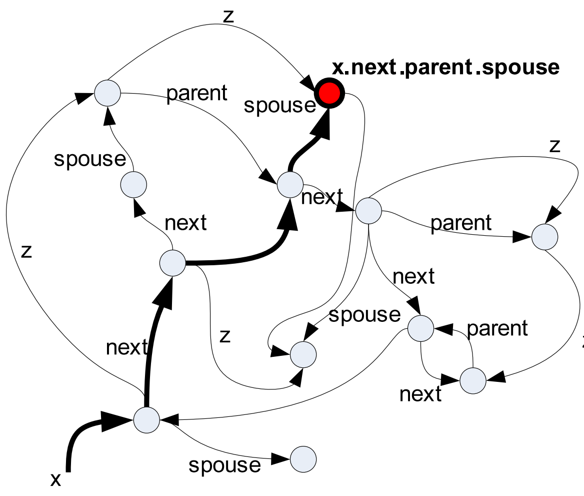


Figure 5.1.: Excerpt from a possible object graph with path `x.next.parent.spouse`

Access paths are subject to refactoring. Most often, there is more than one way to access a certain object. In the example in figure 5.1 for instance, the path `x.next.parent.spouse` identifies the same object as

```
x.z.z
```

The fact that `x.z.z` and `x.next.parent.spouse` identify the same object may just be a coincidence. It may however also be a *structural property* of the program mandated by the program's very design.

It is this second kind of systematic access paths correspondences that is interesting because they are based on architectural decisions.

Relevance The way individual objects are accessed is directly coupled with what modifications and extensions can be made. In the example above, the `next`, `parent` and `spouse` links can be modified without affecting program parts that access only the `z` link. Likewise, intermediary objects can be altered only if they are not used in access

[1]The syntax is not valid in may language.

paths. Moreover, access paths identify the program's data representation. Moving data between objects involves adjusting access paths. In other words, access path modifications can reduce a program's resistance to change to new features more directly than other refactorings. The relevance of access paths has long been recognized by Lieberherr's claim for "structure shyness" [27].

I reckon that it was only this class of refactorings as the last refactoring in a chain that lead Roberts to conceive the idea of postconditions as predicate transformers in his Doctoral Dissertation [39].

Application to subsets Access paths are valid for objects of certain classes. Often, fields are used differently depending on the context in which the class is used. Just consider the `parent` link of a (concrete) `Item` class.[2] Depending on the instantiated subclass, the field may have a different meaning. In a `Person`, it may denote the physical parents, in a `Node`, it may point to the tree node that has the object as one of its children, in a union-find structure, it might represent the canonical element of the set the node belongs to, etc. The set of objects that should be affected by a certain refactoring may thus not be defined by the actual type of the object. Other classifications may be more reprentative as illustrated in the OMS model [35] in figure 5.2 for a contacts database.

The transformations presented in this chapter – just as any other transformation affecting objects of a certain type – can be applied to *any disjoint sets* of such objects. This fact is mentioned and stressed in this chapter because the "Copy" refactoring can be used to *unbundle* different responsibilities and values that are stored in the same field. Example: in the `Item` class, one can introduce different fields for the super-class field `parent` depending on the group to which the object belongs.

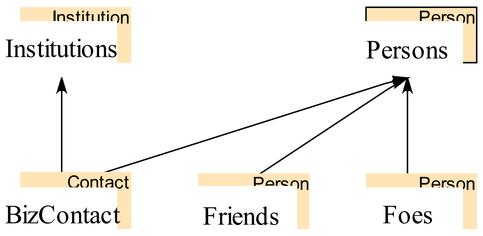


Figure 5.2.: OMS model for contacts: Classification beyond types

Tool support Access refactorings have architectural relevance. This is the first reason why access path modifications got their own, separate chapter. Another is that the investigation of these refactorings does not benefit too much from the general framework defined earlier – the derived criterion that two access paths are equivalent if they

[2]Yes, I want to illustrate bad design here.

return the same value/exception is quite intuitive. Moreover, none of the refactoring tools I tried (IntelliJ IDEA, Xrefactory, JBuilder, Eclipse, etc.) properly supported access path refactorings.[3] The Smalltalk Refactoring Browser does not support them either.

Access refactorings are non-local Access path refactorings are difficult to frame as local refactorings: There is normally only one data representation in the program. Either a data representation is changed or it is not. This is not true for program code: Code can be easily replicated without affecting the program. It is of course a matter of framing the other refactorings correctly such that non-locality does not occur. With access paths refactorings, it is much less easily possible to keep them local. For an illustration of this claim, consider a simple "Move Field" where a field declaration is put in a class that is directly related. Consider the two class definitions below before and after the transformation. The path `A::f` is to be replaced by `A::t.B::f`.

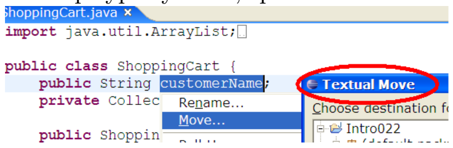
```
class B{        | class B{
}               |   int f;
class A{        | }
  int f;        | class A{
  B t;          |   B t;
}               | }
```

Now consider the following use, before and after a *the first field access replacement* in the refactoring

```
f(A a){                  | f(A a){
  out.println(a.f);      |   out.println(a.t.f);
}                        | }
```

This change would require all write updates to `A::f` to be updated as well! A similar problem occurs when trying to replace a single *write* access `a.f = ...` by `a.t.f = ...`. Refactoring single statements necessitates changes all over the program. Changing a field affects[4] the whole program; neither the new nor the old field is locally confined. This is

[3]Eclipse's "Move" refactoring turns out to be a "textual move", which isn't very helpful.
The only faithful discussion of access path refactorings is found in [39]. There seems to be the real need to discuss this kind of refactorings in the light of more recent implementation possibilities like ownership type systems, specifications etc.



```
import java.util.ArrayList;

public class ShoppingCart {
    public String customerName;
    private Collec
    public Shoppin
```

[4]I would even say: Infects!

certainly not ideal. If the changes are done in a certain order, you still lose the possibility to reason about the whole transformation.

Postconditions Access refactorings are the only refactorings with substantial postconditions that are non-local *and* non-trivial at the same time.[5] It is a mere necessity that they should be made explicit as specifications because non-local properties correspond to architectural decisions. In the example above, it is required that `a.t` is non-null at the beginning of `f(A a)`. This is a condition that is not easy to check as a precondition using static analysis but it is very easy to check dynamically. These postconditions can also be translated to preconditions in which case they can benefit from the invariants and assertions established by preceding refactorings.

Three kinds of access path refactorings: "Replace", "Copy" and "Move" Access path refactorings transform the program in such a way that it does not use a path $p = p_1. \cdots .p_n$ to access a value but a different path $q = q_1. \cdots .q_m$ instead, possibly on a different object. This can mean three different things that are compared in table 5.1.

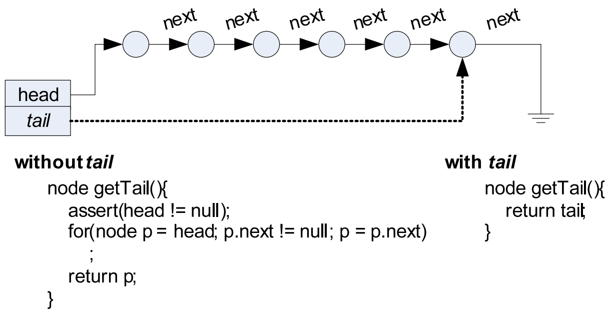
"Replace" The two paths identify the same value because of the structure of the program as in the example above.

It is unusual to find a program where this interpretation can be readily applied if only proper fields are considered. If "Replace" can be applied, there is a redundancy in the object graph that should be considered for elimination because it creates cumbersome maintenance responsibilities. Redundancies however can yield speedy access to objects or they can legitimately result from circular structures, bidirectional associations and objects in multiple structuers like in figure 5.4. This is more common in C++ than in Java where value classes do not exist. Consider the singly-linked list in figure 5.3 for an example where an explicit pointer to the last element in the list is stored and has to be maintained.

"Copy" The kind of redundancies that are needed for valid applications of "Replace" are introduced by the "Copy" refactoring. p will identify the same value as q after the refactoring.

"Move" The path p is replaced by path q, i.e., the structure before the transformation is such that p accesses some value while q is invalid. After the transformation q is valid and refers to the value p used to refer to but p is now *invalid*. The simplest example of such a refactoring is "Move Field" that will serve as an initial example.

[5] "Move Local to Field" and vice versa for instance has non-trivial but local postconditions. Isomorphisms like rename method have non-local but trivial (i.e., no) postconditions. "Move Local to Field" is discussed in chapter 4, but postconditions are not discussed for them.



```
without tail
node getTail(){
    assert(head != null);
    for(node p = head; p.next != null; p = p.next)
        ;
    return p;
}

with tail
node getTail(){
    return tail;
}
```

Figure 5.3.: Linked list with and without tail pointer

There are indeed reasons to believe that these are the only cases that are useful for refactoring programs: *Read* accesses to p can either be left intact or can be mapped to read accesses to q. If they are left intact, write accesses must be left intact as well because accesses to p should still yield the same results. Write accesses to q can be added without harm if q is invalid before. ("Copy") If read accesses to p are mapped to accesses to q, it is either possible that q already has the same value as p ("Replace") or that the value of q must be established before ("Move"). It would be possible to keep the value of p, but it is hard to see why, because this kind of redundancy can be reintroduced at any time by the "Copy" refactoring.

Name	q before	p after	affect p's	affect q's
"Replace"	valid, same as p	valid, same as q	read	none
"Copy"	invalid	valid, same as q	none	write
"Move"	invalid	invalid	read/write	read/write

Table 5.1.: What to do with p and q: Schematic view of "Copy/Replace/Move"

Name	Γ	Γ'	require	ensure
"Replace"	read p	read q	p and q are the same	–
	write p	write p	–	–
"Copy"	read p	read p	–	–
	write p	write p and q	–	p and q are the same
"Move"	read p	read q	–	–
	write p	write q	–	–

Table 5.2.: Conceptual view of "Copy/Replace/Move"

This can be formulated as a conceptual relation between the three refactorings: "Moving" object references involves changing the update as well as the read accesses. "Replacing" object references involves changing read accesses: The structure must already be established by the program. "Copying" object references means changing write accesses: It establishes a structural relation.

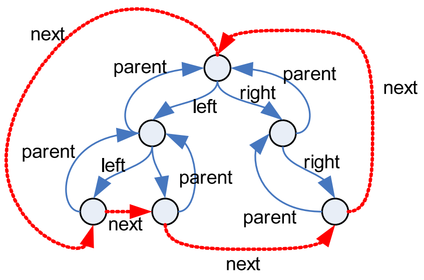


Figure 5.4.: Tree elements that are in a circular list

“Move” is the most complex of the three refactorings. It thus is tempting to believe it is beneficial to programmatically implement “Move” by “Copy”, which establishes the required structure in the object graph followed by a “Replace” where all read accesses to the old path are replaced by the new one. As a last step, references to the old path could be removed.

The following code example schematically illustrates the chain for “Move” as a composite refactoring. Not all conditions are shown.

```
                                   x.n.e.w=...
                 x.n.e.w=...       x.w=x.n.e.w
x.w=...          x.w=x.n.e.w       ens: x.w = x.n.e.w     x.n.e.w=...
...=x.w          ens: x.w = x.n.e.w req: x.w = x.n.e.w    ...=x.n.e.w
                 ...=x.w           ...=x.n.e.w
```

The ens/req pair can be immediately eliminated.

```
ens: x.w = x.n.e.w
req: x.w = x.n.e.w
```

The correctness proof later in this chapter shwos that (value-)equivalence per se is easier to reason about than the necessary invariants in the object graph. It is thus a better idea to use the more complex “Move” as a basis for both the “Copy” and the “Replace” even though the opposit would also be possible.[6] *Once you have “Move” defined, the “Copy” and the “Replace” refactorings come for free.*[7] *Here is how.*

I treat “Copy” and “Replace” as composites of “Move”, not vice versa

[6]The definition of primitive refactorings – refactoring that *are not* composed from primitive refactorings – always ultimately boils down to the kind of utility considerations done here.
[7]For “Replace”, additional postconditions may be necessary.

5.2. “Move” to put together “Copy” and “Replace”

5.2.1. “Copy”

For the *“Copy”* refactoring, consider its trivial case when a redundancy is introduced as a second field with exactly the same content. (in this example `C::f` and `C::f2`)

```
class C{
  X f;
}
```

```
class C{
  X f;
  X f2; // ''Copied'' field
}
```

This kind of redundancy is easy to introduce and does not need any pre- or post-conditions. Just make sure that the second field is always set when the original one is:

```
                                           C temp_obj = $l_1$;
$l_1$.f = $l_2$; $\xRightarrow{\text{“Direct Copy”}}$  X temp_val = $l_2$;
                                           temp_obj.f = temp_val;
                                           temp_obj.f2 = temp_val;
```

Once you have the `f2` field *in the same class as* `f`, you can move it anywhere achieving the same effect as with a complex “Copy”.

5.2.2. “Replace”

For *“Replace”*, a more indirect construction has to be employed that might not always work perfectly without additional postconditions. Consider the example where you have two classes `C1` and `C2` that have the fields `f1` and `f2` respectively. You want to replace accesses to `f1` by accesses to `f2`.

```
class C1{
  X f1;
  ...
}

class C2{
  X f2;
  ...
}
```

The classes are related somehow such that you can employ “Move” on `f1` to put it into `C2`:

```
class C1{
  ...
}

class C2{
  X f1;
  X f2;
  ...
}
```

If your assumption was correct that `f1` and `f2` contained the same value at the beginning, this must still be the case after the “Move”! If you can prove this, you’re done and replacing accesses to `f1` by accesses to `f2` is trivial.

Note that the initial “Move” does not introduce any new uncertainty or problems: The conditions of “Move” are minimal for identifying the object uniquely that contains the corresponding `f2` field.

What might cause problems however – and this is why this decomposition may not always work as is – is that you have to prove that `f1` and `f2` contain the same value. In fact, you could even think that the transformation didn’t do anything for solving the problem!

This is not true however: Before the transformation, the refactoring tool would have had to reason about values in different objects. Such reasoning is complex and should be isolated. After the transformation, the mere *values* of two variables have to be compared that are either both defined (reference to container is non-null) or undefined (reference is null or cannot be reached). This means that after the transformation, a very simple refactoring browser could just textually compare the code that calculates the values for each assignment. Side-effects have to be taken into account though. A more sophisticated refactoring tool could do some data flow analysis to find and compare the proper value of `f1` and `f2`. If all that does not help, the tool can add *postcondition* assertions that checks equivalence: **`assert`** `x.f1 == x.f2`.[8] All three variants benefit massively from the decomposition because they are freed from having to reason about the availability of necessary objects. This is taken care of by the initial “Move”.

[8]Needless to say, that’s my favourite!

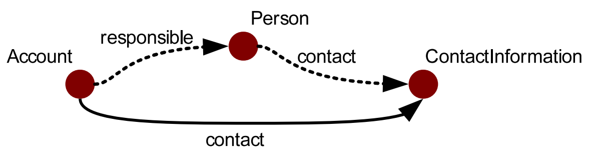


Figure 5.5.: “Remove Indirection” as an example of “Replace” where fixed paths are sufficient

Access path specifications Access paths can be arbitrarily complex. Consider figure 5.3 again. The number of `next` links that have to be followed to reach the tail is not determined in advance. It depends on the number of elements in the list. Figure 5.3 uses a **`null`** reference to signal the end of the list. A sentinel could be used just as well or a reference to the last element or a reference to the first element in the list. Conventions can be arbitrarily complex. Restricting the possible access paths that can be transformed to an arbitrary subset will pose limitations that are hard to justify and hard to understand.

How useful is it to restrict the transformation to fixed access patterns like

$$p_1.\cdots.p_n$$

where all of p_1 to p_n are determined in advance. At least for the “Replace” refactoring, my suspicion is that: restricting access paths refactorings to fixed patterns is not very useful in practice because such patterns can only be used where the structure is limited and fixed. It is exactly in these cases that redundancies are least useful and most unlikely to occur.[9] But this is exactly where they are needed most. *Yet, fixed access patterns is precisely the abstraction I will use: Only fixed access paths that are specified using sequences of fields are allowed as arguments.* A precise definition of “fixed access paths” is given below.

In this chapter, I want to confine myself to simple access paths that are not very useful for “Replace” because they are less intractable to handle formally than totally general access paths. Simple access paths are a pragmatic solution and one that yields tangible results.

A more generous interpretation of fields as association values is still possible. In this case “fields” are not merely the instance variables of objects but all values that are associated in some way with an object and change through certain state updates – “field” values can be computed for example as in figure 5.3. This allows to treat all kinds of generalized access refactorings but puts the burden of interpreting the notions of the access refactorings in terms of generalized fields, which is just as difficult as general access paths, but more tangible. This is what the next paragraph is about

[9]A notable exception is when a level of indirection is added or removed as in figure 5.5, which does realistically occur in practice. A different kind of argumentation would be that legacy programs be assumed to contain simple, redundant data structures and it is worthwhile for that reason to cover “Replace” for fixed access paths.

Field abstractions Access paths do not necessarily have to be specified as fields as suggested when writing $p = p_1. \cdots .p_n$. The association that is resolved through the fields p_1 to p_n could also be put somewhere else – for instance ouside the classes involved. Consider the following example:

```
class A{ int f; }  class B{}
```

"Moving" with $p =$ `A` :: `f` and $q =$ `B` :: `f` is not directly possible using access paths because `A` and `B` are not related by fields. There might still be some relation in the program associating `A` and `B` objects (a static map in this case).

```
    class Prog{ static Map<A,B> assoc }
```

In this example, but also in general, it is necessary that only one `B` object be associated with every `A` object at any time when the `f` field is used. Every kind of association must pay tribute to this fact (a static association as above for instance; getter/setter like interfaces are of course always sufficient). A field is just a model for this kind of behavior. Whether the model exists or not is irrelevant. It is therefore possible to argue about simple, fixed access paths only without loss of generality.

5.3. Possibilities and meaning of access paths

The refactoring that is treated here is of the form

"Move" p to q

This section answers the due question what this refactoring means and how the objects that are accessed by p and q can be resolved.

Consider the example when p and q are simply fields in the same class, i.e., $p = C{::}f$ and $q = C{::}g$, i.e., the field f is to be replaced by a field g in the same class, retaining all updates and reads. This is a simple "Rename Field" as discussed in section 4.13. To simplify the other cases, let's assume that the name of the field to be moved remains the same, i.e., $p_{\text{actual}} = p.f$ and $q_{\text{actual}} = q.f$

Let p_{actual} again be the field f of a class C, i.e., $C{::}f$. If q also starts with a field in C, the "Move" is a simple "Move field": The field is moved to the object identified by the path q in the object graph.

The inverse transformation can also be considered: The field f is moved from the object that is identified by p to the object p originates from. I call this "Reverse Move Field".

Needless to say, the two transformations can be combined: The field can be moved backward along p to some originating object and then forward along q to an object that is pointed to by q.

The four basic possibilities are illustrated in table 5.3.

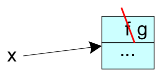
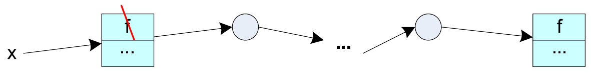
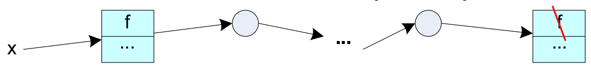
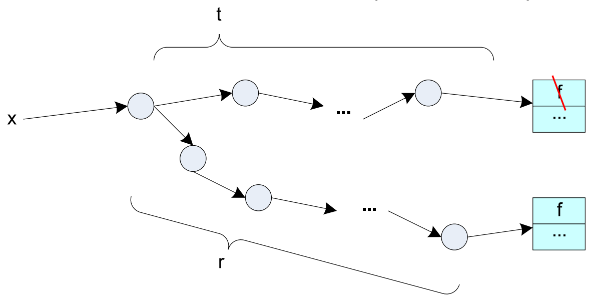
Name	p	q	Meaning
Rename field	$F::f$	$F::g$	rename f in F to g
			
Move field	$F::f$	$F::t.G::f$	move f from F to G: go through t
			
Reverse move field	$F::t.G::f$	$F::f$	move f through t from G to F
			
Move field indirect	$F::t.G::f$	$F::r.H::f$	move f through t from G to H through r
			

Table 5.3.: Meaning of access paths for “Move”

The three non-trivial possibilities table 5.3 move the field either in the direction of a link to an object that is pointed to or from such an object to an originating object. This can be done step by step. In the example in figure 5.6, the field is moved from the original object to `a`, from there it is moved to `b` etc. until it reaches its final destination in nine steps. The “anchor” object `x` is completely irrelevant now and can be omitted.

Decomposing the transformation into atomic steps makes it possible to treat more complex paths such as the one in figure 5.7.

An access transformation is thus always moving a single field f either forward or backwards along certain fields. The whole transformation is specified by the name of the field to be moved and the sequence of atomic transformations to be executed in the order of their execution. The transformation depicted in figure 5.7 for instance can be summarized as follows when Forward movements along $C::g$ are written as $\overrightarrow{C::g}$, reverse movements are written as $\overleftarrow{C::g}$.

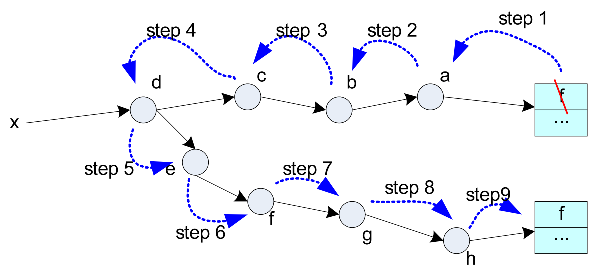


Figure 5.6.: Moving a field step by step

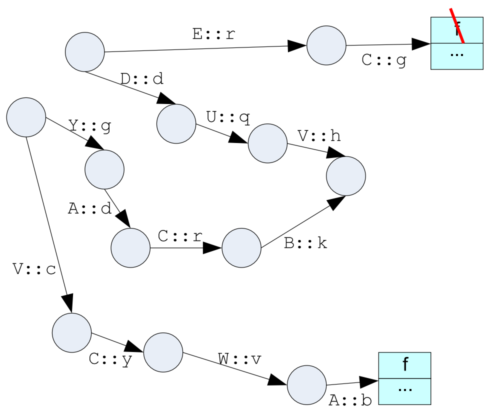


Figure 5.7.: An artificially complex but possible access refactoring

$$\text{move } f \text{ along } \overleftarrow{\texttt{C::g}}, \overleftarrow{\texttt{E::r}}, \overrightarrow{\texttt{D::d}}, \overrightarrow{\texttt{U::q}}, \overrightarrow{\texttt{V::h}}, \overleftarrow{\texttt{B::k}}, \overleftarrow{\texttt{C::r}}, \overleftarrow{\texttt{A::d}}, \overleftarrow{\texttt{Y::g}}, \overrightarrow{\texttt{V::c}}, \overrightarrow{\texttt{C::y}}, \overrightarrow{\texttt{W::v}}, \overrightarrow{\texttt{A::b}}$$

Keep in mind that it does not matter whether the movement is done at once or step by step: Postconditions are transformed together with the program. The refactoring in figure 5.7 is equivalent to the chain

$$\begin{tikzcd}
\boxed{\text{move } f \text{ along } \overleftarrow{\texttt{C::g}}} \arrow[d] \\
\boxed{\text{move } f \text{ along } \overleftarrow{\texttt{E::r}}} \arrow[d] \\
\vdots \arrow[d] \\
\boxed{\text{move } f \text{ along } \overrightarrow{\texttt{W::v}}} \arrow[d] \\
\boxed{\text{move } f \text{ along } \overrightarrow{\texttt{A::b}}}
\end{tikzcd}$$

The forward transformation is a “Move Field”. The “Reverse Move Field” is its inverse as the name suggests. I.e., it is sufficient to examine “Move Field” and show that it is symmetric.[10]

This is what I am doing in the rest of this chapter: Examine the “Move Field” refactoring and its inverse and establish pre- and postconditions.

5.4. The “Move Field” base refactoring

This section discusses and formalizes the “Move Field” refactoring and its inverse, which serves as a basis for all access refactorings. I also discuss how the refactorings can be composed and how the semantic conditions that are necessary for this refactoring can be approximated syntactically. How they are represented as specifications is shown in chapter 6. The specifications that are generated serve as the main illustration in this research how refactorings can be used to convey the assumptions about the program behavior and how these assumptions are translated to specificiations.

I first describe the transformation informally and then switch to a more formal investigation.

[10]Yes, this is actually a very specific reason that inverse transformations are discussed in section 3.3.2. This is another application of the famous Gearloose Principle.

5.4.1. Description

I assume that there is already a field (i.e., an actual instance variable) pointing from the source to the target object, i.e. from the object where the field to be moved resides to the object where it ought to be moved just as in the example below.[11] It is possible to use any other abstraction that *maps* source to target object. For the sake of simplicity, I assume that the abstraction is a simple instance variable. The assumptions about and conditions on the instance variables can then be generalized to arbitrary abstractions.

```
class Src{                class Src{
  X f;                      Target target;
  Target target;            ...
  ...                     }
}              ⇒
                          class Target{
class Target{               X f;
  ...                       ...
}                         }
```

For the transformation to remain well-formed, `f` must not be an existing field in `Target`.[12] Moreover, I assume that `Src` and `Target` are unrelated in $\preceq_\Gamma$ to avoid special cases in my argumentation. As I explain in the introduction to this chapter, “Move” can potentially happen between any disjoint sets of objects, irrespective of how they are related in $\preceq_\Gamma$ and the “field” abstraction could also be **`super`**.[13]

The structure of the heap is changed. β has to take into account the mapping between the two states.

5.4.2. Transformation and applicability

For the transformation, I assume that the field to be moved is f and the class that originally contains this field is called *Src*. The class where f is moved is called *Target*

[11] In section 4.2, I discussed the problem to find the appropriate amount of assumptions for a refactoring. For “Move Field” *per se* it is not clear what kind of structure should be allowed to be already present before refactoring and what is should be allowed to be introduced by the refactoring. Is the class the field is moved to supposed to exist at all before the refactoring? A programmer should be able to bring – with justifiable effort – any code he wants to apply the refactoring to into a shape that is accepted by the refactoring. Writing a new class that encapsulates the functionality of an already existing class just to be able to move a field there if “Move Field” required a new class is not justifiable. But this assumption has been made [11].

[12] I also assume that is does not occur in any of the superclasses of `Target` even though that’s not strictly required by the Java language. In any case, fields are always qualified by the class name `Src::f`, something I ignore in this section for the sake of readability.

[13] This special case is “Pull Up Field”, its inverse is “Push Down Field”.

and the object is refered to by the field *target* in *Src*. In the previous section, $Src = \texttt{Src}$, $f = \texttt{f}$, $target = \texttt{target}$ and $Target = \texttt{Target}$. Using our abbreviated notation, this reads as $\Gamma[Src.\,\mathrm{fields}\,.target = Target \wedge Src.\,\mathrm{fields}\,.f = X]$ where X is any valid type tag.

Forward "Move Field"

Just as for all other refactorings, I first define the forward transformation $\mu^{\text{move field}}_{Src,Target,f,target}$. t_r is a fresh variable for every match of $*$.[14]

$$\begin{aligned}&\Gamma[*.l_r{\leftarrow}x_r.Src{::}f{:=}t_r{\leftarrow}x_r.Src{::}target;l_r{\leftarrow}t_r.Target{::}f\\ &\qquad Src.\,\mathrm{fields}\,.f{:=}undef,\\ &\quad Target.\,\mathrm{fields}\,.f{:=}Src.\,\mathrm{fields}\,.f]\end{aligned}$$

Initialization

It is an *evident* condition that must be fulfilled such that this refactoring can be safely applied and equivalence criterion equation (3.19) is satisfied: *target* must have been initialized when an application wants to access f. Failure to do so will cause a null-pointer exception.

Invariants like $target \neq$ Null are normally assumed to be established by constructors. [11] even requires the *target* object to be initialized in the constructor with a fixed initialization pattern.

Our language does not have constructors.[15] Even if there were constructors, the problem wouldn't have been solved at all: What would the target object to point to be? How can it be ensured that *target* is definitely assigned after the constructor? Moreover, the criteria derived here should be usefully applicable to the implementation of a refactoring tool. It should impose only minimal restrictions on the shape of the program. Giving the constructor a special status and demanding that invariants are established there is incompatible with a program that uses some other method of initialization (like a factory for instance). The solution chosen here is monotonous initialization that is checked dynamically with postcondition assertions: Once the *target* field is initialized, its value must not be reverted to Null.

code mobility principle

5.4.3. Correctness conditions (postconditions)

Informal derivation of correctness conditions

Imagine we keep a "shadow" of field f in *Src* to help understand the required invariants and conditions for sound application of the transformation above. Its value is

[14] I.e., it is an injective naming function from path to identifier.
[15] This decision is legitimated on page 42

what people should get from reading out the refactored f, which is *target.f*. When running the program, the aim is that access to *target.f* always yields the same value as direct access to the shadow *Src*::f. Direct access to *target.f* cannot occur before the transformation and need not be considered. It seems important to understand that assignments to *Src*::*target* will already be present in the source of the program. Indeed such assignments are necessary to establish a relation between source and target objects. It has to be made sure that they do not alter the semantics of the program. Modifying *target* itself is in general tantamount to changing the value of *target.f*. See equation (5.6).

Outline of derivation of conditions

Aliasing can also be a problem: if $x.target = y.target$, assignments to $y.target.f$ can also alter the value of $x.target.f$.

Rationale

The idea to keep a “shadow” fits well with the bisimulation-like equality defined in equation (3.19) and the aim to establish local criteria: β can directly be formulated in terms of *Src*::f, *Src*::*target* and *Target*::f because both the pre- and the poststates are available to β. Following conventional intuition and the criteria in section 3.2.4 (repeated in figure 5.9), β is a mapping between *target.f* and the old shadow field f. The questions that have to be asked for every term is: How can the correspondence be violated and how can the violation be prevented? This is what is done in the paragraph above.

$\beta\ldots$

. . . used to derive the conditions

Organization The rest of this chapter is organized as follows: I first define β and then investigate, for every t the maintenance of the correspondence prescribed by β as mandated by equation (3.19).[16] I am coming up with criteria and show that these criteria are locally sufficient to retain β. Moreover, I give an example (as source-code) where non-adherence to the condition can lead to erroneous behavior. After having done that, I explore known syntactic possibilities that guarantee the local semantic conditions.

β: Correspondence between original and refactored Program

Refresher β is a binary relation on states that relates the state in the original program and the refactored program when the statements being executed are the same. In chapter 3, such statements are called “comparable”. In “Move Field”, individual statements are *replaced* by other statements. Both the structure of the replaced statement and the replacement are fully determined. As explained in chapter 3, it yields the simplest correctness proof if all statements are considered comparable that are not directly inside a replacement or modification as shown in figure 5.9. This could include statements

[16]I.e., it is equation (3.19) that is used.

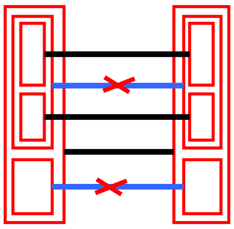


Figure 5.8.: Nested correspondences do not imply outer correspondences

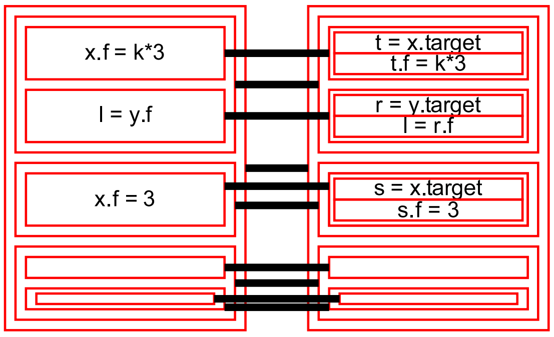


Figure 5.9.: Correspondence for "Move Field" (cf. figure 3.7)

that are nested inside modified program parts as well (see figure 5.8) – for "Move Field" however, this is not necessary.

$\beta(s, r) =$
$\beta(s.\,\mathrm{xcpt}, r.\,\mathrm{xcpt}) \wedge$
$\beta(s.\sigma, r.\sigma) \wedge$
$\beta(s.\,\gamma, r.\,\gamma) \wedge$
$\beta(s.\,\mathfrak{u}, r.\,\mathfrak{u})$

Notations Just as in chapter 4, it is notationally convenient to definitionally extend β from pointwise from its components.

Correspondence The exception state and I/O ought not to be changed by the transformation: $\beta(\mathrm{xcpt}, \mathrm{xcpt})$ and $\beta(\mathfrak{u}, \mathfrak{u})$. This is not quite true for local variables as we introduce a fresh local variable v for every occurence of accesses to f during transformation. $\beta(\sigma, \sigma[v \mapsto t])$ for some determined t that depends on v, the old state and the heap (it contains $l_2.target$). t may be $undef$. More precisely, v could be scoped locally. This would require a concrete formulation of β's dependence on the program counter ($\beta_{[t,\mu(t)]}$). The number of changes is unbounded and so would be the different cases for β. I therefore omit the qualification and I just assume that it is known which concrete variable I am talking about.

For the heap, the allocated objects are the same. We could introduce a bijection between locations to abstract away differences in allocation but that's not necessary. We can thus extend β to heap values:

$$\beta(\gamma_a, \gamma_b) \equiv \forall\, \mathrm{reachable}(loc) : \beta(\gamma_a(loc), \gamma_b(loc)) \tag{5.1}$$

The restriction to reachable(loc) is conservative and unnecessary. It is just a means to make sure the postconditions that result from the derivations can be used as specifications.

It is then easy to define how heap values at the same location relate to each other: f in the original program corresponds to $target.f$; objects of type *Src* and *Target* must be identical if field f is ignored. This is achieved by subtracting f from the domain of the respective value v:[17] $f \lhd v$ All objects that are neither of type *Src* nor of type *Target* are unaffected by the transformation. This is formalized in equation (5.2).

$$\beta(v_a, v_b) \equiv \mathrm{rtt}(v_a) = \mathrm{rtt}(v_b) \wedge \begin{cases} v_a(f) = \gamma_b(v_b(target))(f) \text{ and } f \lhd v_a = v_b & \text{if } v_a \preceq_\Gamma Src \\ v_a = f \lhd v_b & \text{if } v_a \preceq_\Gamma Target \\ v_a = v_b & \text{otherwise} \end{cases} \tag{5.2}$$

This equation can serve as an apt illustration of the wide spectrum anyone has who wants to formalize refactorings. Equation (5.2) could be overly restrictive. As long as the field f is not read or written, the data correspondence need not hold. In the present framework this is easy to formulate with program paths indices on β. Needless to say, loosening the data correspondence renders the proof more complicated.

β is a function on heaps

This section shows that β is a function in its first argument. In section 3.3.2, I show that this property is sufficient to make sure postconditions of the forward transform "Move Field" can be easily transformed to postconditions of the reverse transform. Precise postconditions for the forward transformation are derived from the correctness proof.

β is a function on states. It is trivially a function on exception, local state, I/O. Here I show that it is also a function on heaps. It is not a function on individual heap values because responsibility is moved *between* them. We denote this function on heaps by β. To show that β is indeed a function, consider two heaps γ_1 and γ_2 that solve $\beta(\gamma, .)$. It must hold that $\forall\,\mathrm{reachable}(p) : \beta(\gamma(p), \gamma_1(p))$ and $\forall\,\mathrm{reachable}(p) : \beta(\gamma(p), \gamma_2(p))$. Let p be arbitrary and let $y_1 = \gamma_1(p)$ and $y_2 = \gamma_2(p)$. To show that β is a function, I prove $y_1 = y_2$. Three cases have to be distinguished:

- $x \preceq_\Gamma Src$. $x(f) = \gamma_1(y_1(target))(f)$, $f \lhd x = y_1$, $x(f) = \gamma_2(y_2(target))(f)$ and $f \lhd x = y_2$. Only $f \lhd x = y_1$ and $f \lhd x = y_2$ are needed to deduce $y_1 = y_2$
- $x \preceq_\Gamma Target$. $x = y_1[f \mapsto \gamma_1(y_1(target))(f)]$ and $x = y_2[f \mapsto \gamma_2(y_2(target))(f)]$ leads to $y_1[f \mapsto f_1] = y_2[f \mapsto f_2]$. $y_1 = y_2$ because $f \notin \mathrm{dom}\, y_1$ and $f \notin \mathrm{dom}\, y_2$, which is a well-typedness criterion.

[17] $f \lhd v$ is defined in appendix B.

- otherwise $x = y_1$ and $x = y_2$. $y_1 = y_2$ is trivial.

This paragraph is valid for the data correspondence as defined in *equation* (5.2). Later in this chapter, a new, weaker, data correspondence is introduced where *target* can remain unitialized for a while. β is then still a function. See equation (5.29).

Semantic equivalence for statements

Trivia We are now ready to check equation (3.16) for every possible transition. Most of them are unaffected by μ. They are not considered for the proof. These cases include (the variables in the comment refers to the rule as listed in tables 3.2 to 3.5:

- `skip` because the pre- and poststates are the same,
- $l \leftarrow e$ because $l \neq v$,
- $l \leftarrow (T)e$ because $l \neq v$ and $T \not\preceq$ `ClassCast` if T is related to either Src or $Target$,
- $l \leftarrow$`new` $C[e]$ because $l \neq v$. Moreover `NegArrSize` or (C, N) and $Src/Target$ are unrelated,
- `throw` e because $Src/Target$ and `NullPointer` are unrelated. σ and γ are unchanged by the transition, for xcpt, $\beta = (=)$ which is satisfied because $\mu($`throw` $e) =$ `throw` e
- $l[i] \leftarrow e$ and $l_1 \leftarrow l_2[i]$: the built-in exception and the array type itself is unrelated to both Src and $Target$.
- `init_class` C: the metaclass(C) is unrelated to Src and $Target$. Moreover, $\mu(\text{staticinitializer}(\Gamma(C))) = \text{staticinitializer}(\mu(\Gamma)(C))$ as can be seen from the definition of μ.

Composites The cases for $t_1;t_2$, `while`$(e)\{t\}$, `try`$\{t_1\}$`finally`$\{t_2\}$ and `try`$\{t_1\}$`catch`$(T\ l)\{t_2\}$ are treated analogously to the proof in chapter 4.1. As an illustration, consider $t_1;t_2$.

To make sure I am not concealing any mistakes here, let me write it down step by step and very carefully.

- It has to be shown that

$$(\Gamma \vdash s_0 \xrightarrow{t_1;t_2} s_2) \wedge \beta(s_0, s_0') \Rightarrow (\mu(\Gamma) \vdash s_0' \xrightarrow{\mu(t_1;t_2)} s_2') \wedge \beta(s_2, s_2')$$

- To show the implication, let's assume its left-hand side,

$$(\Gamma \vdash s_0 \xrightarrow{t_1;t_2} s_2) \wedge \beta(s_0, s_0')$$

- $\Gamma \vdash s_0 \xrightarrow{t_1;t_2} s_2$ must have been derived from the rule for chaining statements in table 3.2. Therefore $\Gamma \vdash s_0 \xrightarrow{t_1} s_1$ and $\Gamma \vdash s_1 \xrightarrow{t_2} s_2$

- With $\Gamma \vdash s_0 \xrightarrow{t_1} s_1$ and $\beta(s_0, s'_0)$ and according to the induction hypothesis, we can conclude that $\mu(\Gamma) \vdash s'_0 \xrightarrow{\mu(t_1)} s'_1$ and $\beta(s_1, s'_1)$.

- Because of $\beta(s_1, s'_1)$ and $\Gamma \vdash s_1 \xrightarrow{t_2} s_2$, we get again according to the hypothesis $\beta(s_2, s'_2)$ and $\mu(\Gamma) \vdash s'_1 \xrightarrow{\mu(t_2)} s'_2$

- Using the rule for chaining statements in table 3.2 on $\mu(\Gamma) \vdash s'_0 \xrightarrow{\mu(t_1)} s'_1$ and $\mu(\Gamma) \vdash s'_1 \xrightarrow{\mu(t_2)} s'_2$ together with the definition $\mu(t_1);\mu(t_2) = \mu(t_1;t_2)$, we deduce $\mu(\Gamma) \vdash s'_0 \xrightarrow{\mu(t_1;t_2)} s'_2$.

- Moreover, we still know that $\beta(s_2, s'_2)$, which is what we needed to prove as well.

The remaining non-trivial cases that potentially have to be guarded by local conditions are the following.

- Field update “$l.fld \leftarrow e$”

- Field read “$l_1 \leftarrow l_2.fld$”

- Object allocation “$l \leftarrow \texttt{new}\ C$”

The statements are examined in the order above. When there is more than one deduction rule for a statement, the discussion is organized in “cases” and “subcases”.

As mentioned on page 132, $l \leftarrow \texttt{new}\ C$ introduces an obvious problem with the data correspondence as defined in equation (5.2): The *target* link is Null after the allocation, but equation (5.2) requires that this link be initialized from the beginning. The solution is to weaken the requirement of β. This does not affect the proofs too much, so I decided to ignore object initialization for the other cases to make them more concise and clear and then to introduce during the discussion of object allocation this issue and to point out at that time, what is needed to take this additional consideration into account.

The following says that the values of expressions in the program that are given by the user do not change.

Lemma 5.1. *v introduced by transformation, e given by user.* $[\![e]\!]_{\Gamma}^{(\sigma,\gamma)} = [\![e]\!]_{\mu(\Gamma)}^{(\sigma[v\mapsto t],\beta(\gamma))}$

Reason: $v \notin \mathrm{FV}(l)$ and the only case where γ_r is used is for the `instanceof` operator. $\mathrm{rtt}(x) = \mathrm{rtt}(y)$ from equation (5.2) guarantees that its value does not change.

Field update "$l.fld \leftarrow e$"

We have to show that

$$(\Gamma \vdash s \xrightarrow{l.fld \leftarrow e} s') \wedge \beta(s, r) \Rightarrow (\mu(\Gamma) \vdash r \xrightarrow{\mu(l.fld \leftarrow e)} r') \wedge \beta(s', r')$$

Assuming the left hand side $\Gamma \vdash s \xrightarrow{l.fld \leftarrow e} s')$ and $\beta(s, r)$, there are two cases to be considered: exceptional and normal termination. We consider the derivation resulting in an exception (because of null pointers) as well. It serves as an example for all other cases where the conditions for regular execution are not satisfied and some of the cases above I do not treat in detail.

Case 1: Write Field with exception The additional assumption is according to table 3.4

$$[\![l]\!]^s_\Gamma = \mathrm{Null}$$

Technically, we would have to distinguish the case when the field being accessed is f ($fld = f$) and when it is not. I only treat the more direct case $fld \neq f$ explicitly. For $fld = f$, the exception is caused by read-access to $l.target$ in the translated version instead of the read access to $l.f$. The strategy is the same, but the chain and the exception propagation rule have to be used and combined.

- Assume $\Gamma \vdash s \xrightarrow{l.fld \leftarrow e} s'$ and $\beta(s, r)$
- According to table 3.4,

  $$s = (\mathrm{None}, \sigma, \gamma, \mathfrak{u}) s' \quad = (\mathrm{Some}\, loc, \sigma, \gamma[loc \mapsto \mathrm{init\_obj}(\Gamma, \texttt{NullPointer})], \mathfrak{u})$$

- $r = (\mathrm{xcpt}_r, \sigma_r, \gamma_r, \mathfrak{u}_r)$ is uniquely determined by the definition of β.
  - $\mathrm{xcpt}_r = \mathrm{None}$
  - $\sigma_r = \sigma[v \mapsto t]$
  - $\gamma_r = \beta(\gamma)$
  - $\mathfrak{u}_r = \mathfrak{u}$
- $\gamma_r(loc) = \mathrm{None}$ because of $\gamma(loc) = \mathrm{None}$ and equation (5.1).
- Assuming $fld \neq f$, we thus get the transition $\mu(\Gamma) \vdash r \xrightarrow{\mu(l.fld \leftarrow e)} r'$ with $r' = (\mathrm{xcpt}'_r, \sigma'_r, \gamma'_r, \mathfrak{u}'_r)$ and
  - $\mathrm{xcpt}'_r = \mathrm{Some}\, loc$
  - $\sigma'_r = \sigma[v \mapsto t]$

- $\gamma'_r = \gamma_r[loc \mapsto \text{init\_obj}(\mu(\Gamma), \texttt{NullPointer})]$
- $\mathfrak{u}'_r = \mathfrak{u}$

$\beta(\text{Some}\, loc, \text{xcpt}'_r)$ and $\beta(\sigma, \sigma'_r)$ are immediately obvious from the definition.[18] The third requirement,

$$\beta(\gamma[loc \mapsto \text{init\_obj}(\Gamma, \texttt{NullPointer})], \gamma_r[loc \mapsto \text{init\_obj}(\mu(\Gamma), \texttt{NullPointer})])$$

is satisfied because $\beta(\gamma, \gamma_r)$ and the objects at loc are β-compatible (they are equal and unrelated to both Src and $Target$).

Case 2: Write field without exception The additional assumption is $[\![l]\!]^s_\Gamma \neq \text{Null}$ according to table table 3.4.

Unlike in the case before, where we assumed $\mathit{fld} \neq f$ because it wouldn't have made a difference anyway, we have to consider at least two cases now: $\mathit{fld} = \mathit{target} \wedge \text{ctt}(\gamma(l)) \preceq_\Gamma Src$ and $\mathit{fld} = f \wedge \text{ctt}(\gamma(l)) \preceq_\Gamma Src$. Remember for the rest of this case that we can use the wisdom established in section 5.4.3: the value of a simple expression (like l and e) is never affected by the transformation. We write $x = [\![l]\!]^s_\Gamma$. $\gamma_r(x)(target) \neq \text{Null}$ is guaranteed by $\beta(\gamma, \gamma_r)$.

Subcase: Write field without exception where the field is *target* The additional assumption for this subcase is

$$\mathit{fld} = \mathit{target} \wedge \text{ctt}(\gamma(x)) \preceq_\Gamma Src$$

For this subcase, it has to be shown that

$$(\Gamma \vdash s \xrightarrow{l.target \leftarrow e} s') \wedge \beta(s, r) \Rightarrow (\mu(\Gamma) \vdash r \xrightarrow{l.target \leftarrow e} r') \wedge \beta(s', r') \tag{5.3}$$

- Assume $\Gamma \vdash s \xrightarrow{l.target \leftarrow e} s'$ and $\beta(s, r)$. Just as above, we have (again according to table 3.4), $s = (\text{None}, \sigma, \gamma, \mathfrak{u})$ and the assumption $\beta(s, r)$ leaves only one possibility for $r = (\text{xcpt}_r, \sigma_r, \gamma_r, \mathfrak{u}_r)$. As a reminder:
  - $\text{xcpt}_r = \text{None}$
  - $\sigma_r = \sigma[v \mapsto t]$
  - $\gamma_r = \beta(\gamma)$
  - $\mathfrak{u}_r = \mathfrak{u}$

[18]And so is $\beta(\mathfrak{u}, \mathfrak{u})$, which I omit for the rest of this proof.

- As a terminal state, we get

$$s' = (\text{None}, \sigma, \gamma[x \mapsto \gamma(x)[\mathit{target} \mapsto [\![e]\!]^s_\Gamma]], \mathfrak{u})$$

- Likewise for the transformed version, we get the final state ($[\![e]\!]^s_\Gamma$ unaffected by μ)

$$r' = (\text{None}, \sigma[v \mapsto t], \gamma_r[x \mapsto \gamma_r(x)[\mathit{target} \mapsto [\![e]\!]^s_\Gamma]])$$

- $\beta(\text{xcpt}(s'), \text{xcpt}(r'))$, $\beta(\sigma(s'), \sigma(r'))$ and $\beta(\mathfrak{u}(s'), \mathfrak{u}(r'))$ hold trivially.
- It remains to check the truth-value of $\beta(\gamma(s'), \gamma(r'))$:

$$\beta(\gamma[x \mapsto \gamma(x)[\mathit{target} \mapsto [\![e]\!]^s_\Gamma]], \gamma_r[x \mapsto \gamma_r(x)[\mathit{target} \mapsto [\![e]\!]^s_\Gamma]])$$

- We still know $\beta(\gamma, \gamma_r)$. This is how we derived it in the first place.
- Because of equation (5.1) and the way compliance is defined, it only remains to validate β-compliance of the locations that are potentially affected by μ. It translates to the condition

$$\beta(\gamma(x)[\mathit{target} \mapsto [\![e]\!]^s_\Gamma], \gamma_r(x)[\mathit{target} \mapsto [\![e]\!]^s_\Gamma])$$

- Because $\gamma(x) \preceq_\Gamma \mathit{Src}$, the first case in equation (5.2) is relevant:

$$\begin{aligned}\text{rtt}(\gamma(x)[\mathit{target} \mapsto [\![e]\!]^s_\Gamma]) = \text{rtt}(\gamma_r(x)[\mathit{target} \mapsto [\![e]\!]^s_\Gamma]) \wedge \\ \gamma(x)[\mathit{target} \mapsto [\![e]\!]^s_\Gamma](f) = \gamma_r(\gamma_r(x)[\mathit{target} \mapsto [\![e]\!]^s_\Gamma](\mathit{target}))(f) \wedge \\ f \lhd \gamma(x)[\mathit{target} \mapsto [\![e]\!]^s_\Gamma] = \gamma_r(x)[\mathit{target} \mapsto [\![e]\!]^s_\Gamma]\end{aligned} \quad (5.4)$$

  To evaluate the expression, remember that because of the assumption, $\beta(\gamma, \gamma_r)$. I.e., $\beta(\gamma(x), \gamma_r(x))$, which guarantees the three propositions

  - $\text{rtt}(\gamma(x)) = \text{rtt}(\gamma_r(x))$,
  - $\gamma(x)(f) = \gamma_r(\gamma_r(x)(\mathit{target}))(f)$ and
  - $f \lhd \gamma(x) = \gamma_r(x)$

- The first line in equation (5.4),

$$\text{rtt}(\gamma(x)[\mathit{target} \mapsto [\![e]\!]^s_\Gamma]) = \text{rtt}(\gamma_r(x)[\mathit{target} \mapsto [\![e]\!]^s_\Gamma])$$

  holds because $\text{rtt}(\gamma(x)) = \text{rtt}(\gamma_r(x))$.

- The third line in equation (5.4),

$$f \lhd \gamma(x)[\mathit{target} \mapsto [\![e]\!]^s_\Gamma] = \gamma_r(x)[\mathit{target} \mapsto [\![e]\!]^s_\Gamma]$$

  holds because $f \lhd \gamma(x) = \gamma_r(x)$ and the state update is to the same value.

- The second line in equation (5.4),

$$\gamma(x)[target \mapsto [\![e]\!]^s_\Gamma](f) = \gamma_r(\gamma_r(x)[target \mapsto [\![e]\!]^s_\Gamma](target))(f)$$

ndition

can be simplified to

$$\gamma(x)(f) = \gamma_r([\![e]\!]^s_\Gamma)(f) \tag{5.5}$$

This expression does not obviously hold, which is unfortunate but understandable: If we change the $Target$ object $l.target$ points to, how could we expect that the value of $Target.f$ remains the same? Equation (5.5) tells how: by making sure that the new $Target$ has a value in its f field that is consistent with the shadow field f in l. Not a very helpful proposition, to be honest.

The fact about the prestates we recorded before, $\gamma(x)(f) = \gamma_r(\gamma_r(x)(target))(f)$, can be combined with equation (5.5) and allows us to simplify the condition to something that can be checked in transformed program alone:

conservative approximation

$$\gamma_r([\![e]\!]^s_\Gamma)(f) = \gamma_r(\gamma_r(x)(target))(f) \tag{5.6}$$

condition

In other words, the f field of the newly assigned expression must be the same as the f field that is already present in the object pointed to by $x.target$, in program notation: $e.f = l.target.f$.

We can also derive conservative approximations by generalization from field values to object values and then to locations. This seems the only kind of semantic approximation that is evident. The following conservative conditions result.

- $\gamma_r([\![e]\!]^s_\Gamma) = \gamma_r(\gamma_r(x)(target))$
- $[\![e]\!]^s_\Gamma = \gamma_r(x)(target)$

The last one is what has been suggested at the beginning on section 5.4.3.

Subcase: Write field without exception where field accessed is f For this subcase, the additional assumption is

$$fld = f \wedge \text{ctt}(\gamma(x)) \preceq_\Gamma Src$$

$$(\Gamma \vdash s \xrightarrow{l.f \leftarrow e} s') \wedge \beta(s,r) \Rightarrow (\mu(\Gamma) \vdash r \xrightarrow{v \leftarrow l.target; v.f \leftarrow e} r') \wedge \beta(s',r') \tag{5.7}$$

- Assume $\Gamma \vdash s \xrightarrow{l.f \leftarrow e} s'$ and $\beta(s,r)$

- As above, we get[19] $s = (\text{None}, \sigma, \gamma)$ and $r = (\text{xcpt}_r, \sigma_r, \gamma_r)$ with
  - $\text{xcpt}_r = \text{None}$
  - $\sigma_r = \sigma[v \mapsto t]$
  - $\gamma_r = \beta_d(\gamma)$
- As an end state, we get ($x = [\![l]\!]^s_\Gamma$)

$$s' = (\text{None}, \sigma, \gamma[x \mapsto \gamma(x)[f \mapsto [\![e]\!]^s_\Gamma]], \mathfrak{u})$$

- The derivation tree for the right hand side of equation (5.7) can have only one form:

$$\dfrac{\boxed{\dfrac{x_l = [\![l]\!]^s_\Gamma \qquad x_l \neq \text{Null}}{\mu(\Gamma) \vdash (\text{None}, \sigma_r, \beta_d(\gamma), \mathfrak{u}) \xrightarrow{v \leftarrow l.target} (\text{None}, \sigma[v \mapsto \underbrace{\beta_d(\gamma)(x_l)(target)}_{x_l.target}], \beta_d(\gamma), \mathfrak{u})}} \quad \boxed{\dfrac{x_v = x_l.target \qquad x_v \neq \text{Null}}{\mu(\Gamma) \vdash (\text{None}, \sigma[v \mapsto x_l.target], \beta_d(\gamma), \mathfrak{u}) \xrightarrow{v.f \leftarrow e} (\text{None}, \sigma[v \mapsto x_l.target], \beta_d(\gamma)[x_v \mapsto \beta_d(\gamma)(x_v)[f \mapsto [\![e]\!]^{(\sigma,\gamma)}_{\mu(\Gamma)}]], \mathfrak{u})}}}{\mu(\Gamma) \vdash r \xrightarrow{v \leftarrow l.target; v.f \leftarrow e} (\text{None}, \sigma[v \mapsto x_l.target], \beta_d(\gamma)[x_v \mapsto \beta_d(\gamma)(x_v)[f \mapsto [\![e]\!]^s_\Gamma]], \mathfrak{u})}$$

- All the assumptions of the tree are satisfied from the corresponding transition in the original program.
- It remains to show that

$$\begin{aligned} &\beta((\text{None}, \sigma, \gamma[x \mapsto \gamma(x)[f \mapsto [\![e]\!]^s_\Gamma]], \mathfrak{u}), \\ &\qquad (\text{None}, \sigma[v \mapsto x_l.target], \beta_d(\gamma)[x \mapsto \beta_d(\gamma)(x_v)[f \mapsto [\![e]\!]^s_\Gamma]], \mathfrak{u})) \end{aligned} \tag{5.8}$$

- The first two and the last component, i.e., exception, local state and I/O, trivially satisfy β
- It remains to show that $\beta(\gamma[x \mapsto \gamma(x)[f \mapsto [\![e]\!]^s_\Gamma]], \beta_d(\gamma)[x_v \mapsto \beta_d(\gamma)(x_v)[f \mapsto [\![e]\!]^s_\Gamma]])$
- We have to show

$$\beta(\gamma[x \mapsto \gamma(x)[f \mapsto e_e]], \gamma_r[\underbrace{\gamma_r(x)(target)}_{f\text{-container}} \mapsto \gamma_r(\gamma_r(x)(target))[f \mapsto e_e]]) \tag{5.9}$$

[19]As promised, I am omitting the $\mathfrak{u}$ part because it remains trivially unaffected.

- The procedure is again the same as in the previous subsections: I write down the prepositions that are guaranteed by $\beta(\gamma, \gamma_r)$ and then derive some intuitively clear obligations from equation (5.9). This is the proof procedure we will continue to use for the other cases without further reference to it. To be able to do so, let me finally nail down consistent naming conventions for this section:

	Original	Transformed
Prestate	s, xcpt, σ, γ or γ_s	r, xcpt$_r$, σ_r, γ_r
Poststate	s', xcpt$_a$, σ_a, γ_a or xcpt$_{s'}$, $\sigma_{s'}$, $\gamma_{s'}$	r', xcpt$_b$, σ_b, γ_b or xcpt$_{r'}$, $\sigma_{r'}$, $\gamma_{r'}$

- Let's keep in mind the three propositions guaranteed by $\beta(\gamma, \gamma_r)$.
- We're checking the condition for the locations for which $\beta(\gamma_a(loc), \gamma_b(loc))$ is not be guaranteed by $\beta(\gamma, \gamma_r)$. Which locations are these? This, of course, depends on the definition of β for heap values. In general, all locations have to be examined for which a changed value could have an effect on the truth value of $\beta(\gamma_a(loc), \gamma_b(loc))$.

  The respective poststates are (cf. tree above):

$$\gamma_a = \gamma[x \mapsto \gamma(x)[f \mapsto e_e]] \tag{5.10}$$
$$\gamma_b = \gamma_r[\gamma_r(x)(target) \mapsto \gamma_r(\gamma_r(x)(target))[f \mapsto e_e]] \tag{5.11}$$

  The updated objects are: $\gamma(x)(f)$, $\gamma_r(\gamma_r(x)(target))(f)$. While $\gamma(x)(f)$ identifies a unique heap location x for the update, the same is not true for the second expression.

- The three relevant obligations are now examined for values that are potentially affected. Cf. $v_a(f) = \gamma_b(v_b(target))(f), f \triangleleft v_a = v_b$ and $v_a = f \triangleleft v_b$ in equation (5.2).[20]

[20]Required correspondences in our specific case for $v_a = \gamma_a(x)$ are the following:

$$\gamma_a = \gamma[x \mapsto \gamma(x)[f \mapsto e_e]] \tag{5.12}$$
$$\gamma_b = \gamma_r[\gamma_r(x)(target) \mapsto \gamma_r(\gamma_r(x)(target))[f \mapsto e_e]] \tag{5.13}$$
$$v_a = \gamma_a(x) = \gamma(x)[f \mapsto e_e] \tag{5.14}$$
$$v_a(f) = e_e \tag{5.15}$$
$$v_b = \gamma_b(x) = \gamma_r(x) \text{ (assume types unrelated, therefore } \gamma_r(x)(target) \neq x) \tag{5.16}$$
$$= \gamma(x) \text{ because } f \triangleleft \gamma(x) = globs_r(x) \tag{5.17}$$
$$v_b(target) = \gamma(x)(target) = \gamma_r(x)(target) \tag{5.18}$$
$$\gamma_b(\gamma(r)(target)) = \gamma_b(\gamma_r(r)(target)) \tag{5.19}$$
$$= \gamma_r(\gamma_r(x)(target))[f \mapsto e_e] \tag{5.20}$$
$$\gamma_r(\gamma_r(x)(target))[f \mapsto e_e](f) = e_e \tag{5.21}$$

The first obligation $f \lhd v_a = v_b$, which states that nothing else apart from f has been written, is known from the prestate $f \lhd \gamma(x) = \gamma_r(x)$.

Target consistency $\gamma_a(\gamma(x)(\mathit{target})) = f \lhd \gamma_b(\gamma(x)(\mathit{target}))$ is also quite clear as it can be directly reduced to the prestate condition: $\gamma_a(\gamma(x)(\mathit{target})) = \gamma(\gamma(x)(\mathit{target}))$ and $f \lhd \gamma_b(\gamma(x)(\mathit{target})) = f \lhd \gamma_r(\gamma_r(x)(\mathit{target}))[f \mapsto e_e]$

Consistency of field f is less easy: for $\mathit{loc} = x$, it is unproblematic

$$v_a(f) = \gamma_b(v_b(\mathit{target}))(f) \tag{5.22}$$
$$e_e = \gamma_b(\gamma(x)(\mathit{target}))(f) = e_e \tag{5.23}$$

Consistency of f for the other object cannot be proven because it does not hold in general. In other words, it is another correctness condition for the refactoring. We do not use the condition directly as a check for this refactoring. It does not seem intuitive enough:

$$\forall \text{reachable}\, \mathit{loc} \neq x : \gamma_a(\mathit{loc})(f) = \gamma_b(\gamma_b(\mathit{loc})(\mathit{target}))(f)$$

The prestate conditions says:

$$\forall \text{reachable}\, \mathit{loc} : \gamma(\mathit{loc})(f) = \gamma_r(\gamma_r(\mathit{loc})(\mathit{target}))(f)$$

Remember that

$$\gamma_a = \gamma[x \mapsto \gamma(x)[f \mapsto e_e]]$$

and

$$\gamma_b = \gamma_r[\gamma_r(x)(\mathit{target}) \mapsto \gamma_r(\gamma_r(x)(\mathit{target}))[f \mapsto e_e]]$$

If the propositions $\gamma(\mathit{loc})(f) = \gamma_a(\mathit{loc})(f)$ and

$$\gamma_b(\gamma_r(\mathit{loc})(\mathit{target}))(f) = \gamma_r(\gamma_r(\mathit{loc})(\mathit{target}))(f)$$

hold, the condition reduced to the condition of the prestate and it is satisfied. $\gamma(\mathit{loc})(f) = \gamma_a(\mathit{loc})(f)$ is certainly true because $x \neq \mathit{loc}$.

As for the second equality, we can be sure it holds if $\gamma_r(\gamma_r(x)(\mathit{target}))(f)$ is updated. This means that the equation $\gamma_b(\gamma_r(\mathit{loc})(\mathit{target}))(f) = \gamma_r(\gamma_r(\mathit{loc})(\mathit{target}))(f)$ is potentially falsified when $\gamma_r(\mathit{loc})(\mathit{target}) = \gamma_r(x)(\mathit{target})$[21]

condition

A sensible conservative semantic approximation is therefore:[22]

[21]Unless the value e_e is the same as the old value in $\gamma_r(\gamma_r(\mathit{loc})(\mathit{target}))(f)$, an unlikely case that we disregard.

[22]There is some indication that the original *precise* condition

$$\forall \text{reachable}\, \mathit{loc} \neq x : \gamma_a(\mathit{loc})(f) = \gamma_b(\gamma_b(\mathit{loc})(\mathit{target}))(f)$$

is practically superior because it also allows to concentrate dispersed equivalent data, a frequent

$$\forall\,\text{reachable}\,loc \neq x : \gamma_r(loc)(target) \neq \gamma_r(x)(target) \tag{5.24}$$

In plain English: the *target* fields of two *Src* objects must not point to the same object if the f field of at least one of them is written. This has been anticipated on page 132.

It is worth pointing out that the field f is introduced by the transformation. Other pointers to target objects will never write f. *Target* objects that are not pointed to by a *Src* object are not subject to this restriction either. Equation (5.24) is a bit tough to check. An additional bit could be used in *Target* objects to indicate whether they are or are not being used by a *Src* object to speed up checking.

Field read “$l_1 \leftarrow l_2.fld$”

The cases when $fld \neq f$ or the referenced field does not exist ($[\![l_2]\!]^s_\Gamma = \text{Null}$) are trivial.[23] I therefore assume $x = [\![l_2]\!]^s_\Gamma$, $x \neq \text{Null}$, $\text{rtt}(\gamma(x)) \preceq_\Gamma Src$ and $fld = f$. The poststates s_a and s_b for which we have to check β consistency are:[24]

$$s_a = (\text{None}, \sigma[l_1 \mapsto \gamma(x)(f)], \gamma) \tag{5.25}$$
$$s_b = (\text{None}, \sigma[l_1 \mapsto \gamma_r(\gamma_r(x)(target))(f), v \mapsto t], \gamma_r) \tag{5.26}$$

This time, it is easy to see that the exception part and the heap are β-compatible. What about the local variables? The definition mandates $\gamma(x)(f) = \gamma_r(\gamma_r(x)(target))(f)$. But this is exactly was is guaranteed by $\beta(\gamma, \gamma_r)$ (equation (5.2)).

Object allocation “$l \leftarrow \texttt{new}\ C$” (and how it alters the proofs above)

This statement looks pretty unproblematic – and it actually is! All that is created is a new object. If $C \not\preceq_\Gamma Src$, the required invariants hold trivially.

If $C \preceq_\Gamma Src$ however, one of the required invariants evaluates to false – unconditionally. Here is how:[25] The start states are γ and γ_r. For them, $\beta(globs, \gamma_r)$ holds. The end-states are $\gamma_a = \gamma[loc \mapsto \text{init\_obj}(\Gamma, Src)]$ and $\gamma_b = \gamma_r[loc \mapsto \text{init\_obj}(\mu(\Gamma), Src)]$.

intent of the “Move Field” refactoring. An example from the `TheBank` application that comes with this thesis: The field **`double`** `interest` is stored in class `Account`. If it turns out that `interest` depends only on the `Account`’s `accountType` field, `interest` can be moved to `AccountType` even though there are many accounts that point to a single `AccountType` instance.

[23] It is the same as for field updates.

[24] Remember that the required $\gamma_r(x)(target) \neq \text{Null}$ is guaranteed by $\beta(\gamma, \gamma_r)$.

[25] For notational convenience, we assume $C = Src$

The updated stores look like this: $\gamma_a = \gamma[loc \mapsto \{all_a \mapsto defvals\}]$ and for the transformed program: $\gamma_b = \gamma[loc \mapsto \{target \mapsto \text{Null}, all_a \mapsto defvals\}]$. Now let's check β-compatibility for $\gamma_a(loc)$ and $\gamma_b(loc)$:

$$\{all_a \mapsto defvals\}\,(f) = \gamma_b(\{target \mapsto \text{Null}, all_a \mapsto defvals\}\,(target))(f)$$

in equation (5.2). The left-hand side evaluates to the default value of f, but the right-hand side is undefined (the heap does not map Null to an ordinary heap value, see equation (3.14)). There is no associated target object that could contain the correct value for f. The equation "Some value = no value" is never satisfied, i.e., it is just false! Finding a strategy to satisfy the equation requires finding a way to weaken the requirement β. It ought to be a conservative weaking ($\beta \subset \beta_{\text{new}}$).

The are multiple options, (i.e., multiple different β_{new}s) that can be proposed to solve this problem. As noted previously, one possibility is to ask for establishment of the required invariant condition in the constructor. We pointed out that while this is feasible in formal specification and verification, it is not a satisfactory solution for refactoring. It treats code differently depending on from where it is called. It creates asymmetric obligations and it is an obstacle to truly local checking.

The simplest possibility is to admit that *target* fields are set to Null. The new β would then be:

$$\beta(v_a, v_b) \equiv \text{rtt}(v_a) = \text{rtt}(v_b) \wedge \begin{cases} v_b(target) = \text{Null} \vee (f \lhd v_a = v_b & \text{if } v_a \preceq_\Gamma Src \\ \quad \wedge\, v_a(f) = \gamma_b(v_b(target))(f)) & \\ v_a = f \lhd v_b & \text{if } v_a \preceq_\Gamma Target \\ v_a = v_b & \text{otherwise} \end{cases} \tag{5.27}$$

The proofs before have to be adapted: $x \neq$ Null where x is the location of the *Src* object has to be introduced explicitly as a precondition, i.e., a condition on the prestate. The problem is with write accesses to *target* itself. In the case when we assign a non-null object to *target*, the value of *target.f* has to match the shadow field's content in the untransformed version. How could this be checked? It cannot if we are erasing the field f in *Src*. It cannot be the aim of this formal analysis to conclude that the original and the transformed version have to be run side by side to find out whether the transformation was applied correctly![26] A simple but already sufficient alternative is to demand that initialization of *target* is monotonous. "Resetting" *target* is not possible. This does not affect the equivalence relation and can be readily tested locally in the transformed version of the program:

[26]For practical uses however, it is quite sufficient and even advisable if done automatically by a refactoring tool for testing.

$$\forall\, \mathrm{rtt}(\gamma_r(x)) \preceq_\Gamma Src : \gamma_r(x)(target) \neq \mathrm{Null} \Rightarrow \gamma_{r'}(x)(target) \neq \mathrm{Null} \tag{5.28}$$

equation (5.28) is an additional *sufficient* postcondition

This invariant has to be checked only for assignments to field *target* of *Src* objects.

The only advantage of this definition is that it can be ensured by local testing. We do however, not eliminate the difficulty of initial write accesses to *target*. (The precondition is sufficient to guarantee that *target* writes come before f-reads. This creates a problem with the normative condition for write accesses to *target* fields we found in equation (5.6). We derived it utilizing the precondition in β's definition and concluded that $\gamma_r([\![e]\!]^s_\Gamma)(f) = \gamma_r(\gamma_r(x)(target))(f)$ would guarantee preservation of β compatibility. This does of course hold if $\gamma_r(x)(target) \neq \mathrm{Null}$. For $\gamma_r(x)(target) = \mathrm{Null}$, there is only one defensible strategy. Demand that the assigned *target* has its f field set to the default value of $\mathrm{ctt}(f)$. The reason is that write accesses to f cannot have occured, the shadow field f therefore still has the default value, which is what should be returned by the next read access if there is any. This meta-argument cannot be used for the proof – it assumes a definite beginning in a properly initialized state, i.e., a state compatible with our assumption. The newly discovered invariant has to be made formal as part of β's definition, with which the formal proof becomes trivial:

$$\begin{aligned}\beta(v_a, v_b) \equiv\ & \mathrm{rtt}(v_a) = \mathrm{rtt}(v_b) \wedge \\ & \begin{cases} \begin{array}{l}(v_a(f) = \mathrm{defaultval}(Src.f) \wedge v_b(target) = \mathrm{Null}) \\ \vee\ (f \lhd v_a = v_b \wedge v_a(f) = \gamma_b(v_b(target))(f))\end{array} & \text{if } v_a \preceq_\Gamma Src \\ v_a = f \lhd v_b & \text{if } v_a \preceq_\Gamma Target \\ v_a = v_b & \text{otherwise} \end{cases}\end{aligned} \tag{5.29}$$

This definition of β renders monotonous initialization of *target* unnecessary. *target* could now be reset to Null if the field is set to the default value first. I do not consider this a realistic condition however.

It also has to be pointed out that this construction is not a necessary outcome of the investigation. Other solutions might have been possible. It is a conservative attempt, but not overly restrictive – at least not for the programs I can think of.

5.4.4. Summary of the proof structure in this chapter

1. Assume a derivation for $\Gamma \vdash s \xrightarrow{t} s'$ and assume $\beta(s, r)$, which then determines r as $\beta(s)$.
2. Consider a derivation for $\Gamma \vdash r \xrightarrow{t} r'$. There is only one possibility because the semantics is deterministic.

3. The checkable condition is "$\beta(s', r')$ and the assumptions needed for step 2 under the assumptions established in step 1". This (minimal) condition has to be jointly implied by the pre- and postconditions. I.e., pre- and postconditions are conservative approximations.

5.5. Syntactic approximations

This chapter sketches some syntactic methods and analyses to conservatively approximate the local conditions that must be satisfied. For each approximation, we repeat the desired condition and the more conservative condition the syntactic condition is supposed to guarantee. Time and space do not allow to check the syntactic conditions for their soundness.

Summary of normative conditions

The original Γ and the transformed $\mu(\Gamma)$ version of the program are equivalent wrt. equation (3.16), section 5.4.3 and equation (5.29) if, for every execution of original t and transformed $\mu(t)$, the following holds in $\mu(\Gamma)$:

t	Cond. for execution	Check
$l.target \leftarrow e$	$\gamma_r(x)(target) = \text{Null} \wedge$ $x = [\![l]\!]^s_\Gamma \neq \text{Null} \wedge \text{ctt}(\gamma_r(x)) \preceq_\Gamma Src$	$[\![e]\!]^s_\Gamma = \text{Null} \vee \gamma_r([\![e]\!]^s_\Gamma)(f) =$ $\text{defaultval}(f)$
$l.target \leftarrow e$	$\gamma_r(x)(target) \neq \text{Null} \wedge$ $x = [\![l]\!]^s_\Gamma \neq \text{Null} \wedge \text{ctt}(\gamma_r(x)) \preceq_\Gamma Src$	$\gamma_r([\![e]\!]^s_\Gamma)(f) = \gamma_r(\gamma_r(x)(target))(f)$
$l.f \leftarrow e$	$x = [\![l]\!]^s_\Gamma \neq \text{Null} \wedge \text{ctt}(\gamma_r(x)) \preceq_\Gamma Src$	$\gamma_r(x)(target) \neq \text{Null} \wedge$ $\forall \text{reachable}\, loc \neq x :$ $\gamma_r(loc)(target) \neq \gamma_r(x)(target)$
$l_1 \leftarrow l_2.f$	$x = [\![l_2]\!]^s_\Gamma \neq \text{Null}, \text{rtt}(\gamma(x)) \preceq_\Gamma Src$	$\gamma_r(x)(target) \neq \text{Null}$

If these checks succeed, the state correspondences between Γ and $\mu(\Gamma)$ are retained.

Syntactic approximations: assignments to $target$

For assignments to $Src.target$ ($l.target \leftarrow e$), the following precondition has to be checked if $x = [\![l]\!]^s_\Gamma \neq \text{Null}$:

$$\begin{aligned}
&\textsf{if } \gamma_r(x)(target) = \text{Null}\\
&\textsf{then } [\![e]\!]^s_\Gamma = \text{Null} \vee \gamma_r([\![e]\!]^s_\Gamma)(f) = \text{defaultval}(f)\\
&\textsf{else } \gamma_r([\![e]\!]^s_\Gamma)(f) = \gamma_r(\gamma_r(x)(target))(f)
\end{aligned}$$

For identifying syntactic conditions, it might be useful to split the problem into two parts. Ensure that (1) a non-null $target$ is never overwritten with Null, (2) if a null *target* is overwritten, it must be overwritten with a *Target* object whose f is set to $w = \text{defaultval}(f)$, (3) if an update happens to an already initialized *target*, the f field must not be modified.

Tackling (1) and (2) using static analysis The first two problems can be easily framed as a data-flow problem that can be solved with a simple forward analysis. The fact that only *Src* and *Target* objects and assignments to *target* have to be examined makes the procedure feasible and fast. It is best to keep different data-flow information for *Src* and for *Target*. The kind of information we provide is directly derived from the validity conditions:

$$l.target{\leftarrow}e \text{ is valid} \iff \begin{cases} \text{if } l = \text{Null} \\ \text{if } l \neq \text{ Null} \wedge l.target = \text{Null} \wedge e = \text{Null} \\ \text{if } l \neq \text{ Null} \wedge l.target = \text{Null} \wedge e \neq \text{ Null} \wedge e.f = w \\ \text{if } l \neq \text{ Null} \wedge l.target \neq \text{ Null} \wedge e \neq \text{ Null} \wedge e.f = x.target.f \end{cases} \tag{5.30}$$

For *Target* e, the states are

$$\{e = \text{Null}, e \neq \text{ Null} \wedge e.f = w, e \neq \text{ Null} \wedge e.f \text{ arbitrary}\}$$

For *Src* l, the relevant states are (if $l.target$ is a separate entity in the analysis, there could be only two states: $l = \text{Null}$ and $l \neq \text{Null}$)

$$\{l = \text{Null}, l \neq \text{ Null} \wedge l.target = \text{Null}, l \neq \text{ Null} \wedge l.target \neq \text{ Null} \wedge l.target.f = w, \\ l \neq \text{ Null} \wedge l.target \neq \text{ Null} \wedge l.target.f = \text{arbitrary}\} \tag{5.31}$$

We use the textual representation of the conditions as labels for the states. Names or entities point to abstract locations for every program point (may-alias information). Following the notation in [34, chap. 10], for a program point p and a name v, $Alias(p, v) = L$ means that v can point to any of the abstract locations in L. For the sake of the analysis, Null is also considered a location. This is the crucial point for the unification of typestates and non-null types. With every location, we associate a subset of the possible states above. We initialize the subset to the appropriate condition (i.e., $l = \text{Null}$ resp. $e = \text{Null}$ for Null, $l \neq \text{ Null} \wedge l.target = \text{Null}$ for `new` *Src*, $e \neq \text{ Null} \wedge e.f = w$ for `new` *Target* and $\emptyset$ otherwise). We then propagate the possible states. Control flow joins are set merges. Precision could be increased in cases we have perfect alias information, i.e., if $a \text{ alias}_{\text{must}}\, b$, then state-updates to a also update the typestate of b destructively (as opposed to just adding another possible state to b). This is comparable to “strong” updates in [46].

This procedure is guaranteed to terminate because there is an upper bound of elements in the set and the transfer function is monotone.

If the alias analysis is modular, the data-flow analysis can also be made modular by abstracting methods by transfer functions.

The result of the analysis is the annotated program that can then be checked for compliance with the necessary correctness criterion. Moreover, the algorithm identifies the assignments to $target$ for which equality of $l.target.f$ and $e.f$ must hold. This is what we shall be using below.

Algorithms like this abound and their correctness is straightforward to prove. There is hardly anything that I can contribute to such an algorithm or its explanation.

The proposed solution raises another, important objection however: is the resulting containment a syntactic restriction as advertised in the title of this chapter? Like every static analysis, data-flow analyses try to interpret the behavior of programs at runtime. It is not just a matter of giving for example a CFG whose instances guarantee the desired properties. In that respect, it is not a purely syntactic. This is true for all but the simplest restrictions however. A different argument has to be found.

The main argument in favour of the analysis is this: Restrictions of this kind are normally considered good software development practice. Programmers may intuitivitely adhere to these restrictions and programs are more likely to satisfy them. All intuitively sensible restrictions have been (or at least will be) expressed by a corresponding type system. A type system could be used to ensure exactely the properties we are trying to establish here: Finite state type system for instance ("typestates") allow to express conditions that are satisfied by objects at different times during their lifetime. The model presented here can be translated to a specialized type system based e.g. on [12]. Type systems are syntactic restrictions. It is possible to formulate them as a set of constraints over every semantically meaningful syntactic class. The program satisfies the condition if there is a satisfying assignment for all meta-variables. Such a non-constructive attempt would probably be less useful than a static analysis.

I do personally believe that the analysis suggested here does have unique features that sets it apart from other analyses and I claim it is more useful than simple typestates because it combines non-null types with typestates. Typestates are associated with objects, not with names, i.e., they abstract invariants over an object's state. Even though typestates solve the problem prevalent in imperative languages that objects change over time, it does not extend to the earliest phase of object initialization: before the object comes into existence. This responsibility is accepted by non-null types. Non-null types however provide stepwise initialization only in a very limited context. Combining typestates and non-null types solves the problems both approaches have. Allowing sets of typestates instead of one specific typestates yields more flexibility. This idea is explicitly acknowledged in [12, sec. 3].

Tackling (3) using another static analysis? – probably not The previous subsection has led us with a program where some of the assignments to $target$ have to satisfy $\gamma_r(\llbracket e \rrbracket^s_\Gamma)(f) = \gamma_r(\gamma_r(x)(target))(f)$. An approximation could be to demand $\llbracket e \rrbracket^s_\Gamma = \gamma_r(x)(target)$. Copy propagation analysis could provide the necessary information for both criteria. If we require $\llbracket e \rrbracket^s_\Gamma = \gamma_r(x)(target)$, the statements that always satisfy this condition are probably programming errors. It is also unlikely that we can show that the f fields are the same for some objects. Furthermore, as far as I know, keeping strict correspondence between field values of different objects is not considered good software development practice.

This would mean that we forbid write access to $target$ unless we know that the right-hand side of the assignment is either Null or an object whose f is w. This seems to be quite restrictive. It is almost equivalent to having the field $target$ declared **`final`** but not requiring that the initialization definitely happens during construction.

Syntactic approximations: Assignment to f

For assignments to the field f we do have the condition

$$\forall \,\text{reachable}\, loc \neq x : \gamma_r(loc)(target) \neq \gamma_r(x)(target)$$

I.e., there must be at most one *Src.target* field pointing to every $Target$ object. There have been a lot of proposals for confining points-to relations between objects. Most of them are based on the idea of "ownership". Ownership type systems structure the heap into contexts and limit the point-to relations between them. This seems too restrictive in the limited context treated here: It is indeed allowed for any part of the program to have any number of pointers to arbitrary $Target$ objects. Access to $Target$ objects should thus not be restricted beyond the simple criterion mentioned above.

If we accept the restriction that the $Target$ object is created inside Src, the Universe type system provides an apt framework for reasoning about the problem. Imagine $target$ is declared as **`rep`** inside Src. Let it be the only annotation we introduce for fields in the program. It is still possible to pass around arbitrary pointers to the enclosed object. These references will be readonly to prevent modifications to the internal state of the $Target$ object. We do not care about modifications however and we could just ignore the readonly attribute for access to methods and fields of $Target$. When writing *Src.target* however, ownership types must be taken into account to guarantee that no aliasing through $target$ occurs. All right-hand sides must be **`rep`**. Inference of annotations could work by first introducing **`rep`** for all the entities that require it and then finding an type assignment that satisfy all the restrictions as implemented by Nathalie Kellenberger.

Note that access to $Target$ objects that are not reached through a $target$ field will not lead to accesses of $target.f$ and do not have to be specifically protected.

Syntactic approximations: Reading f

We have to show $\gamma_r(x)(target) \neq$ Null. If $x.target$ were Null, an exception would occur invalidating state equivalence. *The typestates infered in section 5.5 can be used to guarantee this condition.*

If things are really simple, *target* could be declared **`final`** and adherence could be checked (requires definite assignment during “construction”, which does not exist in our language. It has to be made sure that assigned values are non-null.).

6. Implementation Example

Warning. This chapter is largely unrelated to the theoretical material in the rest of this text. It provides a tutorial-like example that illustrates how the material can be implemented in practice. It is also a technical description of some important mechanisms in Visual Studio and Eclipse.

This chapter describes two prototypical implementations of refactoring tools for Spec# and Java implementing the "Move Field" refactoring. The refactoring tools differ from conventional tools in that they add specifications to the program – just as described in the previous chapters.

The goal of this chapter is twofold:

- The tools serve as a demo to illustrate what refactoring with specifications feels like, not as a development aid. For that reason, only the "Move Field" refactoring is implemented. They are not intended to be used in development. Instead, they are designed to be as minimally invasive as possible and to use as few internals of the IDEs they are integrated with as possible.
- The tools are implemented in Visual Studio and Eclipse. These are two of the most prominent IDEs on the market. This chapter is written in a *tutorial-like* manner. After reading it, you will know (i) how to write a simple plugin in Visual Studio and Eclipse (ii) how to implement an Eclipse Quick Assist extension. (iii) how to use Eclipse's `ASTRewrite` framework to implement refactorings and (iv) how to use the Microsoft common compiler infrastructure.

MSR's Spec# language supports contracts, invariants, non-null types, etc. Its specification language is a bit more powerful than what is possible with Java assertions. I have chosen Java and Spec# as examples for the following reason: For a refactoring tool to be useful, it has to be tightly integrated with a development environment. Currently, there are only two languages that are sufficiently supported with tools and at the same time have sufficiently expressive specification capabilities built-in: JML and Spec#.[1]

[1] This rules out conventional C# and Visual Basic. C/C++[2]and Java support simple assertions (via the **`assert`** keyword). This facility is very useful but not quite powerful enough for the conditions required by "Move Field". Additional code instrumentation would be necessary (like adding a reference count to target objects). Unfortunately, there is no standard means to declare code structures that are soley used for assertion checking. It would actually be preferable to express all assertions using only the language's vocabulary as this renders composition with assertions trivial: The assertions could just be transformed together with the source-code.

JML integration is provided for Eclipse by a tool called JMLEclipse [24]. JMLEclipse just patches the original JDT[3] that comes with the Eclipse SDK instead of providing a separate plugin. While it makes perfect sense to reuse the JDT code, I am not sure it is the best idea to just put your own code into a framework that is actively being developed: The latest version of JMLEclipse is for Eclipse 3.0. However, the current version of Eclipse is 3.2, differing sustantially from the 3.0 release.

Spec# [3] is integrated in Visual Studio as a separate "VSPackage" (explained below) that is installed just like any other language package. Even though Spec# is based on C#, the Visual Studio support is separate. It seems however that the Spec# plugin is maintained as an integral part of the Spec# project. Still, refactoring is not supported for Spec# although it is for C#.

The refactoring tools are neither directly implemented for the JMLEclipse plugin nor for Spec#'s Visual Studio integration: Instead, I have written them for Eclipse's Java Development Kit and for C# in Visual Studio (even though Spec# code is generated). This is a viable alternative: JMLEclipse is based on the JDT and the refactoring tool should be usable with JMLEclipse as well once it is available for Eclipse 3.2. The Eclipse tool produces very basic Java **`assert`** in addition to JML assertions. Spec# is based on C# and refactoring will probably become available for Spec# as well. The tool is then usable from Spec# without modifications.

It is much easier to make an external refactoring tool look like a built-in one in Visual Studio than in Eclipse: In particular, there are no Eclipse "Extension Points" for custom refactorings or source transformations in general.[4] Unlike for Visual Studio, it is not easily possible to augment the refactoring sub-menu in the code window context menu.

The rest of the chapter is organized as follows: I first present the design, implementation and use of the refactoring plugin for Visual Studio and describe its limitations. I then show how a corresponding Eclipse refactoring plugin can be written. The Eclipse plugin extends the Quick Assist extension point and therefore behaves differently from the built-in refactorings. I then compare the two implementations. This chapter can also serve as a quick introduction to the plugin mechanisms of the two IDEs.

6.1. Writing a plugin for Visual Studio 2005

The plugin presented in this section is a proper Visual Studio plugin. It is not a Macro and it is not a VSPackage.

[2]Yes, **`assert`** is a macro in C/C++.

[3]Java Development Tools

[4]JDT still provides too few possibilities and standard means to extend the IDE. After all, there is a reason that JMLEclipse was not implemented *on top of* the existing toolkit.

Visual Studio automation overview Macros are written in VBA and developed in what is called the Macros IDE, based on Visual Studio. Macros can also be recorded, offering an easy way to explore the Visual Studio automation model including all automation objects provided by the various built-in programming language packages. Macros are also easy to debug because they can be changed and restarted without restarting Visual Studio.

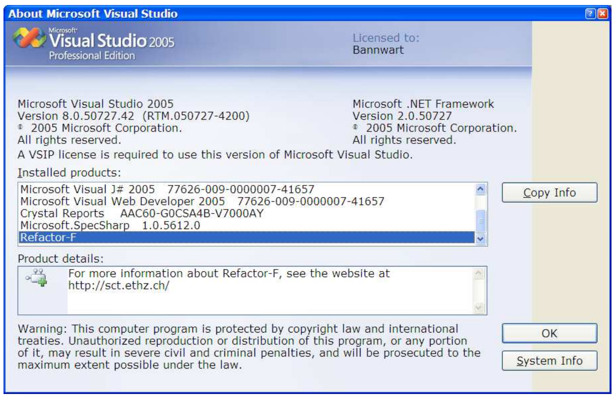
Macros

Plugins are more complex than macros. They are autonomous COM objects that attach to the Visual Studio environment. They implement the `IDTExtensibility2` COM interface. Traditionally, Visual Studio plugins are called add-ins.

Add-ins, plugins

Now how do macros differ from plugins in their abilities? Of course, plugins are compiled components, but that does not make them worse (or better)! There are however things that cannot be done with macros [32]:

1. Add the plugin to the “Options” and “About” dialogs. (see figure 6.1)
2. Create tool windows and sophisticated dialogs (see figure 6.6)
3. Disable and enable commands in menus and toolbars (see figure 6.3)


About Microsoft Visual Studio

Licensed to:
Bannwart

```
Microsoft Visual Studio 2005                        Microsoft .NET Framework
Version 8.0.50727.42  (RTM.050727-4200)             Version 2.0.50727
© 2005 Microsoft Corporation.                       © 2005 Microsoft Corporation.
All rights reserved.                                All rights reserved.
A VSIP license is required to use this version of Microsoft Visual Studio.
```

Installed products:

```
Microsoft Visual J# 2005   77626-009-0000007-41657
Microsoft Visual Web Developer 2005   77626-009-0000007-41657
Crystal Reports    AAC60-G0CSA4B-V7000AY
Microsoft.SpecSharp   1.0.5612.0
Refactor-F
```

Product details:

For more information about Refactor-F, see the website at
http://sct.ethz.ch/

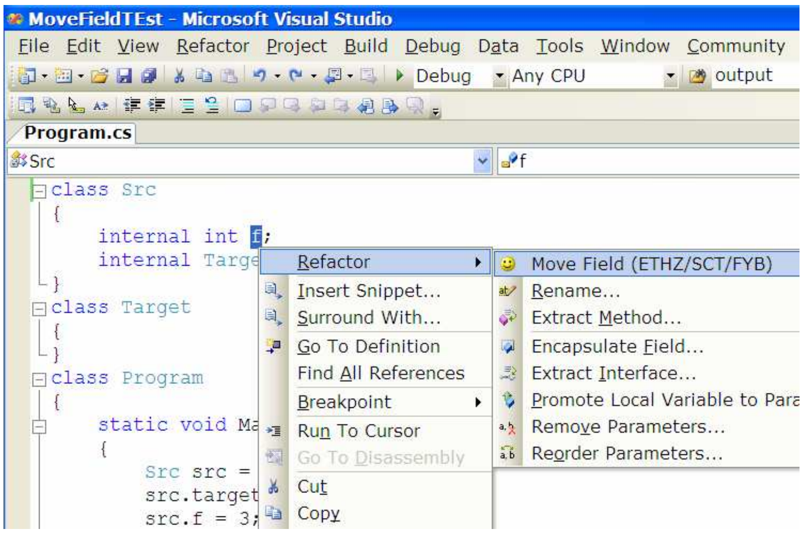
Warning: This computer program is protected by copyright law and international treaties. Unauthorized reproduction or distribution of this program, or any portion of it, may result in severe civil and criminal penalties, and will be prosecuted to the maximum extent possible under the law.

Figure 6.1.: “Move Field” plugin is added to the “About” box. This is impossible with macros.

Plugins are registered in XML files in the user’s home directory. This has changed since Visual Studio 2003.

VSPackages are even more powerful. Support for a new programming language for instance is implemented as a VSPackage. VSPackages can provide custom editors, project types, dialogs, etc. VSPackages have to be registered in the Windows Registry.

The refactoring tool is written as a plugin. Macros are not suitable in this case because most of the code in the tool is interacting with external C# libraries – the Spec# compiler and the compiler support packages. If the libraries are written in C#, it is also a good idea to write the tool itself in C#. Plugins can also be loaded at start-up time. This allows some initialization code to be run like adding entries to the context menu.



```
class Src
{
    internal int f;
    internal Targe
}
class Target
{
}
class Program
{
    static void Ma
    {
        Src src =
        src.target
        src.f = 3;
```

Figure 6.2.: The tool is integrated into the context menu.

6.1.1. Implementation description

The plugin is connected by a XML file with the extension `.Addin`. It has to be placed in the user's `Visual Studio 2005/Addins` directory and contains all the information the IDE needs to load the plugin. Here is the relevant excerpt.

```
<Assembly>..\Projects\FormalRefactoring\
   FormalRefactoring\bin\FormalRefactoring.dll</
   Assembly>
<FullClassName>FormalRefactoring.Connect</
   FullClassName>
<LoadBehavior>0</LoadBehavior>
<CommandPreload>1</CommandPreload>
```

My refactoring plugin uses C# as an implementation language and therefore uses the .NET wrappers instead of implementing a COM object directly. The assembly is specified first, followed by the qualified class name that will be used to instantiate the connection class that mediates between the plugin and the environment. Methods of this class are called when the plugin is loaded, unloaded, the IDE starts, etc.

`LoadBehavior 0` means that the plugin is not started when the IDE is started. This is just important for debugging. `CommandPreload` is 1 if the plugin setup mechanism should be invoked only once instead of each time the plugin is loaded.

Connect code The class invoked by the Visual Studio environment is

```
FormalRefactoring.Connect
```

The method that is called first, as a notification that the plugin is being loaded, is

```
OnConnection(
  object application, ext_ConnectMode connectMode,
  object addInInst, ref Array custom)
```

The `application` parameter is the top level object in the Visual Studio automation model. The `connectMode` parameter is `ext_ConnectMode.ext_cm_UISetup` if the plugin should install itself. If that case, the “Move field” command is installed into the “Refactor” sub-menu of the context menu of the code window. The menu-item is associated with a command string.

When the command’s availability is queried or it should be executed, the following plugin methods are called

```
QueryStatus(string commandName,
  vsCommandStatusTextWanted neededText,
  ref vsCommandStatus status, ref object commandText)
Exec(string commandName,
  vsCommandExecOption executeOption, ref object varIn,
  ref object varOut, ref bool handled)
```

In my plugin, the commands (at the moment only one to be precise) are stored in a dictionary that is consulated at each `QueryStatus` and `Exec` invocation. The invocations are dispatched to the corresponding `UserCmd` item.

```
private static Dictionary<string, UserCmd>
   userCommands
  = new Dictionary<string,UserCmd>();
static Connect() {
    foreach(Type t in UserCmd.userCmds) {
        UserCmd cmd = (UserCmd)Activator.
           CreateInstance(t);
```

```
            userCommands.Add("FormalRefactoring.Connect."
               + cmd.Name, cmd);
        }
    }
```

The important pieces are in the `MoveField` class that extends the abstract class `UserCmd`. `MoveField.Exec` implements the actual transformation.

First, the selected field is retrieved using the current document's code model that allows access to "programmatic constructs in a source code file". Unfortunately, it is not supported for Spec#, so I am expecting C# editor windows here (the resulting code will not be C#, but that's a different story). In any case, it is a pragmatic decision to accept this limitation – it would have been much more cumbersome to use the Spec# compiler to get hold of code elements corresponding to the current selection. The code element is retrieved in

```
    selectedCodeElement(con, out doc, out projitem, out
       codeElement);
```

which basically encapsulates a call to the corresponding environment routines.

In `Exec`, the fields that may point to the field are examined and presented in a dialog box to the user

```
    CodeVariable codeVariable = (CodeVariable)codeElement;
    CodeType cls = (CodeType)codeVariable.Parent;
    string fieldName = codeVariable.Name;
    List<string> possibleTargets = new List<string>();

    foreach(CodeElement m in cls.Members) {
        if(m.Kind == vsCMElement.vsCMElementVariable) {
            CodeVariable tvar = (CodeVariable)m;
            if(tvar.Type.TypeKind == vsCMTypeRef.
               vsCMTypeRefCodeType && tvar.Name !=
               fieldName) {
                possibleTargets.Add(tvar.Name);
            }
        }
    }
```

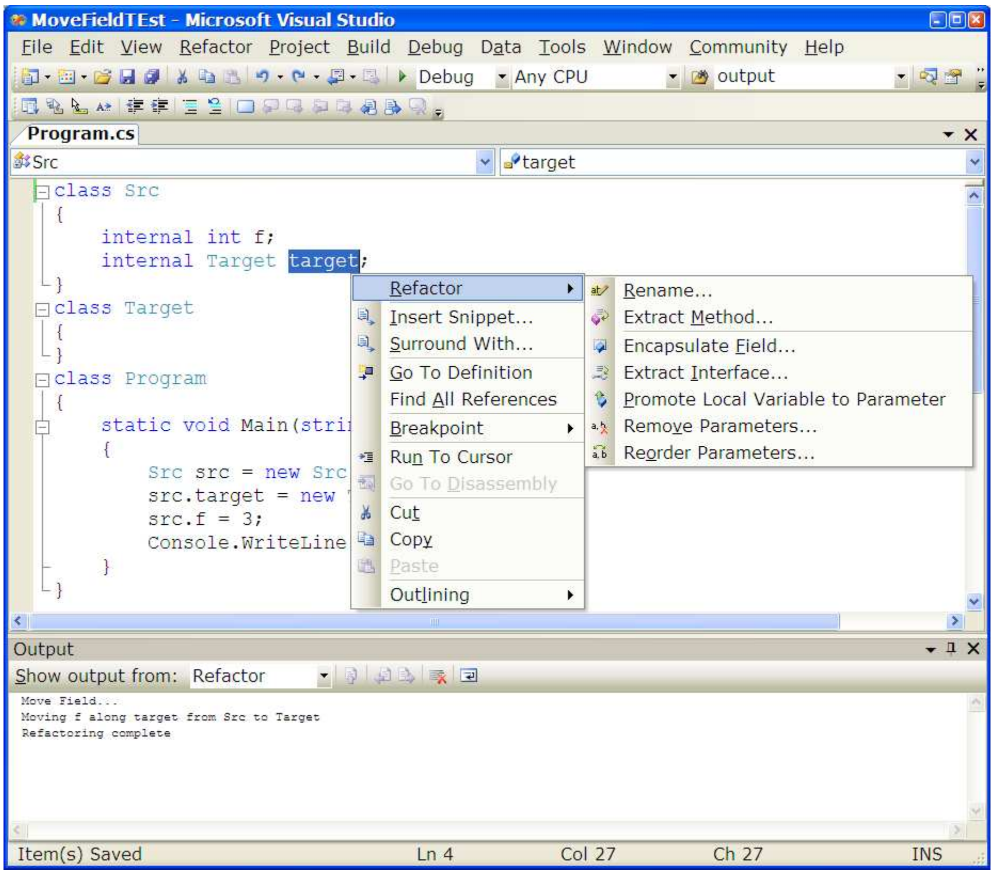
Having selected a field, the actual refactoring method

```
            movefield.MoveFieldRefactoring.performF
```

is called that analyzes and rewrites all changed files in the project. If there are compilation errors, they are printed.

```
if(!movefield.MoveFieldRefactoring.performF(true,
    files, fromClass, toClass, fieldName, target, out
       errs)) {
    foreach(string err in errs)
        con.message(err);
    return;
}
```

That's basically all the connect class contains! Querying the status of the command has been omitted. It is just like the actual transformation without calling `performF` at the end.



```
class Src
{
    internal int f;
    internal Target target;
}
class Target
{
}
class Program
{
    static void Main(stri
    {
        Src src = new Src
        src.target = new '
        src.f = 3;
        Console.WriteLine
    }
}
```

```
Move Field...
Moving f along target from Src to Target
Refactoring complete
```

Figure 6.3.: "Move Field" is not available if the text selection isn't near a field declaration for which the refactoring is applicable.

Refactoring code The plugin calls `performF` as the last step. `performF` takes the following arguments:

- **`string`** `fromclass, toclass, field, target` the fully qualified names of the syntactic entities involved in the transformation.
- **`out string`**`[] errors` messages that made the refactoring fail and have to be reported to the user

The method `perform` actually coordinates the transformation: First, the files are compiled by the Spec# compiler. Secondly, changes to the original source code are derived from the AST returned by the compiler and the context information contained therein.

Spec#'s compiler is based on `System.Compiler`, which is sometimes refered to as Microsoft's common compiler infrastructure (CCI). It is supposed to be used by all languages that use the .NET framework, including HScript, EcmaScript, Zonnon, Comega (C_ω), X++, Spec#, Sing#, Xaml and C/AL.

AST nodes are also defined in `System.Compiler`, which means that the nodes are the union of all node kinds supported by all languages, including the CLR IL itself. The nodes are rewritten destructively in various passes to resolve references, overloads, etc. and to reduce the tree to a model over CLR IL, which is then serialized. The AST nodes are defined as concrete classes and unlike in Eclipse's code model, there are no interfaces that hide the concrete implementation. The CCI nodes are just *records* of public fields, i.e., the code model is *passive*. All fields can be **`null`** in case a value cannot be resolved or there is some other error in the soure code. There are no explicit guarantees as to which values can be expected. Most AST nodes are supposed to be used from more than one language and they do not faithfully render the abstract syntax of the tree (unlike the Eclipse AST nodes).[5]

Within `perform`, the compilation is factored into a separate `compile` routine that handles the intricacy of instatiating compilation units and parsing them.

```
SpecSharpCompilerOptions options = makeOptions();
Compilation c = compile(fileNames, compiler, options,
   out results);
if(c == null) return false;
```

Compilation is done by invoking methods on the `compiler` parameter. A new method had to be introduced that resolves all overloads in a compilation unit but does not simplify the parse tree. It is called `ResolveParseTreeNoReduce`. The strings identifying the fields and classes involved are looked up and the result is stored as AST nodes.

[5] It is clear that this concept of a universal source-level IR is not universally appreciated and the Zonnon compiler for instance uses its own IR that is then translated to `System.Compiler` nodes [20].

`fromclass` becomes `fc`, `toclass` becomes `tc`, `field` becomes `ff`, `tf` becomes `tf`. The lookup is simply done by iterating over all global classes in the compilation unit, even though the `Module` and `TypeNode` classes provide methods to look up types and fields:

-
  ```
  TypeNode Module.GetType(Identifier Namespace,
                          Identifier name)
  ```

  returns the type with the specified name in the specified namespace or null if such a type cannot be resolved. It does not search referenced assemblies. To do so, there is the method

  ```
  TypeNode GetType(Identifier Namespace,
                   Identifier name,
                   bool lookInReferencedAssemblies)
  ```
- `Field TypeNode.GetField(Identifier name)`

  returns the field declared by this type node with the specified name or null if there is no such field.

Unfortunately, interesting source components that could further illustrate how to use other components of the CCI are not available in the source distribution of Spec#. The Visual Studio integration for instance is not shipped.[6] If it were, the tool could have been directly implemented inside the Spec# language package.

Transformation The transformation itself is kept simple. As I said above, a proper refactoring tool needs some serious infrastructure like the one presented in section 6.2. For this simplistic plugin, I just provide what is absolutely necessary to make the refactoring work for simple demo examples. The refactoring is specified as a number of `SourceChange`s.

```
List<SourceChange> changes =
  new List<SourceChange>(100);
```

`SourceChange` is a struct defined in `System.Compiler`. It describes a textual replacement in the source code. The list of changes is first collected and then applied to the source files.

Changes happen at every access of the field `ff`. These changes are collected by an AST visitor that extends `StandardVisitor`. This visitor class is used to transform the tree, that's why all the `visit*` methods return a value of the node type they visit. This feature is not used here and the return value of the default implementation is returned, which is just the identity. In CCI, the "dot" expression x.`ff` is translated to a `MemberBinding` node in the AST after symbol-table lookup.

[6]`Microsoft.SpecSharp.Package`

```
public override Expression VisitMemberBinding(
   MemberBinding qid) {
    if(qid.BoundMember == ff) {
        SourceChange sc = new SourceChange();
        sc.SourceContext = qid.BoundMemberExpression.
           SourceContext;
        sc.ChangedText = tf.Name + "." + ff.Name;
        changes.Add(sc);

        todo.Add(qid);
    }
    return base.VisitMemberBinding(qid);
}
```

`BoundMemberExpression` is the right-hand side of a dot expression, i.e., `ff` in case the field has the name of the variable that identifies it. This `SourceContext` is replaced by `tf.ff`, i.e., which is the string `tf.Name + "."+ ff.Name`.

The `todoFF` list contains all accesses of `ff` in a statement. `todoTF` contains the accesses of `tf`. For these accesses, assert-statements have to be generated. Needless to say, this only works if sub-expressions do not have side effects. This is the case in the sub-set discussed in this text, but will not be true in general for Spec#. For demo purposes, I considered it enough. In the code below, `someStmts` contains all the node types that are considered statements for which assertions have to be generated. `Aux.addLineBefore(changes, n, txt)` inserts a line of text before node `n`, taking into account the indentation of the next line, but nothing else.

```
public override Node Visit(Node node) {
    Node n = base.Visit(node);

    if(someStmts.ContainsKey(n.NodeType)) {
        foreach(MemberBinding mb in todoFF) {
            string txt;
            if(todoLhs.ContainsKey(mb)) {
                // this f-access is on the left
                txt = getAssignAssertionFF(mb, todoLhs
                   [mb]);
            } else // this is on the right
                txt = getReadAssertionFF(mb);

            Aux.addLineBefore(changes, n, txt);
        }
```

```
            foreach(MemberBinding mb in todoTF) {
                string txt;
                if(todoLhs.ContainsKey(mb)) {
                    // this target-access is on the left
                    txt = getAssignAssertionTF(mb, todoLhs
                       [mb]);
                } else // this is on the right
                    txt = getReadAssertionTF(mb);

                Aux.addLineBefore(changes, n, txt);
            }

            todoTF.Clear();
            todoFF.Clear();
            todoLhs.Clear();
        }
        return n;
    }
```

The exact definition of the assertions are taken directly from the table on page 148 (under “Summary of normative conditions”). The plugin computes the assertions as a string. To keep the construction compact, I am using patterns in the string that are replaced by the method

```
string getAssertion(MemberBinding mb,
  AssignmentStatement ass, string format)
```

In the format string, `{x.t}` for instance is replaced by an access of the target field. If an assignment is present, `{e}` is replaced by the right-hand side, etc. The definition of the four methods that calculate the assertion statements just contain a call to `getAssertion` with the correct format string.

```
    private string getReadAssertionFF(MemberBinding mb) {
        return getAssertion(mb, null, "assert␣{x}␣==␣null␣
           ||␣{x.t}␣!=␣null;");
    }

    private string getAssignAssertionFF(MemberBinding mb,
       AssignmentStatement ass) {
        return getAssertion(mb, ass,
            "assert␣{x}␣==␣null"
              +"␣||␣({x.t}␣!=␣null␣&&␣"+
              "forall{{S}␣loc␣in␣enumof({S});"
              +"␣loc.{t}␣!=␣{x.t}});");
    }

    private string getReadAssertionTF(MemberBinding mb) {
        return getAssertion(mb, null,
            "assert␣true;");
    }

    private string getAssignAssertionTF(MemberBinding mb,
       AssignmentStatement ass) {
        return getAssertion(mb, ass,
            "assert␣{x}␣==␣null␣||␣("
            +"{x.t}␣==␣null␣?"
            +"␣{e}␣==␣null␣||␣{e}.{f}␣==␣{default_f}"
            +":␣{e}.{f}␣==␣{x.t.f}"
            +");");
    }
```

For the subset covered in this research, the code above works just perfectly. In C# and most other languages however, field accesses can also be used inside expressions. If the field is accessed inside the condition of a while loop, the assertion will only be checked once instead of before every evaluation of the expression. Problems also occur when right-hand sides are not side effect free.

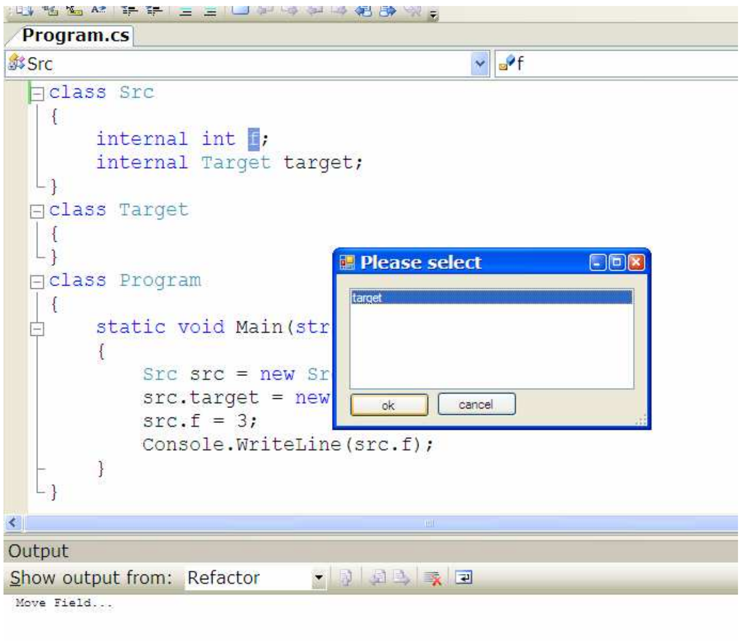
Appying CodeChanges After all the changes are stored in the `changes` list, the changes are applied. The changes are applied from back to front of each file so that the offsets do not have to be adjusted after each insertion:

```
    if(fileChanges.TryGetValue(fileNames[i], out fchanges)
       )
        fchanges.Sort(delegate(SourceChange a,
           SourceChange b) {
            return b.SourceContext.StartPos - a.
               SourceContext.StartPos;
        });

        StringBuilder sb = new StringBuilder(
          c.CompilationUnits[i].SourceContext.SourceText);

        for(int j = 0; j < fchanges.Count; j++) {
            SourceContext cx = fchanges[j].SourceContext;
            sb.Remove(cx.StartPos, cx.EndPos - cx.StartPos
               );
        sb.Insert(cx.StartPos, fchanges[j].ChangedText
           );
    }

    using(StreamWriter w = new StreamWriter(
       newFileNames[i])) {
        w.Write(sb.ToString());
    }
```



```
class Src
{
    internal int f;
    internal Target target;
}
class Target
{
}
class Program
{
    static void Main(str
    {
        Src src = new Sr
        src.target = new
        src.f = 3;
        Console.WriteLine(src.f);
    }
}
```

Output

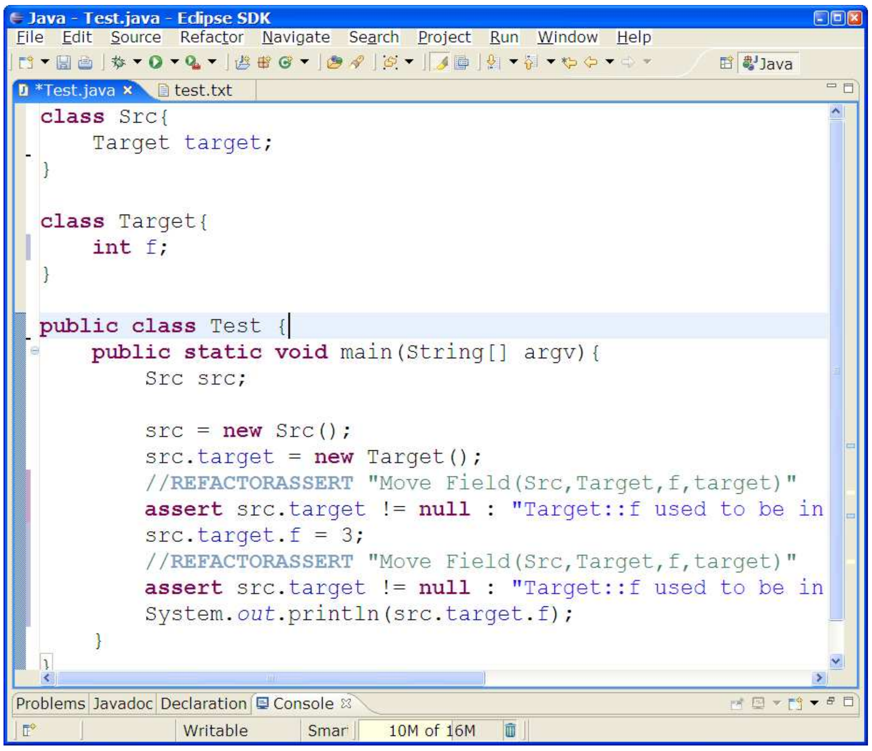
Show output from: Refactor

```
Move Field...
```

Figure 6.4.: The tools show a modal dialog to let the user select the field along which the field is to be moved.

6.2. Writing a plugin for Eclipse 3.2 and Eclipse's refactoring framework

The Eclipse IDE supports refactoring extremely well compared to other development environments. It also provides an extensive framework for source code reorganisation. It is this framework that I want to present in this section. Just as for Visual Studio, I implement a small Eclipse extension that can perform the "Move Field" refactoring – I present the refactoring with just one Java assert statement to safe some space. The source code that comes with this thesis implements the whole range of checks using JML specifications.


Java - Test.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Java
*Test.java
test.txt
Problems Javadoc Declaration Console
Writable
Smar
10M of 16M

```
class Src{
    Target target;
}

class Target{
    int f;
}

public class Test {
    public static void main(String[] argv){
        Src src;

        src = new Src();
        src.target = new Target();
        //REFACTORASSERT "Move Field(Src,Target,f,target)"
        assert src.target != null : "Target::f used to be in
        src.target.f = 3;
        //REFACTORASSERT "Move Field(Src,Target,f,target)"
        assert src.target != null : "Target::f used to be in
        System.out.println(src.target.f);
    }
}
```

Figure 6.5.: The result of executing the refactoring.

Extensions to Eclipse are always called plugins, no matter how they are actually built. In fact, everything except the basic infrastructure is a plugin in eclipse. The implementation is now referred to as a *bundle* since Eclipse 3.0 is trying to adhere to – and extending – the OSGi standard.

Eclipse plugins can be extended by implementing so-called extension points. The thing that implements an extension is called the extension and is itself inside a plugin. Unfortunately, there are no extension points for custom refactorings. It is thus impossible to make custom refactorings look like built-in ones without changing the internals of the JDT plugin.

The plugin presented here extends the Quick Assist extension point, i.e., it implements a Quick Assist. Quick Assists are tools that provide context sensitive assistence for the programmer normally by local transformations on the source code. The great thing about the Quick Assist extension point is that it is intended for exactly the purpose I am using it: Looking at the AST and determining whether some transformation is applicable. The extension point gives you direct access to the underlying compilation unit.

6.2.1. Quick Assist workflow

The class that implements the Quick Assist is `MvFieldQuickAssist` and has to implement `IQuickAssistProcessor`.

The class is required to implement two methods

```
boolean hasAssists(IInvocationContext context) throws
   CoreException;
IJavaCompletionProposal[] getAssists(
   IInvocationContext context, IProblemLocation[]
   locations) throws CoreException;
```

`hasAssists` is queried to indicate whether Quick Assistance is available in the given context. It corresponds to the `Query` method in Visual Studio.

`getAssists` returns a number of of `IJavaCompletionProposal` objects.

Here are the most important methods of the `IJavaCompletionProposal` interface:

```
String getDisplayString();
void apply(IDocument document);
```

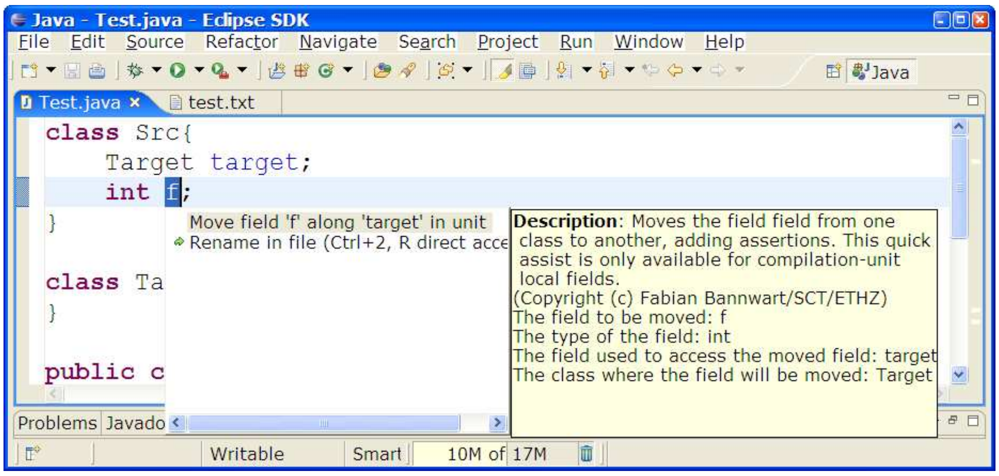
The `IJavaCompletionProposal` objects contain all the information to display and execute the Quick Assist. All the objects returned are displayed to the user, who can select one of them.

`getDisplayString()` returns the string that is displayed in the list-box that pops up when pressing Ctrl+1. `apply` effectuates the changes stored in this object of type `IJavaCompletionProposal`. This method is called when the user selects the proposal.

6.2.2. Code walkthrough

I didn't delve into the details of Visual Studio's CodeModel and I didn't discuss `Query` because it differs from what is actually used for the refactoring. In Eclipse however, there is only one code model, so I will take this chance and explain how `hasAssists` works.

`hasAssists` must be precise, that is return true only if there are actually proposals. The work that has to be done in this routine is consequently about the same what is done in `getAssists` that returns the assists for a given context.



Java - Test.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Java
Test.java
test.txt
Problems
Javado
Writable
Smart
10M of 17M

```
class Src{
    Target target;
    int f;
}

class Ta
}

public c
```

Move field 'f' along 'target' in unit
Rename in file (Ctrl+2, R direct acce

Description: Moves the field field from one class to another, adding assertions. This quick assist is only available for compilation-unit local fields.
(Copyright (c) Fabian Bannwart/SCT/ETHZ)
The field to be moved: f
The type of the field: int
The field used to access the moved field: target
The class where the field will be moved: Target

Figure 6.6.: Choosing the refactoring in the Quick Assist menu.

The assists are stored in an auxiliary class `Proposals`. `hasAssists` constructs the proposals and tests whether they are applicable, that is, whether there are any proposals.

```
public boolean hasAssists(IInvocationContext ct)
   throws CoreException {
    return new Proposals(ct, false).isApplicable();
}
```

The method `getAssists`, used to collect Quick Assists, also constructs the proposals and returns them. Eclipse has the idea that it passes compilation problems that are close to the invocation site as an additional parameter `locations`. They are are just ignored: as long as the proposals can be successfully constructed, it is ok if there are problems around.

```
public IJavaCompletionProposal[] getAssists(
   IInvocationContext context,
        IProblemLocation[] locations) throws
           CoreException {
    return (new Proposals(context, true)).getAll();
}
```

Let's now have a look at the `Proposals` class and its constructor that encapsulates the functionality for "Move Field".

The constructor takes two arguments: `IInvocationContext ct` and `boolean produce` `produce = false` means that the proposals do not actually have to be produced and the constructor aborts once the first proposal is found. `ct` just carries the invocation context that gets passes to `hasAssists` and `getAssists`.

The context provides methods to retrieve the AST for the compilation unit and – what is really handy – the node that is covered by the current selection.

```
cu = ct.getCompilationUnit();
ASTNode p = ct.getCoveredNode();
```

The Quick Assist assumes that the user selects the name of the field to be moved first and then presses Ctrl+1. If there is no selection, the Quick Assist does not have anything to do.

```
if(p == null) return;
```

The same is true if the user does not select a proper name: `Name` is the superclass of all nodes that refer to bindables, i.e., named entities, like procedure names, field names, qualified class names, etc.

```
if(!(p instanceof Name))
    return;
```

The name has to be resolved to a field declaration. This is done with `resolveBinding` that returns a "binding", i.e., the reference to a named entity in Java. If the binding cannot be resolved, `resolveBinding` returns **`null`**.

```
Name pn = (Name)p;
IBinding ffb = pn.resolveBinding();
if(ffb == null) return;
```

`ffb` is actually an `IVariableBinding` (if it really refers to a field). Other possibilities are `IPackageBinding`, `ITypeBinding`, `IMethodBinding`.

Most bindings have an underlying Java element that is represented with the interface `IJavaElement`. Only bindables that are not defined in the Java language itself (like the built-in types) do not have corresponding `IJavaElement`s. All other bindables do have a corresponding `IJavaElement` sub-interface, e.g., `IType`, `IMethod`, `IField`, `ITypeParameter`, etc.

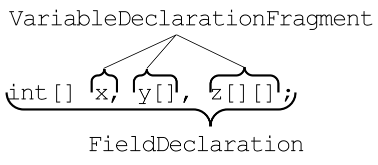
```
IJavaElement ffj = ffb.getJavaElement();
if(ffj == null) return;
if(ffj.getElementType() != ffj.FIELD)
```

```
        return;
```

It is already quite good to have an `IJavaElement` at hand. What is needed however is the corresponding node in the source code. I use the internal method `findDeclaration` here. Using internal APIs is not recommended as they may change without notice. I do so because there are no public methods to achieve the same in a similarly concise manner.

```
    ff = (VariableDeclarationFragment)ASTNodes
            .findDeclaration(ffb, p.getRoot());
    if(ff == null) return;
```

The method `findDeclaration` returns a `VariableDeclarationFragment` AST node. A `VariableDeclarationFragment` is part of a variable or field declaration (type `FieldDeclaration`).



The AST nodes in Eclipse are truly abstract syntax trees. The AST closely follows the structure of the source code and is thus formidably suited for specifying refactorings. This contrasts with the use of the tree used in .NET's CCI that emphasizes the tree as an intermediary representation. It is also referred to as such in the documentation. While I prefer the approach chosen in `System.Compiler`, some language specific hooks that *allow* restoring the concrete program representation – and frameworks that can do it – wouldn't hurt too much.[7]

The `FieldDeclaration` in turn is a child of the `TypeDeclaration`. Calling method `getParent` twice thus returns the class node if everything is parsed as it should be.

```
    FieldDeclaration ffdecl = (FieldDeclaration)ff.
       getParent();
    ASTNode clnode = ffdecl.getParent();
    if(clnode == null || clnode.getNodeType() != ASTNode.
       TYPE_DECLARATION)
        return;
    fc = (TypeDeclaration)clnode;
```

[7].NET does provide `SourceContexts` that allow to retrieve the underlying source code, but not the originating grammar productions.

I restrict the analysis to just one compilation unit and the Quick Assist only updates the current unit.[8] For a public type with a field that is public, protected or has default visibility, the transformation is in general not correct as some accesses to the field can be from outside the current unit.[9]

```
if((fc.getModifiers() & Modifier.PUBLIC) != 0
&& (ffdecl.getModifiers() & Modifier.PRIVATE) == 0)
    return;
```

Quick Assist will present the user with a list of fields along which the selected field is to be moved. There is only one way to do that: iterating through all the fields declared in `fc`. This requires two **for** loops because each field is inside a `FieldDeclaration` containing `VariableDeclarationFragment`s.

```
FieldDeclaration[] fdecl = fc.getFields();
for (int i = 0; i < fdecl.length; i++) ...
    List flds = fdecl[i].fragments();
    for (Iterator itf = flds.iterator(); itf.hasNext()
       ;) ...
        VariableDeclarationFragment tf =
            (VariableDeclarationFragment) itf.next();
```

What follows resembles what is done above for the source class `fc` and source field `ff`. The field declaration uses a type, which must not be an array type nor a primitive type.

```
Type tc_occurence = fdecl[i].getType();


if ((tc_occurence.isArrayType()) || (tc_occurence.
   isPrimitiveType()))
    continue;
```

It is a peculiarity of Java that array brackets `[]` can be put both after the type and after after the variable name in a declaration. For a fragment, they are called extra dimensions (`a[][]` has two extra dimensions)

```
if(tf.getExtraDimensions() > 0)
    continue;
```

[8]This restriction does not apply to the .NET version, which updates all files in the project.

[9]The test could be skipped if absolutely necessary as the transformation is likely (i.e., in the absence of shadowing) to produce compile-time errors.

The type declaration `tc` is obtained as above using the pair `resolveBinding` and `findDeclaration`. The proposals are stored in the `target` list. Every `Proposal` contains a reference to the target field `tf` and the target class `tc`.

```
if(target == null)
    target = new ArrayList<Proposal>(2);

Proposal prop = new
    Proposal(this, target.size(), tf, tc);

target.add(prop);
```

The proposals implement the `IJavaCompletionProposal` interface. An array of `IJavaCompletionProposal` objects is to be returned by `getAssists`. The method `getAll()` provides them from the `target` list.

```
public IJavaCompletionProposal[] getAll() {
    if(target == null) return null;
    return (IJavaCompletionProposal[])target.toArray(
       new IJavaCompletionProposal[target.size()]);
}
```

Applying the transformation As described above, every `Proposal` has an `apply` method that is called when the user chooses to execute the Quick Assist.

The AST is used to work with the transformation framework built into Eclipse. It allows to specify the transformations as tree updates on trees and delegate the responsibility to generate text-edits to the framework. As you'll see, this is much more elegant than the ad-hoc approach used for Visual Studio.

Transformations on source-code are encapsulated as `ASTRewrite` objects. `ASTRewrite` allows you to collect transformation information and then apply it to the AST at once. The transformation is recorded without modifying the original AST. What makes `ASTRewrite` so powerful is the fact that it preserves comments and formatting, and also respects code formatting settings by the user.

```
final ASTRewrite rewriter = ASTRewrite.create(ast);
```

Before applying the tranformation, I distinguish two steps: Moving the declaration and moving the accesses to the field.

```
moveDeclaration(ff, ast, rewriter);
adjustAccesses(targetField, ff,fc,rewriter, ast);
TextEdit ed = rewriter.rewriteAST(document, null);
```

Both `moveDeclaration` and `adjustAccesses` add transformations to `rewriter`, which is what I want to explain. `moveDeclaration` is the simpler one, so I desist from discussing `adjustAccesses`.

The declaration fragment `ff` has to be moved from the source class to the target class. It is not directly possible to unlink the fragment and record the insertion in `rewriter`. Instead, a pseudo-node ("placeholder") has to be created that represents `ff`. This placeholder can be used just as it were `ff` in `rewriter`

```
ASTNode newField = rewriter.createMoveTarget(ff);
```

Except for the field fragment, the new declaration is made from scratch.

```
FieldDeclaration newDecl = (FieldDeclaration)ast.
   createInstance(ASTNode.FIELD_DECLARATION);

List newFrag = newDecl.fragments();
newFrag.add(newField);
```

The newly created declaration has to be inserted. This done with a `ListRewrite` that is tightly integrated with `ASTRewrite`. Whenever there is a list of something, you have to create a `ListRewrite` object and you cannot use `ASTRewrite` directly. In this case, it is the list of declaration in the class body.

```
ListRewrite lrw = rewriter.getListRewrite(targetClass,
targetClass.BODY_DECLARATIONS_PROPERTY);
lrw.insertFirst(newDecl, null);
```

The old declaration of the field has to be removed. If there is only one fragment, the whole declaration has to be removed, otherwise, it is sufficient to remove the fragment. This is a weakness of `ASTRewrite`.

```
rewriter.remove(oldDecl.fragments().size() > 1 ? ff :
   oldDecl, null);
```

6.3. Concluding remarks

This chapter has shown that refactoring with specifications can be supported with relatively modest efforts. The conditions yielded by the proof process are directly applicable.

The effort is likely to increase when trying to make the refactorings correct for all valid Java and Spec# programs, not only those that comply with the subset discussed here. Small transformations that can be modularly applied are then necessary. Examples: Transformations to introduce temporary variables, to extract loops conditionals, take if statements apart, etc. Such facilities are likely to exist already somewhere in Eclipse and Visual Studio. It would greatly facilitate the vision of refactorings with specifications if they were made available for custom transformations.

Even if more supporting facilities are made available to the plugin developer, only the tool vendor can ultimately leverage the enormous potential of specification-supported refactorings. This chapter is an illustration that it is easily possible.

7. Conclusion

In the preceding chapters, I presented techniques for proving refactorings correct, for making difficult refactorings accessible to tool support, for extracting specifications from the application of refactorings at no cost, revealing knowledge about the program that would otherwise remain concealed. The key to this achievement is the first formalization of refactorings that has been applied to realistic language semantics and realistic transformations. The corresponding proof process yields correctness conditions for free. These conditions can then be used as specifications in the transformed program.

I summarize the main technical contributions of this thesis and discuss their relevance and limitations. I also discuss possible future research that is mandated by the insights this text provides.

7.1. Summary and contributions

Refactorings are program transformations that aim at retaining equivalence. The original program is transformed to the refactored program. The two programs are related: The code is related by the transformation function and data correspondence relates the state spaces of the programs. The correspondence of data is not a simple simulation between the transition systems defined by the programs: data correspondence may change depending on the positions of the program counters and it may not exist at all for some points. Moreover, a step in the original program can correspond to multiple steps in the refactored program and vice versa. This flexibility makes it handy to prove refactorings correct and to use the data correspondence that is intutively the right one. Data correspondence is required to be the identity on the state of issued I/O operations. The I/O space is constructed to make it impossible to invalidate correspondence and then restore it. Thus it is guaranteed that externally visible intermediary states are always the same for the original and the refactored program.

The correctness proof proceeds inductively, statement by statement. Conditions that are not necessarily fulfilled are correctness conditions of the refactoring. These are local conditions, i.e., they can be expressed as pre- and postconditions of the statement that is being examined. Local conditions can be easily added as specifications to the transformed program, for instance as assert statements.

Refactorings can be applied one after another. There are no special requirements if specifications are treated as part of the program: Specifications are just transformed together with the program.

7.1.1. Catalogue of simple refactorings

This text contains a systematic catalogue of refactorings. They are broken down to atomic refactorings that either have limited lexical scope or only need a limited number of changes. The catalogue is an illustration how other refactorings can be formalized, analyzed and tackled.

7.1.2. Refactoring responsibilities

The "Move" refactoring allows the programmer to move data from one object to another. The association between the two objects is given by a field in the simplest case. This simple case provides an interesting example and shows that the specifications produced by a refactoring may differ depending on the program's structure and the programmer's intent.

7.1.3. Refactoring as specification by interaction

Visual Studio has *Domain Specific Languages* and "Code Snippets", Eclipse and Net-Beans have *Quick Assists* and *Quick Fixes*, JBuilder has Wizards, SharpDevelop has "Auto code generation". All of them support model driven development in some way or another. Coding can largely be done by applying the right tools in the right order instead of inserting individual characters into a source code file. This *should* be done to a far greater extent. This is a great opportunity for specifications. Programmers can hardly be expected to manually insert specifications into their source if they do not express functional properties. Instead, most specifications should be inserted by code transformers.

This is more powerful than static inference because it can reveal what is not apparent and more powerful than runtime inference because it illustrates what is not tested yet imposed by the programmer.

7.2. Future work

Depending on the programmer's intent, refactorings may have different correctness conditions and different data correspondences. The scope and importance of refactorings can only be understood if these relationships are systematically analyzed. To make this possible, some experience must be gained with complex tool-supported refactorings that benefit from specifications. Refactoring tools that master such refactorings do not exist to date. I have shown how an implemenation could look like in chapter 6. It is not necessary to wait for the widespread adoption of languages that support specifications like Spec# or JML. Normal Java assert statements are quite sufficient. It is more important however that these tools make available features that render "programming by specification" feasible. They include: Reverse refactoring for every forward refactoring, association between code editing steps and specifications that are added, history of code transformations and possibilities to roll them back selectively (sometimes called "non-destructive/selective undo"). The process will increase the number of refactoring(-variant)s that should be implemented. Frameworks that support building and composing refactorings are desparately needed. A notation like the one used throughout this thesis can serve as a basis, but it has limitations (illustrated at the end of chapter 4) that should be avoided.

A. Code For Program Representation

The code listed here corresponds to the mathematical definition in chapter 3.

```
class Name{} /* Opaque type */

class Prog
    : Dictionary<Name,Decl>{}

class Fields
    : Dictionary<Name,TypeTag>{}

class Methods
    : Dictionary<MethodSignature, MethodDecl>{}

class Decl{
    Name superclass;
    List<Name> superifaces;
    Fields fields;
    Methods methods;
    StaticInitializer staticinitializer;
}

class StaticInitializer : Statement { }

class MethodSignature{
    Name mname;
    List<TypeTag> paramTs;
}

class MethodDecl{
    List<Param> prms;
    TypeTag retT;
    MethodBody body;
}

class MethodBody{}
```

```
class ExternalMethodBody
    : MethodBody{}
class NoMethodBody
    : MethodBody{}
class ImplementedMethodBody
    : Statement{}

class TypeTag{}

class SimpleTypeTag : TypeTag{
    Name className;
}

class ArrayTypeTag : TypeTag{
    TypeTag baseType;
}

class Expr{
    enum Kind {
        ADD, MUL, ...
        LDFLD, WRFLD, INVKVIR,
        INVKSPC, ...
    }
    Kind kind;
    Expr[] nodes;
}

class Statement : Expr { }

class Param{
    Name pname;
    TypeTag paramT;
}
```

B. Notations

See also section 3.1.2 for a description of $\Gamma[\cdots]$

$(T)\mathit{option}$	is not an algebraic datatype. Instead, $(T)\mathit{option} = T + \{\mathit{undef}\}$. This means that the partial function definition $A \hookrightarrow B$ is equivalent to $A \to (B)\mathit{option}$, that the constants None and *undef* are equivalent, that $\mathrm{Some}\, x = x$. I use the constructor $\mathrm{Some}\, x$ to denote elements x of $(T)\mathit{option}$ for which $x \neq$ None. This notation is used for the operational semantics to make the rule specifications more concise.
$\mathrm{ctt}(x)$	The function returns the declared type (compile time type) of variable (or field) x. This type can be thought of as being encoded in the name itself.
$\mathrm{rtt}(x)$	returns the runtime-type (i.e., the first element in a heap value) of a value in the object heap. A close look at the operational semantics reveals that $\mathrm{rtt}(x)$ is invariant for the lifetime of the program for any given location and the heap, i.e., $\Gamma \vdash s \xrightarrow{t} s' \wedge \mathrm{rtt}(s.\,\gamma(p)) = T \Rightarrow \mathrm{rtt}(s'.\,\gamma(p)) = T$
β	is the function formulation of β, i.e., $\beta = \{x \mapsto y \mid (x, y) \in \beta\}$ it is well defined only if β is actually a function, i.e., $\beta(x, y_1) \wedge \beta(x, y_2) \Rightarrow y_1 = y_2$
init_obj	returns a function that maps all field values of the specified type to the default value for this type (0 and false for primitive types, Null for references).
$X \lhd F$	Restricts the function F to domain X such that $(X \lhd F)(y)$ is not defined for $y \notin X$.
$x \lhd\!\!\!- F$	Subtracts x from the domainn of F, i.e., $x \lhd\!\!\!- F = (\mathrm{dom}\, F - \{x\}) \lhd F$.
$\{T\ x;\ S\}$	introduces block local variables. It is called the "block statement". Its definition is $$\frac{\Gamma \vdash (\mathrm{xcpt}, \sigma, \gamma) \xrightarrow{S} (\mathrm{xcpt}', \sigma', \gamma')}{\Gamma \vdash (\mathrm{xcpt}, \sigma, \gamma) \xrightarrow{\{T\ x;\ S\}} (\mathrm{xcpt}', \sigma'[x \mapsto \mathit{undef}], \gamma')}$$ Refactoring equivalence for $\{T\ x;\ S\}$ does not have to be considered separately: There are no refactorings for which the block statement would pose any difficulties as it is *formally* equivalent to $S; x \leftarrow \mathit{undef}$ and reasoning is always over individual statements.

$(x : T, \ldots)$	For the mathematical definition of the program representation in chapter 3, I rely on named tuples. The set $Z \equiv (a : X, b : Y)$ is equivalent to the set $Z \equiv X \times Y$ where $a(t) = t.a = \mathrm{fst}(t)$ and $b(t) = t.b = \mathrm{snd}(t)$ if $t \in Z$ where $\mathrm{fst}(x, y) = x$ and $\mathrm{snd}(x, y) = y$ are the usual extraction functions for tuples. Tuples with more than two components are defined similarly. This is the same definition that is sometimes used in relational algebra. A named tuple can also be interpreted as a partial function that returns the value of each component for the name of the component. In the example above, the tuple (x, y) corresponds to the partial function $\{a \mapsto x, b \mapsto y\}$. "$a$" and "$b$" are called accessors in this context.[1]

[1]Programmers should find this interpretation of tuples as partial functions natural because it closely resembles object representation in the operational semantics. The view also corresponds directly to the concept of objects in prototype-oriented languages like Self or ECMAScript that have the dictionary directly attached to them. For an illustration, have a look at the following constructor function `Z` in JScript.NET.

```
function Z(a : X, b : Y){
    this.a = a; this.b = b;
}

var z = new Z(x,y);

print("z.a␣=␣", z.a, ";␣z['a']␣=␣", z["a"]);
print("z.b␣=␣", z.b, ";␣z['b']␣=␣", z["b"]);
```

`Z` returns a tuple. The tuple is named and corresponds to the following class definition.

```
class Z{
  var a : X, b : Y;
}
```

Instead of the usual dot (`z.a`) notation, the square bracket notation `z["a"]` can also be used to access the component. One possibility is to interpret `z[?]` as a partial functions from names to component values.

C. About the Operational Semantics

To the end of establishing local criteria for equivalence during refactoring, I am using an operational semantics. The advantages of operational semantics over other formalizations is well-accepted: the formalism requires only minimal mathematical background, it is easy to derive executable models from the semantics, it is easier to understand than other approaches.

Oheimb's operational semantics This appendix gives a detailed account of this semantics presented in table 3.3 and table 3.4. I originally envisioned using the operational semantics presented in [44]. I found that a simplified variant that is tailored and sufficient for the goal I am persuing here is more appropriate as the formalism in chapter 3 aims at being largely independent of the actual semantics – any operational semantics with a program state similar to the one used here. As I said before, this is intentional: operational semantics are the touchstones of novel theories, not vice versa.

There is another reason not to use David's formalization directly: He had to use the theorem prover Isabelle but I do not. Partial functions are used directly, instead of "$(\alpha, \beta)table$". Interfaces and classes are unified and the notations are much less heavy.

The operational semantics is also *simpler*: side-effects in expressions are dropped for instance. This leads to more rules that are easier to understand. There are separate rules for array access, array update, object access, object update, etc. Many of the rules are very similar. I prefer having multiple similar rules than just one rule that encodes different behaviours (like method-lookup).

At the same time, the text tries to use additional simplifications that do not change the expressiveness of the underlying language: It eliminates different modi when invoking methods and replaces them by two different commands for invoking virtual functions and for invoking non-virtual functions.

Static methods are translated to methods of the corresponding classes (meta-class objects in Smalltalk). This is a unification that is not explicitly recognized in [44]. It does, however, recognize that there is some more room for unification in his model (p. 39, bottom, "One could adopt the Smalltalk view that everything is an object") but it is not clear how useful the proposed unification would be for checking program properties.

Omissions Type safety and well-formedness is not formalized even though I will have to assume certain invariants from time to time. Because the semantics used here is still an adaption of the operational model presented in [44], the text should be easily transfereable to the original semantics with some effort. Moreover, the semantics retains all the properties I am investigating as part of this work. This includes, most importantly, heap, class, method, parameter structure, local variables, fields, etc. Note that properties that are to be ensured by the type system are not subject to investigation. It could be established separately by proving that the transformations result in well-formed programs. Together with a type-safety theorem, this would guarantee such properties.

Capabilities The semantics I am using still has to be reasonably sophisticated because refactoring may only become interesting if relatively complex conditions are taken into account like exceptions, class initialization errors, etc. It would certainly be worthwhile to use an even more detailed specifications [41] if it were more easily accessible. [44] is probably the most appropriate choice as a basis for such a semantics as (i) it supports the features needed and it (ii) is a big-step semantics, which might simplify reasoning. Other semantics like [38] for instance lack features I'd like to have (dynamic class loading and exception handling). Most[1] big-step semantics that have these features however necessarily seem pretty similar and it may not matter too much what to base this work on.

In our formalism, just as in [44], the state consists of the currently active exception (xcpt), the local variables and parameters (σ) and the globals (γ, it includes the heap and class objects). It also includes the sequence of I/O operations $\mathfrak{u} = (a_1^{I/O}, \cdots, a_{|\mathfrak{u}|}^{I/O})$ and the "decision procedure" for inputs $\mathfrak{u}_0$, collectively written as $\mathfrak{u}$ because $\mathfrak{u}_0$ is supposed to be exogenously given and identical for all runs of all programs.

$$s \equiv (\mathrm{xcpt}, \sigma, \gamma, \mathfrak{u})$$

Note that [44] calls the "full state" σ and refers to the locals and globals together as s.

The program is called Γ and the big-step transition relation has a conventional (and unified) format for a statement that is identified by a path t in Γ (statement t):

$$\Gamma \vdash (\mathrm{xcpt}, \sigma, \gamma, \mathfrak{u}) \xrightarrow{t} (\mathrm{xcpt}', \sigma', \gamma', \mathfrak{u}')$$

Unlike [44], no separate expression is used that is evaluated to return the result but assume that a procedure's desired result value is stored in a special variable `result` just as in [38].

[1]All I've seen

The rest of this chapter discusses some additional differences that are related to individual statements.

Exception propagation The rule for exception propagation is simplified. xcpt = $(\mathit{ObjectLocation})\mathit{option}$ just as in [44]. Null is not used instead of None to signal the absence of an exception – unlike in [38]. The only reason is to notationally simplify case distinctions on whether an exception is active or not.

Object allocation Object allocation is the same as in [44]. It is not our primary goal to retain equivalence with respect to allocators. The heap is a function from locations to values.

The heap is not axiomatically modelled. rtt is used to extract the type tag from a heap value. ctt is the declared type of a variable or field. For heap values v, $v(f) = (\mathit{right}(v))(f)$. The full definition of object allocation involves some technicalities like setting the fields of the object to default values. These are covered in [44, p. 40] and can be accepted for our purpose. We use the function init_obj, assumed to return a value instead of an updated state. Object allocation does not cause `OutOfMemory` errors. They are rare in practice and make it difficult to reason about refactorings that increase the heap size. The criteria in this text are correct modulo out-of-memory errors as legitimated in chapter 1.

Raising exceptions In [44], an exception can either be a heap allocated non-null location or a standard exception that is merely identified by its name (p. 42). It seems more sensible to erase this distinction.

Bibliography

[1] A. W. Appel and J. Palsberg. *Modern Compiler Implementation in Java.* Cambridge University Press, New York, NY, USA, 2003.

[2] F. Bannwart and P. Müller. A logic for bytecode. In *Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2005.

[3] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS*, volume 3362 of *LNCS*. Springer-Verlag, 2004.

[4] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change.* Addison-Wesley Professional, 2nd edition, 2004.

[5] P. L. Bergstein. Object-preserving class transformations. In *OOPSLA '91: Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 299–313, New York, NY, USA, 1991. ACM Press.

[6] E. Börger, G. Fruja, V. Gervasi, and R. F. Stärk. A high-level modular definition of the semantics of C#. *Theoretical Computer Science*, 2004. Accepted for publication.

[7] D. Box and A. Hejlsberg. LINQ project overview. Available from `http://msdn.microsoft.com/netframework/future/linq/default.aspx`.

[8] E. Casais. An incremental class reorganization approach. In *ECOOP '92: Proceedings of the European Conference on Object-Oriented Programming*, pages 114–132, London, UK, 1992. Springer-Verlag.

[9] M. Cinneide and P. Nixon. Composite refactorings for Java programs, 2000.

[10] M. O Cinneide. *Automated Application of Design Patterns: A Refactoring Approach.* PhD thesis, University of Dublin, Trinity College, 2001.

[11] M. Cornélio. *Refactorings as Formal Refinements.* PhD thesis, Universidade de Pernambuco, 2004.

[12] R. DeLine and M. Fähndrich. Typestates for objects. In *ECOOP*, pages 465–490, 2004.

[13] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object Oriented Reengineering Patterns.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002. Foreword By-Ralph E. Johnson.

[14] *Standard ECMA-262: ECMAScript Language Specification.* ECMA International, 3rd edition, 1999.

[15] M. Feathers. *Working Effectively with Legacy Code.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[16] M. Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[17] E. Gamma. Private conversation, 2005.

[18] R. Gheyi, T. Massoni, and P. Borba. An abstract equivalence notion for object models. *Electr. Notes Theor. Comput. Sci.*, 130:3–21, 2005.

[19] J. Gutknecht. Component-oriented virtual machines. Lecture script, 2003.

[20] J. Gutknecht. Private conversation, 2005.

[21] P. H. Hartel and L. Moreau. Formalizing the safety of Java, the Java Virtual Machine, and Java Card. *ACM Computing Surveys*, 33(4):517–558, 2001.

[22] J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt. The Geneva convention on the treatment of object aliasing. *SIGPLAN OOPS Mess.*, 3(2):11–16, 1992.

[23] G. Hunt, J. Larus, M Abadi, M. Aiken, P. Barham, M. Fähndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft Research, 2005. Available from `http://research.microsoft.com/os/singularity`.

[24] About JMLEclipse. `http://jmleclipse.projects.cis.ksu.edu/`.

[25] J. Kerievsky. *Refactoring to Patterns.* Addison-Wesley Professional, August 2004.

[26] R. Lämmel. Towards Generic Refactoring. In *Proc. of Third ACM SIGPLAN Workshop on Rule-Based Programming RULE'02*, Pittsburgh, USA, October5 2002. ACM Press. 14 pages.

[27] K. J. Lieberherr. Controlling the complexity of software designs. Available from `http://www.ccs.neu.edu/research/demeter/talks/eth-ibm-04/`.

[28] K. J. Lieberherr, W. L. Hürsch, and C. Xiao. Object-extending class transformations. *Formal Aspects of Computing*, (6):391–416, 1994. Also available as Technical Report NU-CCS-91-8, Northeastern University.

[29] T. Mens, N. Van Eetvelde, S. Demeyer, and D. Janssens. Formalizing refactorings with graph transformations. *J. Softw. Maint. Evol.*, 17(4):247–276, 2005.

[30] Object Mentor. Test driven development.

[31] Microsoft. C# 3.0 overview. Available from `http://msdn.microsoft.com/`.

[32] Microsoft. The spectrum of Visual Studio automation. Available from http://msdn.microsoft.com/.

[33] C. Morgan. *Programming from specifications.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[34] S. S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[35] M. C. Norrie, A. Würgler, A. Palinginis, K. von Gunten, and M. Grossniklaus. OMS Pro 2.0 introductory tutorial.

[36] M. Odersky and M. Zenger. Scalable component abstractions. In *OOPSLA 2005.*

[37] W. F. Opdyke. *Refactoring object-oriented frameworks.* PhD thesis, Champaign, IL, USA, 1992.

[38] A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *European Symposium on Programming (ESOP '99)*, volume 1576, pages 162–176. Springer-Verlag, 1999.

[39] D. B. Roberts. *Practical analysis for refactoring.* PhD thesis, 1999. Adviser-Ralph Johnson.

[40] R. F. Stärk. Formal specification and verification of the C# thread model. *Theoretical Computer Science*, 2005.

[41] R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine—Definition, Verification, Validation.* Springer-Verlag, 2001.

[42] R. F. Stärk, E. Börger, and J. Schmid. *Java and the Java Virtual Machine: Definition, Verification, Validation with Cdrom.* Springer-Verlag New York, Inc., 2001.

[43] L. Tokuda and D. Batory. Evolving object-oriented designs with refactorings. In *ASE '99: Proceedings of the 14th IEEE International Conference on Automated Software Engineering*, page 174, Washington, DC, USA, 1999. IEEE Computer Society.

[44] D. von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic.* PhD thesis, Technische Universität München, 2001.

[45] D. A. Watt. *Programming Language Design Concepts.* John Wiley & Sons, 2004.

[46] J. Whaley and M. Rinard. Compositional pointer and escape analysis for java programs. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 187–206, New York, NY, USA, 1999. ACM Press.

[47] P. Wu and K. Lieberherr. Shadow programming: Reasoning about programs using lexical join point information. In *Proceedings of the 4th International Conference on Generative Programming and Component Engineering*, 2005.

All links have been verified on February 21, 2006.