Refactorings as Formal Refinements

por

Márcio Lopes Cornélio

Recife, junho 2004
Refactorings as Formal Refinements

Márcio Lopes Cornélio

Esta tese foi apresentada à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Doutor em Ciência da Computação.

A thesis presented to the Federal University of Pernambuco in partial fulfillment of the requirements for the degree of Doctor (Dr.) in Computer Science.

Orientador (Supervisor):
Profa. Dr. Ana Lúcia Caneca Cavalcanti

Co-orientador (Co-supervisor):
Prof. Dr. Augusto César Alves Sampaio

Recife, junho 2004
Cornélio, Márcio Lopes


Inclui bibliografia e apêndices.


004.432.4 CDU(2.ed.) UFPE
005.117 CDD(21.ed.) BC2004-481
Acknowledgments

I would like to thank my supervisor, Ana Cavalcanti, for her teaching, advice and guidance along these years, and for providing constant direction. She has been a source of encouragement and inspiration.

I am also indebted to my co-supervisor, Augusto Sampaio, for his guidance, valuable discussions and suggestions, and for his refined perception. He has also been a source of inspiration.

I am thankful to my thesis committee that has contributed with comments, and corrections which helped me to improve this thesis. Special thanks to Paulo Borba for valuable suggestions and insightful comments.

Many thanks to Leila Silva for voluntarily and patiently proof-reading parts of this thesis, and for discussions about this work. Several improvements of it are due to her comments. Thanks to Alexandre Mota and Luís Carlos Menezes for useful discussions on program transformation during the development of this work.

I thank my colleagues of the Department of Computing Systems of the State University of Pernambuco for encouragement during the final stage of this work.

Thanks to the Centre of Informatics staff for their support in difficult moments. Thanks to José Roberto Pereira, Carlos Melo, Rodrigo Santana, and Mário Sérgio. A special thanks to Marlice Novais, for her friendship and support.

My parents, my sister and my brothers have provided continued support and stimulus, which can never be fully acknowledged. I cannot thank them enough, but I will start dedicating this work to them.

I am deeply grateful to Roxana de Siqueira for her encouragement, patience, and love along these years.

I am most fortunate to have so many friends so that it is difficult to name them all in this space. Their friendship, help and encouragement in all worrisome moments have been a distinguished asset.

Most importantly, I thank God for hidden care and for my salvation. Fundamentally, nothing else really matters. Glory and praise unto the Lord.

The work described in this thesis was financially supported by the Brazilian government through Capes, one of the Brazilian agencies that afford for postgraduate studies in Brazil.
Resumo

A reestruturação de programas no contexto da orientação a objeto é também conhecida como refactoring e consiste em mudanças na estrutura interna de um software, sem modificar seu comportamento externo, a fim de melhorar sua legibilidade e torná-lo mais fácil de passar por futuras mudanças. Na prática, refactoring baseia-se em compilação e testes para assegurar a preservação do comportamento.

Trabalhos como os de Opdyke e Roberts foram realizados com vistas à formalização de refactorings por meio da identificação de condições que devem ser satisfeitas para assegurar que uma mudança num programa preserva o comportamento do mesmo. As condições, geralmente escritas na linguagem do cálculo de predicados, são introduzidas como pré e pós-condições dos refactorings. Outras abordagens para a prova de preservação do comportamento de refactorings usam formalismos como análise conceptual e reescrita de grafos. Contudo, não há técnica algébrica que apresente refactorings como transformações que preservam o comportamento, com prova deste fato.

Nossa principal contribuição constitui-se na apresentação de refactorings como transformações de programas escritos em ROOL (Refinement Object-oriented Language), uma linguagem baseada em Java, com classes, controle de visibilidade, ligação dinâmica, e recursão. A linguagem ROOL permite que raciocinemos sobre programas orientados a objetos e especificações, pois a mesma une estas construções como no cálculo de refinamentos de Morgan. A semântica de ROOL é baseada em weakest preconditions. Um conjunto de leis de programação está disponível tanto para os comandos imperativos de ROOL quanto para construtores relacionados à orientação a objetos. A prova, na semântica de ROOL, de que tais leis são corretas, é também uma contribuição do presente trabalho.

Apresentamos refactorings como regras algébricas de refinamento envolvendo programas. A prova da preservação do comportamento é realizada pela aplicação de leis de programação a um lado da regra a fim de obtermos o lado oposto. Nós generalizamos a técnica padrão de refinamento de dados a fim de lidar com hierarquia de classes.

Neste trabalho também apresentamos como obter um sistema estruturado segundo um padrão de projeto, por meio da aplicação de regras de refactoring. Padrões de projeto constituem-se num objetivo natural para a realização de transformações por meio da aplicação de refactorings. Trabalhos presentes na literatura sobre padrões de projeto que propõem a formalização dos mesmos, em geral, concentram-se em suas descrições formais, não na transformação de um sistema com vistas a estruturá-lo de acordo com padrões de projeto. Também apresentamos a transformação de uma aplicação monolítica para uma aplicação estruturada segundo um padrão arquitetural.
Abstract

Program restructuring in the context of object-oriented programming is known as refactoring. This consists of changes made to the internal structure of software in order to improve its legibility and make it easier to modify without changing its external behaviour. In practice, refactoring usually relies on compilation and tests in order to guarantee behaviour preservation.

Works like those by Opdyke and Roberts have already been done in the direction of refactoring formalisation by means of the identification of conditions that must be satisfied to guarantee that a change to a program is behaviour preserving. The conditions, which are usually written in the predicate calculus, are introduced as pre- and postconditions of the refactorings. Other approaches for the proof of refactoring behaviour preservation use formalisms such as concept analysis and graph rewriting. However, there is no algebraic technique that presents refactorings as behaviour preserving transformations, with proofs carried out. This avoids changes of notation and facilitates mechanisation.

Our contribution is to present refactorings as transformations of programs written in the language ROOL (Refinement Object-Oriented Language), which is a Java-like imperative language with classes, visibility control for attributes, dynamic binding, and recursion. It allows reasoning about object-oriented programs and specifications, as both kinds of constructs are mixed as in Morgan’s refinement calculus. The semantics of ROOL, as usual for refinement calculi, is based on weakest preconditions. A set of programming laws is available for the imperative constructs of ROOL as well as for its object-oriented features. The correctness of these laws, which is also a contribution of the present work, is proved against the semantics of ROOL.

We present refactorings as algebraic refinement rules involving program terms. The proof that these rules are behaviour preserving is accomplished by the application of the programming laws of one of the sides of the rule to obtain the other side. The proofs of some refactoring rules also involve data refinement of classes. We generalise the standard data refinement technique from single modules (classes) to class hierarchies.

Design patterns arise as a natural objective for refactoring a system. The literature on design patterns already presents works that propose the formalisation of design patterns. They usually concentrate on the formal description of patterns, not on the transformation of a system with the intention of obtaining a final system structure according to a design pattern. In this work, we also present how to obtain a system that is in accordance with design patterns by the application of refactoring rules proved to be behaviour preserving. We also present the transformation of a monolithic application to a well-structured one according to an architectural pattern.
Contents

1 Introduction
  1.1 Refactoring and Patterns ........................................ 1
  1.2 Formal Methods .................................................... 3
  1.3 Refinement Calculi and Object-Orientation ....................... 4
  1.4 Objectives .......................................................... 6
  1.5 Thesis Overview ................................................... 8

2 Refactoring—State of the Art .......................................... 9
  2.1 Program Restructuring .............................................. 9
  2.2 Refactoring .......................................................... 10
    2.2.1 Formalisms ..................................................... 13
    2.2.2 Languages ...................................................... 15
    2.2.3 Refactoring Models ............................................ 15
  2.3 Design Patterns .................................................... 16
    2.3.1 Formalization of Design Patterns ........................... 17
  2.4 Conclusions .......................................................... 18

3 ROOL and Laws .......................................................... 19
  3.1 Syntax ............................................................... 19
  3.2 Typing ............................................................... 23
    3.2.1 Typing environment ......................................... 23
    3.2.2 Typing Rules .................................................. 24
  3.3 Semantics ............................................................ 27
    3.3.1 Environments and states ..................................... 27
    3.3.2 Extended typing ............................................... 28
    3.3.3 Commands and parameterised commands ..................... 29
    3.3.4 Programs and method calls ................................ 31
  3.4 Refinement .......................................................... 32
  3.5 Laws ................................................................. 33
    3.5.1 Simulation Laws ................................................. 33
## CONTENTS

- G.1.4 Parameter type ........................................... 278
- G.1.5 Method calls ........................................... 280
- G.1.6 Casts .................................................... 282
- G.1.7 Commands and expressions .......................... 283
- G.2 Further object-oriented programming laws ............ 286
  - G.2.1 Changing a superclass ................................ 289
  - G.2.2 Class invariant ...................................... 290

### H Typing Rules

293
List of Figures

1.1 Class Person before refactoring ............................................. 2
1.2 Class Person after refactoring .............................................. 3
1.3 Formalisation of Refactorings ................................................. 7
3.1 A class in ROOL ................................................................. 23
6.1 The system before refactoring ................................................... 134
6.2 The four-layer architecture ..................................................... 137
6.3 The class Application in the end of Stage 1 ................................. 141
6.4 The class Application in the end of Stage 2 ................................. 142
6.5 The class BusinessCollection in the end of Stage 2 ....................... 143
6.6 Class BusinessCollection in the end of Stage 3 ......................... 144
6.7 Classes RepositoryClass and RepositoryClassRef ....................... 144
A.1 Law (inline delegate) ............................................................ 156
A.2 Law ((undo)inline delegate) .................................................... 158
LIST OF FIGURES
List of Tables

3.1 Grammar for expressions and predicates ................................. 20
3.2 Grammar for commands and parameterised commands .................. 21
3.3 Programs in ROOL ......................................................... 22
3.4 Typing of Expressions ....................................................... 25
3.5 Typing of Parameterised Commands ........................................ 25
3.6 Typing of Commands ......................................................... 25
3.7 Syntactic Transformations .................................................... 29
3.8 Semantics of some commands ................................................ 30

H.1 Typing of Expressions ......................................................... 293
H.2 Typing of Predicates .......................................................... 294
H.3 Coercion Rules ................................................................. 294
H.4 Typing of Parameterised Commands ........................................ 294
H.5 Typing of Commands .......................................................... 295
Chapter 1

Introduction

Changes are intrinsic to software. After a software product is delivered to a customer, new requirements often arise. Also, the use of a software product may reveal mistakes that were not realised during development [48]. Sometimes, however, changes to a software do not affect its external behaviour. For instance, changing the name of a function, or eliminating duplicate code through the use of procedures, should not affect a software’s external behaviour. Indeed, such modifications change just the internal software structure. This activity is called software restructuring [45].

Many practitioners recognise that changing an object-oriented software is easier than conventional software [62]. Some changes to object-oriented software can be made just by the addition of new classes, subclasses, or methods in a class. However, changing an object-oriented software may require changing abstractions described by means of classes and their relationships like, for instance, moving an attribute or a method between classes.

When a structural change is made to a class or a set of classes, changes may also be needed elsewhere in a program in order to preserve its behaviour. For example, changing a method name implies changing the old name of the method to the new one in all calls that occur in the program. Other changes may affect inheritance hierarchies. In object-oriented programming, behaviour preserving transformations are known as refactorings [69, 42].

We propose an approach to refactoring that is based on transformation rules between programs written in an object-oriented language whose semantics has been formally described. Each rule is described by means of a meta-program on its left-hand side and another meta-program on the right-hand side and allows us to transform a program into another. The correctness proof of these rules is based on the application of laws of programming whose soundness is proved against the language’s semantics. Programming laws state properties of program constructs [46].

1.1 Refactoring and Patterns

Due to the complexity of object-oriented applications, changes must be done in a disciplined way, so that the behaviour of the program is preserved. Indeed, refactoring is defined as the “process of
changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure” [42].

As an example of a possible need for code design improvement, and also for disciplined change of program, let us consider the class Person in Figure 1.1 which embodies two different real-world concepts: person and telephone. It has an attribute name, which records the name of a person, and attributes areaCode and homeTelNumber, which record an area code and a home telephone number, respectively. We use the Java [1] programming language notation to describe this class.

Clearly there are two independent abstractions inside this class. In order to obtain a better design, it is necessary to split the class Person, so that the concept of telephone is described in a separate class. So, we should extract a reusable component: the class that describes telephones. This extraction requires refactoring the existing class Person. The classes that result from refactoring are presented in Figure 1.2. The class Person now is a client of class TelephoneNumber. Person has an attribute of type TelephoneNumber. This attribute is used as a target of calls to methods of TelephoneNumber.

As illustrated in our example, sometimes it is necessary to restructure a class in order to achieve reuse. One reason for this is that it is difficult to determine, in an initial design, all the important concepts for an application and how they interrelate. Indeed, in the example we presented, two concepts are described in a single class. After refactoring, the opportunities for reuse increase as the different classes can determine, for instance, new class hierarchies.

The changes in the example we presented are simple and can be done by hand. In fact, programmers have been doing changes as this one for years. Nonetheless, simple transformations can be part of a sequence of other transformations; and it is required that, afterwards, the system must behave as before any transformation. If the small refactorings are shown to be correct, then large changes composed of small refactorings will also be correct.

The practical approach to refactoring usually relies on program compilation and test cycles [42]. Compilation detects, for example, that a new class has the same name as an already existing class.
1.2. FORMAL METHODS

Formal methods are mathematical techniques for system specification, verification, and reasoning. Systems are specified using formal specification languages, which have well-founded mathematical basis. They include theories like first-order logic, sets, and algebra, which allow the verification of specification properties. The use of such languages reduces ambiguity, inconsistency, and incompleteness, which often arise when using informal development techniques.

Two approaches can be taken in a formal development process: one based on specification verification, and the other one based on specification transformation. In the first approach, for a given specification, a design or program is proposed and verified to satisfy the specification. In the second one, a specification is refined until a concrete design or a program is obtained. The transformational approach can be more effective than the verification-based one, because it seems generally less difficult to develop a program and verify its correctness at the same time than to verify a program against its specification in a retrospective way [34].

The formal development process is not necessarily intrinsic to the specification language. A
system can be developed using the verification-based or using the transformational approach, or even mixing these two approaches in spite of the language used. However, the semantic model of a language must give support to both approaches. One of the most well-known transformational techniques are the refinement calculi \[65, 5, 7, 66\], which involve a wide spectrum language and a set of correctness preserving rules. By using these rules, we can calculate a refined program that is correct by construction with respect to its specification.

The language Z \[79, 78\] is an example of a formal specification language that can be used in a development process that mixes the verification-based approach as well as the transformational one. Traditionally, a Z abstract specification is proposed for a system, then a concrete specification is presented and verified to be correct against the abstract specification. The correctness is based on refinement proof obligations \[89\]. This phase of the development process is based on specification verification. The concrete specification of the system, however, is not executable. It is necessary to transform this concrete specification in order to obtain program code. This transformation can be done by means of a Z refinement calculus \[20\], which presents a set of conversion and refinement laws for Z.

Extensions to formal specification languages, like Z, to deal with object-oriented features were proposed mainly in the beginning of the 1990's. Among the Z extensions we can find Object-Z \[72\], and MooZ (Modular object-oriented Z) \[58, 59\]. These extensions are used for the specification of object-oriented systems, but there is still a gap between the most concrete specification we can obtain at the end of the development and program code. The development process usually applied in the context of object-oriented formal specification languages is based on specification verification. There is no refinement calculus defined for these languages.

1.3 Refinement Calculi and Object-Orientation

Refinement calculi have been extensively used as a formal basis for stepwise development in the context of imperative programming languages. Different approaches were proposed by Back \[5, 7\], Morgan \[65, 64\], and Morris \[66\]. Their languages are extensions of the language of guarded commands of Dijkstra \[34\]; they integrate specification and executable constructs in a unified language. This integration is the key for a stepwise development process in which a program is developed through a series of transformations within a single language.

Refinement calculi are convenient for describing object-oriented developments, as we can specify classes at various abstraction levels. As behavioural subclassing involves intermingled programs and specifications \[3\], refinement calculi are a natural choice because it unifies specifications and program code in a single language.

Utting \[87\] extended a refinement calculus to support object-oriented programming. He defines a model for multiple dispatch late binding and specialises this model to deal with single dispatch. He also formalised the notion of modular reasoning in which all objects are ordered by a substitution
1.3. REFINEMENT CALCULI AND OBJECT-ORIENTATION

relation. In his definition, an object of a class \( A \) can be substituted for an object of class \( B \), if methods of \( A \) are refined by methods of \( B \). He separates implementations and specifications (types), and checks behavioural conformance of types to their supertypes. Data refinement is only allowed between the implementation and a specification of an object. Utting does not consider visibility control, and recursive method calls. Also, he does not propose object-oriented programming laws.

Mikhajlova and Sekerinski [63] define a language in which all attributes are private and methods are public. Class constructors are concerned only with object creation and are not part of the class interface. They also define a refinement relation between classes which is based on the algorithmic and data refinement supported by the refinement calculus. In their approach, a class \( C_1 \) is refined by a class \( C_2 \) if the constructor of \( C_2 \) refines that of \( C_1 \), and each method of \( C_2 \) refines the corresponding method of \( C_1 \). Subclassing is a syntactic relation between classes, implying just in conformance of interfaces. They allow contravariance of input parameters and covariance of output parameters. In order to establish behavioural subclassing, they require that declaring one class as a subclass of another raises the proof obligations that class refinement holds between these classes.

Interface refinement is proposed by Mikhajlova and Sekerinski as a generalisation of class refinement, as it introduces a refinement relation for input and output parameters of the corresponding methods of two classes. They also define client refinement as being of two types: implicit and explicit. In implicit client refinement, a client class does not know that the class of which it is a client is refined, whereas in the explicit case the refinement is known. They do not present laws for object-oriented programming. Leino [54] has extended existing refinement calculi with object-oriented features, but restricting inheritance and not dealing with classes and visibility.

Cavalcanti and Naumann [21, 24, 22] present a language called ROOL, which is a subset of sequential Java. This language includes specification constructs from Morgan’s refinement calculus, recursive classes, visibility control, dynamic binding, and recursive methods. It has a copy semantics rather than a reference semantics. This simplifies the semantics: ROOL has a predicate transformer semantics allowing us to reason about object-oriented program development and to study formal refinement of programs. The imperative constructs of ROOL are based on Morgan’s refinement calculus [64]. In particular, the syntax of commands is based on that of Dijkstra’s language of guarded commands [34].

In the context of the refinement calculus for imperative programming, there are well established laws that can assist and form a basis for formal program development [64]. Indeed, the laws for imperative programming are well known [46]. In a response to the lack of formal programming laws for object-oriented programming [13], Borba and Sampaio [14] present a set of basic laws for ROOL. These laws deal with imperative commands of ROOL as well as with medium-grain object-oriented constructs. Cornélio et al. [33] address the proof of the soundness of the laws of commands of ROOL. Borba et al. [16, 17] present a comprehensive set of laws for object-oriented programming. They concentrate on object-oriented features, and they show that this set of laws is sufficient to transform an arbitrary program into a normal form expressed in terms of a small subset of the
language operators. There is already a mechanisation of the normal form reduction strategy [57]. These laws not only clarify aspects of the semantics, they also serve as a basis for deriving more elaborate laws and for practical applications of program transformations like those we present in this work. In [33, 36], these laws are used to prove rules that support compiler construction in the algebraic style proposed by Sampaio [75].

1.4 Objectives

We propose an approach to refactoring that is based on transformation rules between meta-programs in rool [21, 22]. Each rule is described by means of a meta-program on its left-hand side and another target meta-program on the right-hand side. Moreover, each rule has a set of conditions that must be satisfied in order to allow the rule to be applied. In this way, we still have refactorings described in the same language as the one that we use to write programs.

An object-oriented language with a formal semantics is essential for the proof that program transformations are semantics preserving. We adopt the notion that a behaviour-preserving transformation is a semantics-preserving transformation. Behaviour preservation implies that a program behaves the same before and after a transformation. The behaviour of a program, what is expected from a program execution, is expressed as the meaning of a program. The notion of behaviour preservation we adopt in this work is related to sequential programs that do not involve real-time requirements, and are not used as components nor are part of a framework as changing them may impact clients.

Having a set of basic laws for object-oriented programming is crucial for the derivation of more elaborate programming laws that can be useful for the practical development of programs. Our main objective in this work is to formalise and prove refactoring practices as those presented by Fowler [42]. Other refactoring rules also arise from the process of formalisation of already registered refactoring rules.

The laws of object-oriented programming proposed by Borba et al. [16, 17, 14, 15] for command and classes form a basis for the proofs of the program transformations described by refactorings. A program that appears on the left-hand side is, by means of law applications, transformed into another program, the one that appears on the right-hand side, provided some side-conditions are satisfied. We prove the soundness of the laws against the weakest preconditions semantics of rool [21, 22].

Also, data refinement is required for the derivation of refactoring rules. We propose a law for change of data representation inside a class, which is similar to the traditional data refinement law for a single program module [64]. Besides that, we use a law for change of data representation in class hierarchies: a generalisation of traditional data refinement laws.

Refactoring an object-oriented system has the purpose of obtaining a better design. Furthermore, refactoring may have the objective of obtaining a system that is structured in accordance
1.4. OBJECTIVES

Design Patterns

Refactoring Rules

Laws of Commands | Laws of Classes | Data Refinement Laws

Semantics

Figure 1.3: Formalisation of Refactorings

with a design pattern [30, 40, 43]. Design patterns capture knowledge of software experts: a pattern is a solution to a problem in a given context. The design embodied in a pattern may not be realised in a software because, for instance, a designer is unfamiliar with design patterns. In this case, refactoring may be necessary; code that conforms to design patterns can be obtained by the application of several refactorings.

We explore the application of refactoring rules for obtaining programs that are in accordance with a design pattern [43]. Differently from refactoring rules, design patterns are not presented as rules, but as development strategies. The reason is that a program must match the left-hand side of a rule and satisfy its side-conditions in order for the rule application to be possible. Design patterns, however, are a possible goal of object-oriented refactoring; it is difficult to identify the class of programs that can or should be redesigned. Here, we apply refactorings to a small particular system, which we use as a case study, in order to obtain a final system according to a design pattern. We also deal with the transformation of a poorly-structured system into one that is in accordance with an architectural pattern.

We summarise the strategy we follow for the formal derivation of refactoring rules in Figure 1.3. Some of these rules were initially presented in [32]. We use programming laws that deal with command, classes and also laws for data refinement in order to derive refactoring rules. Based on refactoring rules and, eventually in data refinement, we transform a system into one structured according to a design pattern.

In summary, the objectives of this thesis are as follows.

1. Formalisation of refactorings already available in the literature;
2. Identification of new refactoring rules as a result of the formalisation process;
3. Proof of the soundness of the refactoring rules by the application of programming laws that deal with commands, object-oriented constructs like classes and methods, and simulation;
CHAPTER 1. INTRODUCTION

4. Exemplification of the introduction of design and architectural patterns from particular systems by applying refactoring rules and laws of programming;

5. Proof of the programming laws of ROOL that deal with commands of the language;

6. Proof of the programming laws that deal with object-oriented features.

The study of the data refinement laws is left as future work. The soundness of simulation, however, has already been established in [23].

1.5 Thesis Overview

In the next chapter we present a survey of previous work on refactoring. We present the language ROOL and its semantics in Chapter 3, where we also present some laws of ROOL.

We define that refactorings which, when a applied to a class, for instance, do not change other parts of a system to be compositional. They do not affect the context in which a class that is being refactored appears. We present these refactorings in Chapter 4 along with their proofs.

In Chapter 5 we present refactorings that might change the context in which the class that is being refactored appears. These refactorings are said to be contextual, and their proofs are usually in the form of development strategies.

The application of refactoring rules and other object-oriented programming laws to a system, with the aim of obtaining a design in accordance with a well-known pattern, is exemplified in Chapter 6. In this chapter we also present an example of a poorly structured system that is transformed into a well structured one which follows a layered architectural pattern.

Finally, in Chapter 7 we summarise the contributions of this research and describe future work.
Chapter 2

Refactoring—State of the Art

In this chapter we present a survey of works related to refactoring. First, we present works on program restructuring in contexts other than object-oriented programming. Then, we present related works on refactoring. Finally, we present works on design patterns.

2.1 Program Restructuring

In [45], Griswold investigated meaning-preserving transformations to restructure programs written in a block-structured programming language. The language he analysed in his research was Scheme. Many transformations are well-known compiler optimisations or their inverses, like extracting or inlining a function. However, his transformations have a different aim from compiler optimisations; his transformations concern program restructuring for aiding maintenance, but are, in fact, similar to local compiler optimisations. In order to ensure that the transformations are meaning preserving, he uses Program Dependence Graphs to reason about the correctness of transformation rules. His research focused on transformation rules of the syntactic constructs of a block-structured language, so these transformations do not take into account inheritance matters. He recognises that class hierarchies complicate transformations and make analysis and transformations more complex. He discusses how his approach might be applied to object-oriented systems, for dealing, for instance, with method extraction.

The Demeter system provides a high-level interface to class-based object-oriented systems. The well-known Law of Demeter originated from work with this system. The goal of the law is organise and reduce the behavioural dependence between classes to make sure that methods have limited knowledge of an object model [55]. A proof that any object-oriented program written in a bad style can be systematically transformed into a program that obeys the Law of Demeter was presented [56]. An algorithm that transforms any object-oriented program into an equivalent program which satisfies the law is available. The algorithm uses a data structure known as the class dictionary graph. The vertices of the graph are classes; construction vertices are instantiable, whereas alternation vertices denoted abstract classes. Two types of edges represent the relationship
between two vertices. Alternation edges represent inheritance relationship, whereas construction
dges represent part-of relationship “uses” and “knows”.

Bergstein [9] presents a small set of primitive object-preserving class transformations, that is,
the reorganisation of a class hierarchy does not change the set of objects which the classes define
and programs after a transformation accept the same inputs and produce the same outputs as
before a transformation. These primitive transformations help form a theoretical basis for class
organisation. The set of transformation is shown to be correct. Bergstein’s rule for abstracting
common parts in a hierarchy can be seen as a derived rule in the framework presented in [17, 16].
Bergstein’s rule is similar to refactoring for pulling up and pushing down attributes and methods.
There is no argument for completeness in terms of a normal form expressed in terms of a small
set of object-oriented constructs as in [17, 16]. Consequently, his notion of completeness does not
cover all possible transformations that can be applied to object-oriented programs. In particular,
there are no transformations for dealing with type tests and casts, nor he deals with type changes.

Banerjee and Kim [8] applied restructuring operations in the context of database schema evo-
lution. They defined a set of schema transformations, which are used for schema evolution: the
dynamic definition and subsequent changes to a database schema in an object-oriented database
environment. They identified a set of invariant properties of an object-oriented schema which must
be preserved across schema changes, for instance, attributes of a class, whether defined or inher-
ited, have distinct names. There are no rules allowing changing the location of a method in a class
hierarchy.

2.2 Refactoring

The seminal work on the formalisation of refactoring of object-oriented programs was presented
by Opdyke [69]. He identified 23 primitive refactorings and gave examples of three composite
refactorings. Each primitive refactoring has a set of preconditions that would ensure the behavior
preservation of the transformation. Behavior preservation is argued in terms of seven program
properties, which are related to inheritance, scoping, type compatibility, and semantic equivalence.
The properties are the following:

1. **Unique Superclass**: every class must have exactly one superclass.

2. **Distinct Class Names**: every class in the system must have a unique identifier.

3. **Distinct Member Names**: attributes and methods have unique names in a single class. Meth-
   ods can be redefined in subclasses.

4. **Inherited Member Variable Not Redefined**: a subclass cannot redefine an attribute of its
   superclass.
5. **Compatible Signatures in Member Function Redefinition:** redefinitions of methods have the same signatures as the redefined method.

6. **Type-Safe Assignments:** every expression that is assigned to a variable must be of the type or a subtype of the type of the variable.

7. **Semantically Equivalent References and Operations:** operationally, it means that before and after a refactoring, a program has to produce the same output for a given set of inputs.

The importance of the achievement of Opdyke is not only the identification of refactorings, but also the definition of the preconditions that are required to apply a refactoring to a program without changing its behaviour. Each refactoring is (informally) shown to be behaviour-preserving by arguing that the preconditions satisfy the seven properties above.

Roberts [74] goes a step further than Opdyke: he gives a definition of refactoring that focuses on their pre- and post-conditions. The definition of post-conditions allows the elimination of program analysis that are required within a chain of refactorings. This comes from the observation that refactorings are typically applied in sequences intended to set up preconditions for later refactorings. Pre- and postconditions are all described as first-order predicates; this allows the calculation of properties of sequences of refactorings.

Roberts also takes the position that a refactoring is correct if a program that meets its specification continues to meet its specification after the refactoring. A suite of tests is understood as a form of specification; the definition of correctness is based on test suites. In summary, a refactoring is correct if a program that passes a test suite continues to pass the test suite after the refactoring. There is no semantic-based proof that refactoring preserves the behaviour of a program or continues meeting its specification. He recognises that formal proofs of semantically equivalent references and operations are difficult to produce. His definition of refactoring is simply a program transformation that has a precondition that a program must satisfy for the refactoring to be legally applied. According to him, this avoids formal proofs of correctness. Roberts also examines techniques for using run-time analysis to assist refactoring. He discusses dynamic refactoring in which the program, while running, checks for certain properties, applies appropriate refactorings, and then can retract those refactorings.

Roberts automates the basic refactorings proposed by Opdyke; composite refactorings can be defined based on the basic refactorings. As part of his research, he developed the Refactoring Browser, a tool to refactor Smalltalk programs [73].

Notice that both Opdyke and Roberts formalise refactorings for automation purposes only. For this reason, the condition **Semantically Equivalent References and Operations** presented by Opdyke cannot be strictly checked. From Robert’s work, it is clear that the specification that a program meets is a test suite, not a description formalised as a first-order predicate, for instance.

Most of the low-level refactoring presented by Opdyke are described by laws of programming in ROOL [17][16]. For instance, refactorings `delete_member_functions` and `create_member_function` can
be seen as applications of law (method elimination) from left to right, and from right to left, respectively. In the case of delete_member_functions, maybe law (method elimination) should be applied more than one time. On the other hand, Opdyke’s refactoring convert_instance_variable_to_pointer cannot be described in ROOL as it has a copy semantics. Other refactorings we have not addressed are change_class_name and a similar one that concerns variable name. Class and variable renaming are purely syntactic operations. Opdyke also presents composite refactoring that are built on low-level refactorings. The composites refactorings abstract_access_to_member_variable and convert_code_segment_to_function are described, in the present work, as (Encapsulate Field) and (Extract Method), respectively.

Roberts implements a subset of the refactorings proposed by Opdyke. Only those related to renaming are not addressed in the present work. Since Robert’s work extends that of Opdyke, some refactorings presented by Roberts are described as programming laws in ROOL [17, 16]. For instance, refactorings Pull Up Instance Variable and Push Down Instance Variable are similar to law (move attribute to superclass) when applied from left to right, and from right to left, respectively.

Tokuda [86, 85] uses the properties proposed by Opdyke for behaviour preservation. He implements the refactorings proposed by Opdyke for C++, and others that are not listed in Opdyke’s work, like inherit, which establishes a superclass-subclass relationship between two existing classes. Tokuda views a refactoring as a parameterised behaviour-preserving program transformation. Refactorings check enabling conditions to ensure that program behaviour is preserved, identify source code affected by a change, and execute all changes. His experiments and analysis showed that the invariants proposed by Opdyke are not sufficient due to complexities introduced by the language being transformed. For this reason, when a refactoring was found to change the behaviour, he defined new invariants. One of these new invariants is No instantiation side-effects, which requires the constructor of a class to have no side-effects besides initialising the object created. He also identified new refactorings.

Tokuda takes the position that refactorings are behaviour-preserving due to good engineering and not to any mathematical guarantee. He argues that, given a mature refactoring implementation, refactorings should be treated as trusted tools in the same way as compilers transform source code to assembly even without mathematical proof to guarantee correctness. As Tokuda’s focus is the implementation of refactorings for the language C++, we cannot describe a refactoring like decorator, which involves pointers. In fact, he also defines refactorings based on design patterns [43]. We do not address the definition of transformation rules to introduce design patterns in a single step.

Fowler [42] presents a catalog of refactorings. Each refactoring is given a name and a short summary that describes it. A motivation describes why the refactoring should be done; there is also a mechanic, a step-by-step description of how to carry out the refactoring, and, finally, an example. Fowler suggests that, before starting a refactoring, one should have a solid suite of tests
2.2. REFACTORING

that must be self-checking. Every change must be followed by program compilation and test. There are no conditions to be satisfied in order to guarantee behaviour preservation. In fact, Fowler’s approach to refactoring is based on compilation and test cycles. His book is a landmark in making refactoring known to programmers in general.

Back [6] studies a method for software construction that is based on incrementally extending the system with a new feature at a time. He refers to this method as stepwise feature introduction. Introducing a new feature may destroy some already existing features, so the method must allow checking that old features are preserved. A layered software architecture is proposed to support this method. He also takes into account correctness conditions and reasons about their satisfaction in the refinement calculus. He assumes that each class in a system has a class invariant, which expresses the conditions on the attributes that must be established when the class is instantiated, and which must be preserved by each operation on the class. Methods have preconditions, which state the assumptions that must hold when the methods are called, and possibly postconditions, which express properties that hold when the calls return. Data refinement is used to prove the correctness of an implementation. Although the approach seems similar to ours, no programming laws are presented or are explicitly used for refactoring programs.

2.2.1 Formalisms

A variety of formalisms has been used to deal with restructuring and refactoring. Snelting and Tip [77] use concept analysis to restructure class hierarchies. Their method analyses a class hierarchy along with a set of applications that use it. A table is constructed that precisely reflects the usage of the class hierarchy. A concept lattice is constructed from the table, which factors out information that variables, for instance, have in common. Situations in which a class can be split can also be detected. They showed that the technique is capable of finding anomalies such as redundant attributes. The class hierarchy that results from the application of the proposed technique is guaranteed to preserve the behaviour of the original hierarchy. The formal basis of this work is concept analysis.

Program slicing [83, 10] deals with a specific kind of restructuring: function or procedure abstraction. Lakhotia and Deprez [52] present a transformation called tuck for restructuring programs by decomposing large functions into small functions: it breaks large code fragments and tucks them into new functions. The challenge they faced was creating new functions that capture computations that are meaningfully related. There are three basic transformations to tuck functions: (1) related code is gathered by driving a wedge (which is a program slice bounded with single-entry and a single-exit point) into the function, then (2) the code isolated by the wedge is split, and (3) the split code is folded into a function. These transformations even create functions from non-contiguous code.

Komondoor and Horwitz [51] address the conditions under which it is possible to move a set of selected statements together so that they can be extracted while preserving semantics. They use
control flow graphs to represent pieces of code. They present an algorithm that move a selected set of control graph nodes together so that they can be extracted whilst preserving the semantics. They identified conditions based on control and data dependence that are considered to be sufficient to guarantee semantic equivalence.

Restructuring can also be dealt with by means of graph transformations. The software is represented as a graph, and restructuring corresponds to transformation rules. Graph rewriting appears as a lightweight formalism [60]. Mens, Demeyer, and Janssens [61] present the formalisation of refactoring using graph rewriting, a transformation that takes an initial graph as input and transforms it into a result graph. This transformation occurs according to some predefined rules that are described in a graph-production which is specified by means of left-hand and a right-hand sides. The first one specifies which parts of the initial graph should be transformed, while the last one specifies the result after transformation. Well-formedness is expressed by means of type graphs and forbidden subgraphs. A type graph is a meta-graph expressing restrictions on the instance graphs that are allowed. A graph is well-formed only if there exists a graph morphism into a type graph. Forbidden graphs exclude illegal configurations in a graph, so that a graph satisfies the constraint expressed by a forbidden graph if there does not exist a morphism between the graph and the forbidden graph. The notion of equivalence is that for each refactoring, one may catalog that types of behaviour that need to be preserved. A refactoring is access preserving if each method implementation accesses at least the same variables after refactoring as it did before the refactoring. They consider also two other types of behaviour: update preserving, a method updates at least the same variables after a refactoring as it did before the refactoring; and call preserving if each method implementation performs at least the same method calls after a refactoring as it did before refactoring.

Graph rewriting is considered a suitable formalism for specifying refactoring because graphs are a language-independent representation of the source code, rewriting rules are considered precise and concise to specify source code transformation, and the formalism allows proving behaviour preservation. However, they recognise that it is difficult to manipulate nested structures in method bodies in refactoring such as move method and push down method so that it is necessary to use techniques that tackle the inevitable complexity of large graphs.

These formalisms are usually used for the description of transformations or are used as the formal basis for transformations so that it is possible to guarantee that they do not change program behaviour. However, they are not concerned with giving a language semantics, but describing a transformation. In this way, they may be useful in a refactoring tool, allowing us to check refactoring preconditions. Besides these formalisms, Philipps and Rumpe [70] suggest the existing refinement approaches are a way to formally deal with the notion of behaviour preservation required by refactorings. Behaviour preservation is not a notion specific to the domain of refactoring, it also occurs, for instance, in the area of refinement techniques [65, 7]. Our work is in this direction. We describe refactoring by using a language that has a weakest precondition semantics and a set of
2.2. REFACTORING

laws effectively used in the derivation of refactoring rules.

2.2.2 Languages

There are definitions for restructuring programs written in different programming languages. As we have already seen, the work of Griswold [45] deals with restructuring programs written in the functional programming language Scheme. Thompson and Reinke [81, 82] addressed refactoring of programs written in the Haskell programming language. They characterise refactoring as diffuse—refactoring requires changes throughout a module or a system of modules—, and bidirectional—it can be applied in one direction and in the reverse direction. As an example of refactoring, they present *demoting definition*, a refactoring that moves the definition of an auxiliary function to the scope of the function that calls it, since the auxiliary function is not used elsewhere.

Class-based object-oriented languages have already been addressed. Roberts, Brant, and Johnson [73] present a tool for refactoring Smalltalk programs. Refactoring for Java programs is presented by Fowler [42]. Tokuda and Batory [85] automate refactorings for C++.

2.2.3 Refactoring Models

Refactorings can also be applied at higher levels of abstraction than source code. Design models, for instance, can be the target of refactorings. These models can be specified using, for example, the Unified Modelling Language [12].

Sunyé et al. [80] present a set of design refactorings for models described in the Unified Modelling Language. They present refactorings of class diagrams and statecharts. In order to guarantee behaviour preserving transformations of statecharts, they specify the constraints that must be satisfied before and after the transformation using the OCL at the meta-model level.

Astel [2] proposes using an UML tool as an aid in finding smells—a structure in code that suggest the possibility of refactoring—and performing some elaborate refactorings. It is a tool that bases class diagrams directly on code, allowing code manipulation by the direct manipulation of the diagram. Among the reasons for refactoring in UML, he highlights the fact that many people prefer to visualise classes and their relationships, and that the level of abstraction is higher when compared to code. Also, smell detection can be done by visualising the classes of a system. For instance, it is easy to visualise large classes. Refactoring can be done by simple drag-and-drop actions. He argues that it is necessary to use a tool that generates diagrams from code, and the tool needs to keep the code and the model synchronised.

Gheyi and Borba [44] introduce and formalise modelling laws; their emphasis is on refactoring of models described in Alloy [47]. An Alloy model is a sequence of signatures, which are used to define new types, and formulas, used to record constraint information. Besides a basic type, a signature introduces relations. The basic laws they propose deal with properties of signatures, formulas, and relations. The laws they propose are supposed to be the basic transformations that
serve as a basis for more elaborate laws for practical applications of model transformation.

Bottoni, Parisi-Presicce, and Taentzer [18] present an approach to maintain the consistency of specification and code after refactoring. The specification can be composed of UML diagrams of different types; they show that some refactorings require modifications in several diagrams at once. Refactorings are expressed by pre- and postconditions. To ensure consistency between source code and structural and behavioural models, they use graph transformations. Both code and models are represented by graphs. Each refactoring is described by means of a set of transformation schemes.

Porres [71] focus on the implementation of refactoring as a collection of transformation rules, which receive one or more model elements as parameters, and perform a basic transformation based on the parameters. They use their own scripting language SMW to manipulate models based on the Python programming language. A metaclass of the metamodel of UML is written as a class in Python; SMW scripts resemble OCL. One of the elements of the transformation rule is a guard that defines when the rule can be applied; there is also a body that implements the effect of the rule. As refactorings are group of rules, the guard of one rule can refer to the guards of other rules in the same transformation. The execution of a transformation is described by a sequential algorithm that accepts a transformation to apply and a set of model elements.

A refactoring transformation is considered to be correct if the transformation terminates: the transformed model is syntactically correct, and the transformation preserves some observable properties of the model. Porres argues that the number of rules in a transformation and the number of elements in a model are finite, so the transformation terminates. Syntactic correctness is ensured by the fact that rules give as results well-formed models. Behaviour preservation requires a semantic interpretation of UML given, for example, by graph transformations.

Boger et al. [11] present a refactoring browser integrated in a UML tool. They concentrate on the detection of conflicts that may be introduced after the application of a refactoring. They classify conflicts as warnings and errors. Warnings indicate that conflicts might cause a side effect. For instance, they consider that renaming a method that overrides a method of a superclass may be behaviour preserving in some cases, but an unwanted design change in others. Errors indicate that an operation will cause damage to the model or code. They also address refactoring of state machines, like merging of states and formation of composite states. In our case, we rule out any kind of conflicts, because we must always preserve a program’s behaviour. As a consequence, we avoid method overriding in refactorings.

2.3 Design Patterns

Patterns record the knowledge and expertise that has been built up along many years of software engineering. They can be found in any part of the development process, for instance, architecture, analysis, and design.

Patterns can also arise in specific areas like real-time programming. In fact, patterns come from
the observation of existing systems, motivated by the desire to uncover solutions that are repeatedly applied. In the context of the design of object-oriented systems, Coad [30] presents the concept of patterns and its application to object-oriented analysis and design. He also explores seven patterns, presenting them by means of graphical notation, a textual description, and guidelines for the application of each pattern.

Gamma et al. [40] propose design patterns as a mechanism for expressing design structures. In that work, they present a catalog of design patterns that they have discovered when building their own class libraries and collected from the literature. Besides that, they classify patterns according to their common aspects [43].

Cinnéide [29, 28] discussed the automatic introduction of design patterns through the application of refactorings. In developing a transformation for a particular design, certain motifs, observed to occur across catalogues of patterns, are defined as minipatterns that are combined in various ways to produce different patterns. For each minipattern identified, a minitransformation is developed, which comprises a set of preconditions, a sequence of transformation steps, a set of postconditions and an argument demonstrating behaviour preservation. Each minitransformation is defined in terms of low-level refactorings.

2.3.1 Formalization of Design Patterns

A formal description of design patterns has already been provided by Flores et al. in [41], where elements that constitute a general object-oriented design and their formal model are presented. They use the RAISE Specification Language to formally specify properties of design patterns. In fact, they introduced a general model that allows describing an arbitrary object-oriented design and not only patterns. They also formally specify how to match a design against a pattern. In this way, its is possible to verify that a given subset of a design corresponds to a given pattern. This link is given by using a renaming map, which associates the names of entities (classes, methods, attributes, and parameters) in the design with the names of corresponding entities in the pattern. Several consistency conditions must be satisfied in the renaming.

Eden [38] uses a declarative language called LePUS (LanguagE for Pattern Uniform Specification) [39], which is mostly graphic. A program in LePUS is modelled as a set of entities (classes and methods) and relations (inheritance, method invocation, object creation etc). Every well-formed LePUS diagram translates to a formula in higher-order logic that allows reasoning about specifications. LePUS formulae are used to describe design patterns in the form of logic statements; patterns are transcribed to formulae.

Lano et al. [53] used theories similar to those used for giving the semantics of VDM++ [49], which consist of a collection of type, constant, attribute and action symbols, and a set of axioms describing the types of attributes, the effects, and the dynamic properties of the actions. A system $D$ is said to refine a system $C$ if there is a theory interpretation from the theory of $C$ to the theory of $D$. They characterise design patterns as a transformation from a “before” system consisting of
a set of classes into an “after” system consisting of a collection of classes organised according to a design pattern. They prove that the “after” system is an extension, via a suitable interpretation, of the theory of the “before” system. An extension usually introduces new symbols which are defined by axioms. They use VDM++ to write the “before” and “after” systems, then they establish an interpretation between these systems.

2.4 Conclusions

In this chapter we presented a survey about program restructuring. We concentrated on works about refactoring, with focus on the description of works related to formalisms for describing refactoring, target languages for refactoring, and model refactoring. We also presented a survey of works on design patterns, mainly related to their formalisation.

As can be observed from the presentation of current works, no work on the formalisation of refactoring relies in a uniform basis for the description of program transformations. In other words, to prove that a refactoring is correct, it is described using a specific formalism like graph transformations. Existing refinement techniques can also be used as tools for the proof of correctness of refactorings. We can prove that a refactoring is correct in a uniform way, without changing the language which is used to present a refactoring. A refactoring could be represented as a transformation from a program to a refactored one, both written in a language, and the transformation from one to the other expressed almost in the same language used to write the program as we introduce meta-variables for classes, attributes, methods, local variables. Such language must have a formal semantics and laws that serve as a sound basis for software development.
Chapter 3

ROOL and Laws

ROOL [21][22], an acronym for Refinement Object-oriented Language, is a Java-like imperative language with classes, inheritance, visibility control for attributes, dynamic binding, and recursion. It allows reasoning about object-oriented programs and specifications, since both kinds of constructs are mixed as in refinement calculus languages [64][65]. The semantics of ROOL, as usual for refinement calculi, is based on weakest preconditions. The imperative constructs of ROOL are based on the language of Morgan’s refinement calculus [64], which is an extension of Dijkstra’s language of guarded commands. In a refinement calculus, specifications are regarded as commands. In fact, we use the term command to refer to commands, in its usual sense, and programming constructs in which specifications and commands are mixed.

This chapter is organised as follows. First we present the abstract syntax of ROOL, then we present its typing system, its semantics, a notion of program and class refinement, and, finally, a list of programming laws. The sections about the syntax, typing, semantics, and refinement are based on the technical report that introduces the language ROOL along with its weakest precondition semantics [22]. The section that presents the laws of ROOL is based on [16][17][14][15][33].

3.1 Syntax

First, we define the data types for ROOL. Data types are either class names (N) or primitive (bool, int, and others). Data types T are the types of attributes, method parameters, local variables, and expressions.

\[ T \in \text{Typ} ::= N \mid \text{bool} \mid \text{int} \mid \ldots \text{other primitive types} \]

For writing expressions, ROOL provides typical object-oriented constructs (Table 3.1). We assume that x stands for a variable identifier, and f for a built-in function; self and super have a similar semantics to this and super in Java, respectively. The type test e is N has the same meaning as in e instanceof N in Java: it checks whether non-null e has dynamic type N; when
CHAPTER 3. ROOL AND LAWS

\[ e \in \text{Exp} ::= \text{self} | \text{super} \quad \text{special 'references'} \\
| \text{null} | \text{error} \\
| \text{new } N \quad \text{object creation} \\
| x \quad \text{variable} \\
| f(e) \quad \text{application of built-in function} \\
| e \text{ is } N \quad \text{type test} \\
| (N)e \quad \text{type cast} \\
| e.x \quad \text{attribute selection} \\
| (e; x : e) \quad \text{update of attribute} \]

\[ \psi \in \text{Pred} ::= e \quad \text{boolean expression} \\
| \psi \Rightarrow \psi \\
| (\lor i \cdot \psi_i) \\
| \forall x : T \cdot \psi \\
| e \text{ isExactly } N \quad \text{strict type test} \]

Table 3.1: Grammar for expressions and predicates

\( e \) is \text{null}, it evaluates to false. The expression \((N)e\) is a type cast; the result of evaluating such an expression is the object denoted by \( e \) if it belongs to the class \( N \), otherwise it results in error. Attribute selection \( e.x \) results in a run-time error when \( e \) denotes \text{null}. The update expression \((e_1; x : e_2)\) denotes a copy of the object denoted by \( e_1 \) with the attribute \( x \) mapped to a copy of \( e_2 \). If \( e_1 \) is \text{null}, the evaluation of \((e_1; x : e_2)\) yields \text{error}. Indeed, the update expression creates a new object rather than updating an existing one.

The expressions that can appear on the left of assignments, as the target of a method call, and as result arguments constitute a subset \( Le \) of \( \text{Exp} \). They are called left-expressions.

\[ le \in \text{Le} ::= le_1 \mid \text{self}.le_1 \mid ((N)le).le_1 \]

\[ le_1 \in \text{Le} ::= x \mid le_1.x \]

The predicates of ROOL (Table 3.1) include expressions of type \text{bool}, formulas of the first-order predicate calculus, and strict type tests of the form \( e \text{ isExactly } N \).

The imperative constructs of ROOL, including those related to object-orientation concepts, are specified in the Table 3.2. In a specification statement \( x : [\psi_1, \psi_2] \), \( x \) is the frame, and the predicates \( \psi_1 \) and \( \psi_2 \) are the precondition and postcondition, respectively. It concisely describes a program that, when executed in a state that satisfies the precondition, terminates in a state that satisfies the postcondition, modifying only the variables present in the frame. In a state that does not satisfy \( \psi_1 \), the program \( x : [\psi_1, \psi_2] \) aborts: all behaviours are possible and nontermination too. The variable \( x \) is used to represent both a single variable and a list of variables; the context should make clear the case. Two specification statements are distinguished: the first is \( x : [\text{false}, \text{true}] \)—we also refer to it as \text{abort}—which is never guaranteed to terminate (precondition \text{false}), and when it does, it can assign any values to the variables in \( x \) (postcondition \text{true}); the second is the
3.1. SYNTAX

Table 3.2: Grammar for commands and parameterised commands

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c \in \text{Com}$</td>
<td>::= $le := e$ multiple assignment</td>
</tr>
<tr>
<td></td>
<td>$</td>
</tr>
<tr>
<td></td>
<td>$</td>
</tr>
<tr>
<td></td>
<td>$</td>
</tr>
<tr>
<td></td>
<td>$</td>
</tr>
<tr>
<td></td>
<td>$</td>
</tr>
<tr>
<td></td>
<td>$</td>
</tr>
<tr>
<td></td>
<td>$</td>
</tr>
<tr>
<td></td>
<td>$</td>
</tr>
<tr>
<td>$pc \in \text{PCom}$</td>
<td>::= $pds \bullet c$ parameterisation</td>
</tr>
<tr>
<td></td>
<td>$</td>
</tr>
<tr>
<td></td>
<td>$</td>
</tr>
<tr>
<td>$pds \in \text{Pds}$</td>
<td>::= $\emptyset</td>
</tr>
<tr>
<td>$pd \in \text{Pd}$</td>
<td>::= $\text{val } x : T</td>
</tr>
</tbody>
</table>

specification $x : [\text{true, false}]$, also known as miracle, which terminates when execute in any state and establishes false as postcondition.

In program derivation, it is also useful to assume that a condition $\phi$ holds at a given point in the program text. This can be characterised as an assumption of $\phi$, written $\{ \phi \}$, whose definition is given by the specification statement : $[\phi, \text{true}]$. If $\phi$ is false, the assumption reduces to abort. Otherwise, it behaves like skip, a program that always terminates and does nothing. The definition of skip is given by the specification statement : $[\text{true, true}]$. The empty frame guarantees that no variables are changed.

Complementary to assumptions are coercions. A coercion to $\phi$, written $[\phi]$, whose definition is given by the specification statement : $[\text{true, } \phi]$, behaves like skip if $\phi$ is true, and miracle otherwise.

We define methods as parameterised commands in the same style as Back [4, 27], because Morgan’s approach may lead to some inconsistencies [26]. Parameterised commands can have the form $\text{val } x : T \bullet c$, or $\text{res } x : T \bullet c$, which correspond to the parameter passing mechanisms known as call-by-value, and call-by-result, respectively. The parameterised command application $pc(e)$ yields a command which behaves as the one obtained by passing the arguments $e$ to the body of the parameterised command. Parameters that are passed by different parameter passing mechanisms are declared in the usual way. For example, for parameters $x$ and $y$ which are passed by value and result, respectively, we have the following declaration: $\text{val } x : T; \text{res } y : T'$. A parameterised command with an empty parameter declaration behaves like an ordinary command.

For alternation, we use an indexed notation for finite sets of guarded commands. A method in ROOL can use its name in calls to itself in its body. This is the traditional way to define a recursive method. ROOL also includes the construct $\text{rec } Y \bullet c \text{ end}$, which defines a recursive command using
CHAPTER 3. ROOL AND LAWS

\[
\text{Program } ::= \text{cds} \bullet c \\
\text{cds} \in \text{Cds} ::= \emptyset \mid \text{cd} \text{cds} \\
\text{cd} \in \text{Cd} ::= \text{class } N_1 \text{ extends } N_2 \\
\quad \{ \text{pri } x_1 : T_1 ; \}^* \\
\quad \{ \text{prot } x_2 : T_2 ; \}^* \\
\quad \{ \text{pub } x_3 : T_3 ; \}^* \\
\quad \{ \text{meth } m \equiv (pds \bullet c)\}^* \\
\text{end}
\]

Table 3.3: Programs in ROOL

the local name \( Y \). A (recursive) call \( Y \) to such a command is also considered to be a command. The iteration command can be defined using a recursive command.

Blocks \text{var } x : T \bullet c \text{ end} and \text{avar } x : T \bullet c \text{ end} introduce local variables. The former introduces variables that are demonically initialised; their initial values are arbitrary. The latter introduces variables that are angelically chosen \[7\]. This kind of variable is also known as logical constant, logic variable, or abstract variable. In practice, angelic variables only appear in specification statements.

As methods are seen as parameterised commands, which can be applied to a list of arguments, yielding a command, a method call is regarded as the application of a parameterised command. A call \( le.m \) refers to a method associated to the object denoted by \( le \). In a method call \( e_1.m(e_2), e_1 \) must be a left expression.

A program in ROOL is a sequence of class declarations followed by a command (Table 3.3). A class declaration \text{cd} introduces a class named, for instance, \( N_1 \). The clause \text{extends} indicates the immediate superclass of \( N_1 \). If the superclass is omitted, the pre-defined class \text{object} is the superclass of the class being declared. The class \text{object} has no attributes or methods.

The visibility control clauses \text{pri}, \text{prot}, and \text{pub} introduce private, protected, and public attributes, respectively. The variables \( x_1, x_2, \) and \( x_3 \) along with their types \( T_1, T_2, \) and \( T_3 \) can be lists. Private attributes are visible just inside the class in which they are declared; protected attributes are visible inside the class in which they are declared and in its subclasses; and public attributes are visible to all declared classes. The visibility control clauses can appear zero or more times in a class declaration.

A class in ROOL can have a (possibly empty) list of method declarations. A method is introduced by a clause \text{meth } m \equiv (pds \bullet c), \) where \( m \) is the name of the method, and \( pds \bullet c \) is the body of the method: a parameterised command. All methods in ROOL are public.

An example of a class in ROOL is presented in Figure 3.1. The class \text{Person} has private attributes \text{name}, \text{areaCode}, and \text{homeTelNumber} of type \text{string}. The method \text{getName} has a result parameter \text{arg} to which is assigned the value of the attribute \text{name} of such class. The method
3.2. TYPING

```plaintext
class Person
    pri name : string;
    pri areaCode : string;
    pri homeTelNumber : string;
    meth getName ≡ (res arg : string • arg := self.name)
    meth getTelephoneNumber ≡ (res tel : string • tel := self.homeTelNumber)
end
```

Figure 3.1: A class in ROOL

`getTelephoneNumber` also has a result parameter and gives the telephone number of a person. This class is similar to the one presented in Figure 1.1

3.2 Typing

ROOL has as phrase types the data types $T$ and those for predicate formulas, commands, parameterised commands, and complete programs.

$$\theta ::= T \mid \text{pred} \mid \text{com} \mid \text{pcom}(pds) \mid \text{program}$$

In the next sections we define typing environment, and typing rules for commands and programs.

3.2.1 Typing environment

The typing environment is a record that registers the typing and inheritance information of a sequence of class declarations. It also registers typing information concerning method parameter and local variable declarations in scope. We use a dot for denoting field access, as usual.

There are two disjoint sets of names: the set $CNames$ of class names and the set $LNames$ of local names. A local may be either an attribute, a method, a method parameter, or a local variable. The class names `object` and `main` are distinct from all other class names. The class `object` is the superclass of all classes without an explicit superclass declared. The class `main` refers to the main part of a whole program in ROOL; it is not really a class.

A typing environment $\Gamma$ is a record with six fields: `attr`, `meth`, `vis`, `cnames`, `supcls`, and `locals`. The first of these, `attr`, is a finite partial function from $CName$ to $LSignature$. Formally, it is written $CName \rightarrow LSignature$. A $LSignature$ associates a local name with a type. Formally, we have $LSignature = LName \rightarrow T$. The field `attr` records the names and types of all declared and inherited attributes of every declared class. The `meth` field records the names and signatures of all declared and inherited methods of all classes. Formally, `meth` has type $CName \rightarrow MDecs$, where $MDecs = LName \rightarrow Pds$.

The third field of a typing environment, `vis`, records the visibility of attributes of the declared
classes. The type of vis is given by $CName \rightarrow (LName \rightarrow Visibility)$, where Visibility is given by the set \{pri, prot, pub, ipri\}. Private attributes that are declared in a class, and not inherited, are associated to pri; the inherited private attributes are associated to ipri; prot refers to the protected (either inherited or declared) attributes, and pub refers to the public attributes that were declared or inherited by a class.

The cnames field is a set that contains the names of all declared classes. This set is the common domain of attr, meth, and vis. The class name object is supposed to be in cnames, while main, which is not a class itself, is not in cnames. As the class object has no attributes or methods, its fields attr and meth are associated to the empty signature (empty set).

The supcls field of a typing environment associates each class name to the name of its immediate superclass. The type of supcls is $CName \rightarrow CName$. All declared classes have a direct superclass, which can be object. On the other hand, object itself does not have a superclass. Moreover, the inheritance relationship is not allowed to have circularities. Formally, we have $\text{supcls}^+ \cap \text{id}
\text{CName} = \emptyset$, where id $T$ is the identity relation on $T$. We define the subtype relation $\leq \Gamma$ based on the inheritance relationship recorded in supcls, which is defined by $T_1 \leq \Gamma T_2 \iff (T_1, T_2) \in (\Gamma.\text{supcls})^+ \lor T_1 = T_2$. If $T_1$ and $T_2$ are related by the transitive closure of $\Gamma.\text{supcls}$ or are equal, then $T_1$ is a subtype of $T_2$.

The last field of a typing environment, locals, records the types of the visible attributes of the current class, and of method parameter and local variables in scope. The type of locals is $LSignature$.

The types of phrases, in the context of a collection of class declarations, a specific class, and some method parameter and local variable declarations, are given by the typing relation $\triangleright$. For example, $\Gamma, N \triangleright c : \text{com}$ asserts that $c$ is a command that can appear in the body of a method in class $N$, in a context captured by the typing environment $\Gamma$. Similarly, $\Gamma, N \triangleright e : T$ attests that $e$ is an expression of type $T$ in a method of class $N$.

Any typing in the context of a typing environment $\Gamma$ and of a class $N$ holds only if $\Gamma$ is well-formed and derivable using the typing rules that are presented later. A typing environment $\Gamma$ is well-formed if the following conditions are satisfied. First, the conditions above hold. Secondly, the current class must be declared: $N \neq \text{main} \Rightarrow N \in \Gamma.\text{cnames}$. Thirdly, all visible attributes of $N$ must be present in the domain of the locals field. In other words, all but the inherited private attributes (those associated to ipri) must be in dom $\Gamma.\text{locals}$. If $N$ is main, then $\Gamma.\text{locals}$ contains only parameter and local variables.

In the next section we illustrate the typing rules for commands, parameterised commands, and programs. The whole set of rules can be found in Appendix $H$.

3.2.2 Typing Rules

The typing rules for expressions in ROOL are presented in the Table 3.4. The type of self is the one of the current class. The type of a null object is that of any declared class. The expression $e$ is $N''$
3.2. TYPING

\[
\begin{align*}
N \neq & \text{main} & N' \in & \text{enames} \\
\Gamma, N \triangleright \text{self} : N & & \Gamma, N \triangleright \text{null} : N' \\
\Gamma, N \triangleright e : N' & & N'' \leq \Gamma N' \\
\Gamma, N \triangleright e & & N'' : \text{bool} 
\end{align*}
\]

Table 3.4: Typing of Expressions

\[
\begin{align*}
(\Gamma; x : T) \triangleright c & : \text{com} & \text{par} \in \{\text{val, res}\} & \Gamma, N \triangleright \text{le} : N' & \Gamma.\text{meth} N' m = \text{pds} \\
\Gamma \triangleright (\text{par} x : T \bullet c) & : \text{pcom}(\text{par} x : T) & & \Gamma, N \triangleright \text{le} : N' & \Gamma.\text{meth} N' m = \text{pds}
\end{align*}
\]

Table 3.5: Typing of Parameterised Commands

\[
\begin{align*}
\Gamma \triangleright \text{le} : T & & \Gamma \triangleright e : T' & T' \leq \Gamma T & \text{sdisjoint le} \\
\Gamma \triangleright \text{le} := e & : \text{com} & & & & \\
\Gamma \triangleright (\text{res} x : T \bullet c) & : \text{pcom}(\text{res} x : T) & \Gamma \triangleright \text{le} : T' & T \leq \Gamma T' & \text{sdisjoint le} \\
\Gamma \triangleright (\text{res} x : T \bullet c)(\text{le}) & : \text{com} & & & & \\
\Gamma \triangleright \text{le}.m : \text{pcom}(\text{pds}) & & \Gamma \triangleright e : T & \Gamma \triangleright \text{le}.m : \text{pcom}(\text{pds}) & \text{aptype} \Gamma \text{pds} e T \\
\text{sdisjoint(}\text{le}, \text{rvargs} \text{pds} e \text{)} & & \Gamma \triangleright \text{le}.m : \text{com} & & \\
\end{align*}
\]

Table 3.6: Typing of Commands

is well-typed when the type of \(e\) is a superclass of \(N''\). This condition is also applicable to the type cast \((N'')e\).

Typing rules for parameterised commands appear in Table 3.5. The type of a parameterised command records its parameter declarations. The parameterised command \((\text{par} x : T \bullet c)\) has type \(\text{pcom}(\text{par} x : T)\), provided the command \(c\) is well-typed in an environment that records the parameters \(x\). The parameter passing mechanism is any of those present in ROOL \((\text{val, res})\). The parameterised command \(\text{le}.m\) is well-typed only if the type of \(\text{le}\) is a class with a method named \(m\). The method parameters are recorded in the \(\text{meth}\) field of the typing environment.

**Commands**

The typing rules for commands are presented in Table 3.6. An assignment \(\text{le} := e\) is well-typed if the type of \(e\) is a subtype of that of \(\text{le}\). The condition \(\text{sdisjoint le}\) requires that, if \(\text{le}\) is a list, then no member of \(\text{le}\) is a prefix of another one after deleting \(\text{self}\) and type casts, in order to avoid aliasing. For instance, neither \(x, x.y\) nor \(x, \text{self}.x\) is \(\text{sdisjoint}\), but \(x, y.x\) is. For a result parameter, the parameterised command can only be applied to left-expressions. An application of \((\text{res} x : T \bullet c)\) to a result argument \(\text{le}\) is well-typed if \(T\) is a subtype of the type of \(\text{le}\). The command \(\text{le}.m(\text{e})\) can potentially modify \(\text{le}\). So, it is required that \(\text{le}\) and the arguments passed by result are \(\text{sdisjoint}\).
avoiding aliasing. The types of the list \( e \) of arguments must satisfy the condition \( \text{aptype} \, \Gamma \, \text{pds} \, T \), which enforces these types to be adequate with respect to the parameter passing mechanisms that are used. The constraints imposed by \( \text{aptype} \) are similar to those that appear in the typing rules for explicit parameterised commands.

**Programs**

The typing rule for a program is shown in what follows. The typing \((\emptyset; \ x : \ T) \triangleright c : \text{program}\) asserts that the program \( c \) is well-typed in a typing environment in which only global variables \( x \) are in scope. The free variables of a program are those used for input and output. In \((\emptyset; \ x : \ T)\) information about the global variables \( x : \ T \) is recorded in the \textit{locals} field of the typing environment.

The valid environment \( \Gamma \), in which the command \( c \) is required to be well-typed, is determined by the sequence of class declarations \( \textit{cds} \) and variable \( x \). The environment \( \Gamma \) contains information about the class \textit{object}. The function \( V\text{Decs} \) extracts information from the sequence of class declarations \( \textit{cds} \). It also checks its validity, as explained in what follows. The condition \( V\text{Meth} \, \Gamma \, \textit{cds} \) checks that the method bodies in the sequence of class declaration \( \textit{cds} \) are well-typed in the environment \( \Gamma \). The conditions \( \ll_{\textit{cds}} \) is well-founded and \( \textit{noredandrec} \, \Gamma \, \textit{cds} \) verify that mutually recursive calls and recursive definitions for methods with more than one definition, respectively, do not exist in the sequence \( \textit{cds} \).

\[
\begin{align*}
\Gamma, \text{main} \triangleright c : \text{com} \quad & \quad \Gamma = ((V\text{Decs} \, \textit{cds} \, \text{main}); \ x : \ T) \\
V\text{Meth} \, \Gamma \, \textit{cds} \quad & \quad \ll_{\textit{cds}} \ \text{is well-founded} \quad \textit{noredandrec} \, \Gamma \, \textit{cds} \\
\hline
(\emptyset; \ x : \ T) \triangleright \textit{cds} \bullet \ c : \text{program}
\end{align*}
\]

The function \( V\text{Decs} \) is defined based on the function \( D\text{ecs} \), which extracts the typing and visibility information about attributes and methods, and the inheritance relationship from a sequence of class declarations. The application of \( D\text{ecs} \) to a sequence of class declaration \( \textit{cds} \), like a typing environment, is a record with four fields: \textit{attr}, \textit{meth}, \textit{vis}, and \textit{supcls}. The function \( D\text{ecs} \) is total and it does not guarantee that there are no inappropriate or missing declarations in the sequence of class declarations. If the sequence of class declarations does not contain invalid declarations, and \( N \) is either \textit{main} or a class declared in \( \textit{cds} \), then \( V\text{Decs} \, \textit{cds} \, \textit{N} \) yields the typing environment for \( \textit{cds} \) which is extracted by the function \( D\text{ecs} \). The \textit{locals} field of this environment contains the visible attributes of \( N \), if it is not \textit{main}; it is empty if \( N \) is \textit{main}. The domain of \( V\text{Decs} \, \textit{cds} \, \textit{N} \) is characterised by some conditions. First, \textit{main} is not declared in \( \textit{cds} \). Secondly, there is no redeclarations (hiding) of an attribute in a subclass. Methods can be overridden in a subclass, but its signature has to be maintained. Finally, the class \( N \) has to be declared in \( \textit{cds} \) or it is \textit{main}.

The typing rule for programs also restricts method declarations. The restrictions are described by the function \( V\text{Meth} \, \Gamma \, \textit{cds} \): the method bodies in the sequence of class declarations \( \textit{cds} \) are well-typed in the typing environment \( \Gamma \). The body of a method \( m \) is checked in a context where
3.3. SEMANTICS

the locals are the visible attributes of \( N \), the class in which the method is declared.

The relation \( \ll \) associates a method \( m \) of class \( N \), represented by \((N, m)\), with a method \( m' \) of class \( N' \), written as \((N, m) \ll_{cds} (N', m')\). We write \((N, m) \ll_{cds} (N', m')\) when in the sequence of class declarations \( cdfs \), the body of \( m \) of class \( N \) can call the method \( m' \) in class \( N' \). If \( \ll_{cds} \) is well-founded, there is no mutual recursion in \( cdfs \). In other words, \( \ll_{cds} \) has no circularities. For instance, it excludes the relation \( \{(D, m_2) \mapsto (C, m_1), (D, m_2) \mapsto (C', m_1), (C', m_1) \mapsto (D, m_2)\} \), in which there is mutual recursion involving method \( m_1 \) and \( m_2 \) of class \( C \) and \( D \), respectively.

The \( \ll \) relation does not capture situations in which recursion is introduced in the same chain of inheritance by means of recursive calls; it only looks for calls to methods, in the same chain of inheritance, that have distinct names. The condition \( \text{noredandrec} \) \( \Gamma \) \( cdfs \) establishes that if a method \((N_1, m)\) contains a call to \((N_2, m)\), then either \( N_1 \) and \( N_2 \) are not in the same chain of inheritance (they are not superclasses of one another) or there is no redefinition of \( m \) in a proper superclass or subclass of \( N_1 \). For instance, for classes \( C \) and \( C' \), such that \( C' \leq_{cds} C \), and a method \( m \) declared by \( C \) and redefined by \( C' \) the relation \( \{((C, m) \mapsto \{(C, m), (C', m)\}), (C', m) \mapsto \emptyset\} \) does not satisfy the condition \( \text{noredandrec} \), as \((C, m)\) is included in the set related to \((C, m)\) itself, and \( C \) and \( C' \) are paired to \( m \) in the domain. So, there is a redefinition of \( m \), which is originally recursive. Also, for the relation \( \{((C, m) \mapsto \emptyset), (C', m') \mapsto \{(C, m), (C', m)\}\} \), in which \( C \) is a superclass of \( C' \), the condition \( \text{noredandrec} \) is not satisfied because \((C', m)\) is associated to \((C', m)\) and \((C, m)\) is in the domain.

3.3 Semantics

The semantic functions, denoted by \([\cdot]\), are defined for each derivable typing, except method call, by induction on the typing rules. Some of the constructs are treated by means of syntactic transformations; and the semantics of method calls goes beyond structural recursion on typing derivations; it is discussed later. In the semantics definition proposed in [22], there are assignments to self, which is not in the grammar of ROOL as proposed in Section 3.1. Also, semantic definitions presented in [22] use a value-result parameter passing mechanism (vres), which is not present in the syntax of ROOL as we present here. These constructs play a central role in the semantics, but should not really appear in user programs. Since the emphasis of our work is on laws of program transformation, we left out of the grammar the constructs we do not need to consider in our laws.

3.3.1 Environments and states

Most semantic functions take an environment \( \eta \) as argument. An environment is a finite partial function that associates a class name to its meaning \((CName \mapsto CMeaning)\). The meaning of a class is described by a function that associates method names to parameterised commands \((LName \mapsto PCom)\). The parameterised commands have an extra parameter \( me \), passed by value-result, which provides the attributes of the object upon which the method is called.
The set of environments that are compatible with a given typing environment $\Gamma$ is denoted by $[\Gamma]$. The environments $\eta$ in this set have the following characteristics. First, all declared classes in $\Gamma$ have a meaning in $\eta$ ($\text{dom}\ \eta = \Gamma.cnames$). Secondly, the methods that appear in $\eta$ associated with a class $N$ are exactly the methods declared in $\Gamma.meth$, including the inherited ones. Formally, we have $\forall N : \text{dom}\ \eta \bullet \text{dom}(\eta N) = \text{dom}(\Gamma.meth N)$. Lastly, the parameter declarations for these methods in $\eta$ are those recorded in the $\text{meth}$ field of the typing environment $(\Gamma.meth)$ together with the extra value-result parameter $me$.

$$\forall N : \text{dom}\ \eta; \ m : \text{dom}(\eta N) \bullet \exists c : \text{Com} \bullet \eta N m = (\text{vres me : N}; \ \Gamma.meth N m \bullet c)$$

The command $c$ is derived as a fixpoint in the environments we construct later.

A state records the values of the attributes of the current class, and the values of the parameters and local variables. In other words, it associates values to the elements of $LName$. The state also records the class of the current object. Object values associate values to attribute names. In both cases, the values must be type-correct.

### 3.3.2 Extended typing

The restrictions imposed by typing rules are appropriate for user programs, but not for the semantics. Some of these constraints are relaxed in the definition of what we call an extended typing system; it can be characterised by making three changes to the rules presented in Section 3.2.2. The first change drops the constraints involving the predicate $\text{visib}$. With this change, attributes of the parameter $me$ become visible in the context of a method call.

The second change drops the subtyping constraint in typing rules for type tests and type casts. For instance, $e$ is $N''$ requires that $e$ has type $N'$ with $N'' \leq \Gamma N'$; this condition is dropped. From the semantic point of view, $e$ is $N''$ can only convey useful information if $N''$ is a subtype of the declared type of $e$. This constraint, however, is incompatible with the semantics of assignment. Consider, for example, that we have a class $A$ extended by $B$, which, in turn, is extended by $C$. Suppose we have variables $a : A$, $b : B$, $c : C$ in $\Gamma.locals$. In this way, the command $a := c$, and the predicate $a$ is $B$ are well-typed. The weakest precondition of $a := c$ with respect to $a$ is $B$, however, involves the substitution $(a \text{ is } B)[c/a]$, resulting in the predicate $c$ is $B$. The original typing system disallows this predicate, because the type of $c$ is not a supertype of $B$. We need typability to be preserved by substitution on formulas, and this only holds in the extended type system.

We also add a condition on the rules for $\text{le}$.m, $(\text{res x : T} \bullet c)(le)$, and $(\text{vres x : T} \bullet c)(le)$. This condition restricts $le$ only to casts of subclasses of the declared types, so that they still satisfy the subtyping conditions in rules for casts of the original typing system. This is the third change to the extended typing system.

In the next section, we present the semantics for commands and parameterised commands.
### 3.3. SEMANTICS

#### 3.3.3 Commands and parameterised commands

The semantic equations for commands take an environment \( \eta \) as argument. The semantics of a command \( [\Gamma, N \triangleright c : \text{com}] \) \( \eta \) is given by a function from formulas to formulas: a predicate transformer representing weakest preconditions.

The semantics of assignment to simple variables are interpreted as substitutions on formulas.

\[
\Gamma \triangleright x : T \quad \Gamma \triangleright e : T' \quad T' \leq_{T} T
\]

\[
[\Gamma \triangleright x := e : \text{com}] \eta \psi = (e \neq \text{error} \land \psi[e/x])
\]

Assignment to general left-expressions are manipulated by means of syntactic transformations, yielding assignments of update expressions to simple variables and to \texttt{self}.

Based on the following semantic definition, which refers to the relation \( \rightarrow \) that appears in Table 3.7, syntactic transformations can be used to simplify parameter passing and assignments involving general left-expressions.

\[
c \rightarrow^{*} c' \quad [\Gamma, N \triangleright c' : \text{com}] \eta = g
\]

\[
[\Gamma, N \triangleright c : \text{com}] \eta = g
\]

The transitive closure \( \rightarrow^{*} \) reduces a command that has a derivable typing to one that has a direct semantic definition.

The transformations that deal with parameter passing are the first five ones in Table 3.7. The parameter passing mechanism \texttt{vres} corresponds to the traditional convention of call-by-value-result. \( FV \; c \) stands for the set of free variables of \( c \), similarly for commands \( c \). We use \( \alpha(pd) \) to denote the variables declared by \( pd \). The last eight transformations deal with assignment to left-expressions,
rewriting them into assignments to simple variables and self, using update expressions. The function upd, when applied to a left-expression le, to a list of attributes \( \mathcal{X} \), and to an expression \( e \), results in an object copied from le but with each attribute in \( \mathcal{X} \) changed to take the corresponding value in the list \( e \).

In the case of syntactic transformations for parameter passing, assignments to self occur when the argument corresponding to the me parameter is self. In the semantics for method calls, self is always assigned an object of the current class, so that the target and the source of the assignment have the same type.

The weakest precondition that guarantees that self := e establishes \( \psi \) is a disjunction over the subclasses \( N'' \) of the type \( N' \) of e. The test e isExactly \( N \) holds when the value of e is an object of class \( N \), but not of any of its subclasses. Each disjunct requires \( \psi \) to hold when self takes the value e, and the attributes \( x \) of \( N'' \) take the value e.x. There is no need to check that e is not error because error isExactly \( N'' \) is false, for all \( N'' \). If self is assigned an object of the current class, the semantics simplifies to e \( \neq \) error \( \land \psi[e, e.x/self, x] \).

The semantics of specification statements is similar to Morgan’s definition:

\[
\begin{align*}
[\Gamma; x : T \triangleright x : \{\psi_1, \psi_2\}] : \text{com} \eta \psi &= \psi_1 \land (\forall x : T \bullet \psi_2 \Rightarrow \psi)
\end{align*}
\]

The semantics of control constructs and blocks is standard. They appear in Table 3.8.
3.3. SEMANTICS

3.3.4 Programs and method calls

The meaning of a complete program is the meaning of its main command in a proper typing environment $\Gamma$ that includes the global variables $x$ of the command $c$. The typing environment $\Gamma$ is extracted by the function $VDecs$.

\[
[\Gamma, \text{main} \triangleright c : \text{com}]\eta = f \quad \Gamma = ((VDecs \, cds \, \text{main}); \, x : T)
\]

\[
\eta = \text{Meths} \, \Gamma \, cds
\]

\[
[\emptyset; \, x : T \triangleright cds \, \bullet \, c : \text{program}] = f
\]

The environment $\eta$ records the methods available for the object of the classes declared in $cds$. This environment is extracted from the sequence of class declarations $cds$ by the function $\text{Meths}$.

The semantics of method calls cannot rely on the copy rule due to dynamic binding. To define the semantics of method call, we need to consider all methods that could be called; an environment is used to record the meaning of these methods.

We assume there is an enumeration $((N_i, m_i))$, with $i \geq 1$, of methods declared in a program $cds \, \bullet \, c$. In the sequence $(N_i, m_i)$, a method is listed after all those it can call. If a method has several definitions, then a definition in a superclass is listed before one in a subclass. The meaning of a method $(N_i, m_i)$ in this sequence depends only on the meaning of methods $(N_j, m_j)$, with $j < i$.

We define $\text{Meths} \, \Gamma \, cds$ to be the last of a sequence of environments $\eta_i$. The first environment is $\eta_0$; for all classes $N$ in $\Gamma.cnames \setminus \{\text{object}\}$ and $m$ in $\text{dom}(\Gamma.meth \, N)$, it is defined as follows.

\[
\eta_0 \, N \, m = (\text{vres} \, me : N; \, pds \, \bullet \, \text{abort}), \quad \text{where} \quad pds = \Gamma.meth \, N \, m
\]

We also define $\eta_0 \, \text{object} = \emptyset$. In order to define $\eta_i$ from $\eta_{i-1}$, we establish the semantics of the method $(N_i, m_i)$ to be associated to $(N, m_i)$, for all $N \leq_F N_i$. A method $(N, m_i)$ is available in $N_i$ and in all subclasses of $N_i$. A redefinition of $m_i$ in class $N$, a subclass of $N_i$, occurs in a position $j$ of the sequence of methods, with $j > i$, and the meaning of $m_i$ is recorded in $\eta_j \, N \, m_i$. The definition for these is the following.

\[
\eta_i \, N \equiv \eta_{i-1} \, N \oplus \{m_i \mapsto \mu_{m_i}^{(i-1)}(\text{vres} \, me : N; \, pds \, \bullet \, meI \, \Gamma' \, N \, m \, c)\}
\]

The parameterised command $(pds \, \bullet \, c)$ is the body of $(N_i, m_i)$, and $\oplus$ denotes the function overriding operator. The $\text{locals}$ field of the typing environment $\Gamma'$, differently from that of $\Gamma$, includes the attributes of $N$, the parameter $me$, and the variables declared by $pds$. Accesses to attributes of $N$ or calls to methods of $N$ in $c$ need to be made through $me$. Recursive calls receive the adequate extra argument. These changes are performed by the function $meI$ [22]. For instance, if $x$ and $m$ are an attribute and a method of $N$, respectively, after the application of $meI$ they become $me.x$ and $me.m$. Only free occurrences of attributes of $N$ are changed by $meI$; occurrences of $\text{self}$ are replaced with $me$. 
The semantics of the explicit recursion construct uses fixpoints.

\[
\begin{align*}
\llbracket \Gamma \vdash \mu_Y c : \text{com} \rrbracket \eta &= f \\
\llbracket \Gamma \vdash \text{rec } Y \bullet c \text{ end} : \text{com} \rrbracket \eta &= f
\end{align*}
\]

Similarly to a method body, the body of a recursion construct is regarded as a context. The context here is a function from commands to commands; \( \mu_Y \) is the least fixpoint of \( c \), as a function of \( Y \).

The lattice for this fixpoint is that of monotonic predicate transformers in the state space \( [\Gamma, N] \), where \( N \) is the current class.

We present the semantics of method calls \( le.m(e) \). The method to be applied is given by the environment \( \eta \), according to the dynamic type of \( le \) in a particular state. For all subclasses \( N' \) of the static type \( N'' \) of \( le \), the parameterised command \( \eta N' m \) has the form \( (\text{vres } me : N'; \text{pds } \bullet c) \). We define \( f_{N'} \) to be \( [\Gamma, N \triangleright (\text{vres } me : N'; \text{pds } \bullet c)((N')le, e) : \text{com}] \eta \). The cast \((N')le\) is necessary because the \( me \) parameter of \( \eta N' m \) is passed by value-result and has type \( N' \). We define \( [\Gamma, N \triangleright le.m(e) : \text{com}] \eta \psi \) to be \( f_{N'} \psi \). Thus, the semantics of method calls is the disjunction of \( (le \text{ isExactly } N' \land f_{N'} \psi) \), over the possible classes \( N' \).

\[
[\Gamma, N \triangleright (\eta N' m)((N')le, e) : \text{com}] \eta = f_{N'} \text{ for all } N' \leq \Gamma, N'' \text{ the type of } le
\]

The predicate is false in states where \( le \) is \text{null} as \text{null isExactly } N' = false. In order for this equation be satisfied \( (\eta N' m) \) must be typable in \( \Gamma, N \). Attributes in the body of \( m \) are accessed by the \( me \) parameter. These attributes are not necessarily visible in the context of a method call, thus references to them are only typable in the extended typing system.

### 3.4 Refinement

The objective of refining a program is to derive an efficient and executable program from an initial specification. In rool, this initial specification can range from a specification statement—as in traditional refinement calculi—with an empty sequence of class declarations, to a program, with an arbitrary number of class declarations. Even if we begin the refinement with a specification statement, typically the objective is to obtain a program that possibly uses classes. So, we need to establish a refinement relationship between programs.

We refine a program by refining its command part and its class declarations.

**Definition 1** For sequences of class declarations \( cds \) and \( cds' \), commands \( c \) and \( c' \) with the same free variables \( x : T \), define \( (cds \bullet c) \sqsubseteq (cds' \bullet c') \) if and only if, for all \( \psi \),

\[
[\emptyset, x : T \triangleright (cds \bullet c) : \text{program}] \psi \Rightarrow [\emptyset, x : T \triangleright (cds' \bullet c') : \text{program}] \psi
\]
This is the standard definition for refinement in the context of a weakest precondition semantics. The refinement of commands has already been presented by Cavalcanti and Naumann \cite{22}; they proved that the command constructors of ROOL are monotonic with respect to their constituents commands (Theorem 3 in \cite{22}).

The refinement of a class is related to data refinement because a class is a data type. We must consider the refinement of a class declaration \(cd_1\) by another class declaration \(cd_2\) in a context of a sequence of class declarations \(cds\). Class names play an important role in object-oriented programs. For instance, classes with distinct names describe different types even if they have the same attributes and methods.

If \(cd_1\) and \(cd_2\) declare classes with the same name, \(cd_1\) is refined by \(cd_2\) in the context of \(cds\), if and only if, \(cds\ cd_1\) and \(cds\ cd_2\) are well formed, and for all commands \(c\), we have that \((cds\ cd_1 \bullet c) \subseteq (cds\ cd_2 \bullet c)\).

On the other hand, if \(cd_1\) declare a class with a distinct name from those present in \(cds\), the declaration of \(cd_1\) can appear in the same program representing a different design that implements the same functionality. In this situation, we have that \((cds \bullet c) \subseteq (cds\ cd_1 \bullet c)\).

**Theorem 1** The command constructors are monotonic with respect to refinement of their constituent commands.

The proof of this theorem is based on two lemmas. The first lemma (Lemma 15 in \cite{22}) establishes refinement between parameterised commands; the second lemma (Lemma 16 in \cite{22}) establishes that the weakest preconditions defined are monotonic predicate transformers.

### 3.5 Laws

In this section we present some of the laws of ROOL. The whole set of laws can be found in Appendices D and E. These laws were firstly presented in \cite{14,15,33}. We have proved all the laws correct against the weakest precondition semantics of ROOL. The proofs can be found in Appendices F and G. We also need to apply class refinement, which is a notion related to data refinement. This is achieved by the application of the simulation laws presented in the next section.

In the laws we write ‘\((\rightarrow)\)’ when some conditions must be satisfied for the application of the law from left to right. We also use ‘\((\leftarrow)\)’ to indicate the conditions that are necessary for applying a law from right to left. We use ‘\((\leftrightarrow)\)’ to indicate conditions necessary in both directions.

#### 3.5.1 Simulation Laws

Simulation links concrete and abstract states, allowing us to change the representation from abstract to concrete. A formula, known as coupling invariant, relates existing (abstract) variables to new (concrete) ones.
Traditional techniques of data refinement deal with modules that encapsulate variables. This is the case when we deal with private attributes of a class. Law \(\langle\text{private attribute-coupling invariant}\rangle\) in what follows allows us to change private attributes in a class, relating them with new attributes. The application of this law changes the bodies of the methods declared in the class. The changes follow the traditional laws for data refinement \[64\]. Some of these laws are presented in Appendix D. By convention, the attributes denoted by \(x\) are abstract, whereas those denoted by \(y\) are concrete. The coupling invariant \(CI\) relates abstract and concrete attributes. The notation \(CI(mts)\) indicates the application of \(CI\) to each of the methods in \(mts\): applying \(CI\) changes the methods according to the laws of data refinement \[64\], that is, guards are augmented in order to assume the coupling invariant and every command is extended by modifications to the concrete variables so that they maintain the coupling invariant. We write \(\text{pri}\ a : T;\ ads\) to denote the attribute declaration \(\text{pri}\ a : T\) and all declarations in \(ads\); whereas \(mts\) stands for declarations of methods and initialisers. The symbol \(\preceq\) indicates that this law involves a simulation between attributes that are related by the coupling invariant \(CI\).

**Law \(\langle\text{private attribute-coupling invariant}\rangle\)**

\[
\begin{align*}
\text{class } A & \text{ extends } C \\
\text{pri } x : T; & \text{ ads} \\
mts & \\
\text{end} & \\
cds & \bullet c
\end{align*}
\[
\begin{align*}
\text{class } A & \text{ extends } C \\
\text{pri } y : T'; & \text{ ads} \\
CI(mts) & \\
\text{end} & \\
cds & \bullet c
\end{align*}
\]

Law \(\langle\text{superclass attribute-coupling invariant}\rangle\) below allows us to change attributes in a class, relating them with already existing attributes by means of a coupling invariant. The application of this law changes the bodies of the methods declared in the class and in its subclasses. Similarly to the previous law, the changes follow the traditional laws for data refinement \[64\]. The notation \(CI(cds')\) indicates that \(CI\) acts on the class declarations of \(cds'\). The application of \(CI\) to a class declaration changes the methods of such a class according to the laws of data refinement \[64\], as in the previous law. In order to apply this law the coupling invariant \(CI\) must refer only to public and protected attributes in \(adsA\), since it is used in the subclasses of \(A\).

**Law \(\langle\text{superclass attribute-coupling invariant}\rangle\)**

\[
\begin{align*}
\text{class } A & \text{ extends } C \\
adsA & \\
mts & \\
\text{end} & \\
cds' & cds
\end{align*}
\[
\begin{align*}
\text{class } A & \text{ extends } C \\
\text{adsC} & \\
CI(mts) & \\
\text{end} & \\
CI(cds') & cds
\end{align*}
\]

\(\square\)
3.5. LAWS

CI refers only to public and protected attributes in \( adsA \);
\( cds' \) only contains subclasses of \( A \)
\( cds \) contains no subclasses of \( A \)

While law \(<\text{private attribute-coupling invariant}>\) is a standard simulation law for modules, law \(<\text{superclass attribute-coupling invariant}>\) is an original contribution of a joint work presented in \([17]\) \([16]\). It is worth emphasising that the simulation relations are sound, which means that, for well-formed contexts \( cds \cdot c \), if \( cds_1 \preceq_{cds \cdot c, CI} cds_2 \), then \( (cds \cdot cds_1 \cdot c) \subseteq (cds \cdot cds_2 \cdot c) \). This has been proved in \([25]\) for law \(<\text{private attribute-coupling invariant}>\).

The law \(<\text{Data refinement—variable blocks with initialisation}>\), in what follows, is a data refinement law for variable blocks. It considers variable blocks whose bodies start with an initialisation of the abstract variables.

**Law** \(<\text{Data refinement—variable blocks with initialisation}>\)

\[
\text{var } vl; \text{ avl} \cdot \text{avl} : [\text{true, init}]; c_1 \text{ end}
\]

\[\subseteq\]

\[
\text{var } vl; \text{ cvl} \cdot \text{cul} : [\text{true, } (\exists \text{ avl} \cdot CI \land \text{init})]; c_2 \text{ end}
\]

provided

\( c_1 \preceq c_2 \)

The variables of \( cvl \) are not free in \( \text{init} \) and \( c_1 \), and are not in \( avl \);

The variables of \( avl \) are not free in \( c_2 \);

The command \( c_2 \) is a data refinement of command \( c_1 \) and can be calculated from the original command \( c_1 \), the abstract variables \( avl \), the concrete variables \( cvl \), and the coupling invariant \( CI \) using data refinement laws \([64]\). The law \(<\text{Data refinement—variable blocks with initialisation}>\) is based on a similar law presented in \([20]\).

The laws presented in this section allows us to change the representation of attributes and variables. This is useful, for instance, to relate attributes of a class with attributes of another class.

3.5.2 Laws of Commands

In this section we present the programming laws of ROOL that apply to the commands of the language, which are regarded as small grain constructs. The whole set of laws for commands can
be found in Appendix D.

Assigning a left expression to itself leaves the state of a program unchanged, so it is equivalent to \texttt{skip}. We require the left expression not to be \texttt{error}, otherwise the program aborts.

\textbf{Law} \(\langle := \texttt{skip} \rangle\) If \(le \neq \texttt{error}\), then \((le := le) = \texttt{skip}\)

Branches of an alternation can be eliminated if their guards are always false.

\textbf{Law} \(\langle \texttt{if elim} \rangle\) If \(i\) ranges over \(1..n\) and \(\neg (\psi \land \psi_i)\), for all \(i\), then

\[
\text{if } \psi \rightarrow \left( \begin{array}{c}
\text{if } \psi \rightarrow c \\
\text{fi}
\end{array} \right) \left( [i \cdot \psi_i \rightarrow c_i] \right) \begin{array}{c}
\text{fi} \\
[ j \cdot \psi_j \rightarrow c_j]
\end{array}
\right) =
\text{if } \psi \rightarrow c \begin{array}{c}
\text{fi}
\end{array} \left( [j \cdot \psi_j \rightarrow c_j] \right)
\]

If \texttt{miracle} precedes another command, such a composition behaves as \texttt{miracle}, in other words, \texttt{miracle} is a left zero of sequential composition. The same is true for \texttt{abort}.

\textbf{Law} \(\langle ; - \texttt{miracle \ left \ zero} \rangle\) \texttt{miracle; c} = \texttt{miracle}

The sequential composition of two assignments can be combined into just one assignment. The notation \(e_2[e_1/le]\) stands for the replacement of every free occurrence of \(le\) in \(e_2\) with \(e_1\).

\textbf{Law} \(\langle ; - := \texttt{combination} \rangle\) \((le := e_1; le := e_2) = (le := e_2[e_1/le])\)

We can move a variable declaration which is present in a branch of an alternation to outside the alternation, provided the variable does not occur free in the guards.

\textbf{Law} \(\langle \texttt{var-; if dist} \rangle\) If \(i\) ranges over \(1..n\) and \(x\) is not free in \(\psi_i\), then

\[
\text{if } [i \cdot \psi_i \rightarrow (\texttt{var} x : T \cdot c_i \texttt{end}) \texttt{fi} = \texttt{var} x : T \cdot \texttt{if} [i \cdot \psi_i \rightarrow c_i \texttt{fi end}}
\]

The relation between angelic and demonic variable blocks is stated in the following law. The angelic variable block avoids abortion by choosing a value to \(x\) equal to that of \(e\). The local variable block avoids a miracle by choosing a value to \(x\) equal to that of \(e\).
Law (avar - var relationship) If $x$ is not free in $e$, then
\[
\text{avar } x : T \cdot \{ x = e \}; \ c \ \text{end} = \text{var } x : T \cdot \{ x = e \}; \ c \ \text{end}
\]

A very useful command law in the context of object-oriented programs deals with the elimination of method calls. When there are no redefinitions or visibility concerns, the elimination of a method call can be characterised by the copy rule.

Law (method call elimination)

Consider that the following class declaration

\[
\text{class } C \text{ extends } D
\]
\[
\text{ads}
\]
\[
\text{meth } m \triangleq pc
\]
\[
\text{mts}
\]
\[
\text{end}
\]

is included in $cds$. Then

\[
\text{cds}, A \triangleright le.m(e) = \{ le \neq \text{null} \land le \neq \text{error} \}; \ pc[le/self](e)
\]

provided

\[(\leftrightarrow) \ m \text{ is not redefined in } cds \text{ and does not contain references to super; and all attributes which appear in the body } pc \text{ of } m \text{ are public.}\]

Using the above law, we can replace a call $le.m$ with the body of the method $m$, which is defined by the parameterised command $pc$, substituting every occurrence of self for $le$. If a private attribute of class $C$ appears in the body of $m$, this substitution would cause a compilation error due to visibility concerns. This is the reason for requiring the attributes to be public. We also require that super does not appear inside $m$ because, after class elimination, the parameterised command we obtain is located inside a different class from the one that declares the method. These classes may have different superclasses leading to a compilation error or to a behaviour distinct from that of the method call. A method call $le.m(e)$ aborts when $le$ is null or error. On the right-hand side, we have the assumption $\{ le \neq \text{null} \land le \neq \text{error} \}$, which also aborts when $le$ is null or error. This assumption behaves like skip when $le$ is not null nor error.

The law (command refinement-class refinement) in what follows states that the refinement of a command in a class leads to the refinement of such a class as a whole. This can be expressed by the following law.
3.5.3 Laws of Classes

The following laws apply to classes, but they are still basic in the sense that more elaborate laws, which can be applied to practical program transformation, need to be derived. Here we use the notation \( cd_1 =_{cds,c} cd_2 \) as an abbreviation to \( (cds\ cd_1 \bullet c) = (cds\ cd_2 \bullet c) \), meaning that the class declarations \( cd_1 \) and \( cd_2 \) are equivalent in the context of the sequence of class declarations \( cds \) and main command \( c \). The whole set of laws for classes can be found in Appendix E.

The notation \( B.a \), which appears in provisos, denotes the use of the name \( a \) by means of expressions of type \( B \), strictly. The subclass relation is denoted by \( \leq \); it is an abbreviation for \( \leq_{\Gamma} \), where \( \Gamma \) is the typing environment corresponding to the relevant sequence of class declarations.

The following law allows changing the visibility of attributes from private to public. Applying this law from left to right makes a private attribute public; in this direction there are no provisos. For applying this law from right to left, the attribute cannot be used anywhere outside the class where it is declared; this is required by the proviso.

\[
\text{Law (change visibility: from private to public)}
\]

\[
\begin{array}{c}
\text{class } C \text{ extends } D \\
\text{pri } a : T; \ ads; \\
mts \end{array} =_{cds,c} \begin{array}{c}
\text{class } C \text{ extends } D \\
\text{pub } a : T; \ ads; \\
mts \end{array}
\]

provided

\( (\leftarrow) \ B.a, \text{ for any } B \leq C, \text{ does not appear in } cds \text{ or } c. \)

Using the following law, we can remove a method from a class. In order to remove a method from a class, it cannot be called by any class in \( cds \), in the main command \( c \), nor inside class \( C \).
3.6. CONCLUSIONS

For applying this law from right to left, the method $m$ cannot be already declared in $C$ nor in any of its superclasses or subclasses. In other words, we can always introduce a new method in a class.

$$\textbf{Law (method elimination)}$$

\[
\begin{array}{l}
\text{class } C \text{ extends } D \\
\quad \text{ads}; \\
\quad \text{meth } m \triangleq (pds \bullet c) \; mts \\
\end{array}
\quad \Rightarrow_{cds,c}
\begin{array}{l}
\text{class } C \text{ extends } D \\
\quad \text{ads}; \\
\quad \text{mts} \\
\end{array}
\]

provided

$(\rightarrow)$ $B.m$ does not appear in $cds$, $c$ nor in $mts$, for any $B$ such that $B \leq C$;

$(\leftarrow)$ $m$ is not declared in $mts$ nor in any superclass or subclass of $C$ in $cds$.

\[\square\]

As already said, the whole set of laws can be found in Appendices [D] and [E].

3.6 Conclusions

In this chapter we presented the language ROOL. Besides the usual commands expected in object-oriented languages, ROOL includes specification statements as commands, supporting reasoning about object-oriented programs. The semantics of ROOL is based on weakest preconditions. We discussed the context in which program refinement holds. Finally, we presented some programming laws of ROOL.
Chapter 4

Compositional Refactorings

Refactoring is the process of restructuring a program without changing its behaviour [69, 42]. In practice, the verification that the behaviour of a program is not changed after refactoring relies on looking for type errors detected by means of compilation and also on running test suites. The word refactoring is also used to mean a change that is made to a program.

In this chapter we present a list of refactoring rules written in ROOL. These rules are inspired by the work presented by Fowler [42]. Strictly, some of them are not the same refactorings presented by Fowler; differences between them are discussed along the presentation. The (copy) semantics of ROOL is the main cause of differences between our refactorings and Fowler’s ones. As a consequence, in our refactorings, we restrict type of parameters to be basic. The refactorings we present here cover most of the refactorings presented by Opdyke in his pioneering work [69], whose focus is the automated refactoring of C++ programs. Indeed, most of the low-level refactorings presented by Opdyke are captured by the laws of commands and classes of ROOL that appear in Chapter 3 and Appendices D and E. For instance, the refactoring create_member_variable presented by Opdyke [69, p. 56] is equivalent to the Law 83 (attribute elimination) of ROOL, when applied from right to left. More elaborate refactorings proposed by Opdyke are also presented by Fowler, like the one for adding a function argument.

In this chapter we present refactoring rules whose application does not affect the context of the classes being refactored. In other words, the fact that these classes are being modified does not affect classes other than those to which we apply the refactoring rule. Also, the application of the refactoring rule does not affect the main program. In this sense, we say that these refactorings are compositional, even though we may impose restrictions (side-conditions) on classes that are not being refactored and on the main program.

We present refactorings that deal mainly with methods and attributes. These basic refactorings can be used in the derivation of others. We exemplify this by the proof of a refactoring that allows the extraction of classes. Refactorings that deal with commands are also presented in this chapter, but their proofs are relegated to Appendix A.
CHAPTER 4. COMPOSITIONAL REFACTORINGS

4.1 Notation

The refactoring rules are described by means of two boxes written side by side, along with where and provided clauses. We use the where clause to write abbreviations. The provisos for applying a refactoring rule are listed in the provided clause of the rules. The left-hand side of the rule presents the class or classes before the rule application; the right-hand side presents the classes after the rule application: the transformed classes. We must note, however, that many of the refactoring rules are equalities and can be applied in both directions. The equivalence of sequences of class declarations $cds_{LHS}$ and $cds_{RHS}$—which appear, respectively, on the left-hand side and on the right-hand side of a rule—is denoted by $cds_{LHS} =_{cds,c} cds_{RHS}$. This is an abbreviation for the behavioural equivalence: $cds_{LHS} cdss \cdot c =_{cds,c} cds_{RHS} cdss \cdot c$. The refinement of a sequence of class declarations $cds_{LHS}$ by a sequence of class declarations $cds_{RHS}$ is denoted by $cds_{LHS} \sqsubseteq_{cds,c} cds_{RHS}$.

The notation $cds, N \triangleright c$ asserts that $c$ is a command that appears in the body of a method in class named $N$, whose declaration is in $cds$.

In class declarations, we write $\text{pri} \ a : T; \ ads$ to denote the attribute declaration $\text{pri} \ a : T$ and all additional declarations in $ads$. This declaration is similar for protected and public attributes. Also, $mts$ stands for declarations of methods. A method $m$ is defined by means of a parameterised commands in the form $(pds \bullet c)$. The parameters of $m$ are $pds$, and the names of such parameters inside $m$ is indicated by $\alpha(pds)$.

We also use the notation $c[c']$ to express that in the command $c$ there may be occurrences of the command $c'$; later occurrences of $c[c'']$ denote the command obtained by replacing the occurrences of $c'$ with $c''$. Similarly, the notation $c[exp]$ expresses that there might be occurrences of the expression $exp$ in the command $c$. In the same way, $exp_1[exp_2]$ expresses that $exp_2$ may occur in the expression $exp_1$. Also, we express that a command $c_1$ and an expression $e_1$ may occur inside a command $c$ by using the notation $c[c_1][e_1]$. Meta-variables with different names denote different values and variables.

The notation $B.m$ refers to uses of the name $m$ via expressions of type $B$. For instance, if we write that $B.m$ does not appear in $mts$, we mean that $mts$ contains no expressions such as $e.a$, for any $e$ of type $B$, strictly.

For the derivation of the refactoring rules, we assume that the conditions that appear in each rule are satisfied. Also, in the derivations, we write ‘(l/r)’ or ‘(r/l)’ after the name of a law to indicate that we apply the law from left to right or from right to left, respectively. If a law is applied more than once, we also indicate this with the number of times between parentheses, like in (2x), which indicates two applications of the law.
4.2. REFACTORING RULES

Rule 4.1 \textit{(Extract/Inline Method)}

\begin{align*}
\text{class } A \text{ extends } C & \quad \text{ads;} \\
\quad \text{meth } m_1 \triangleq (pds_1 \cdot c_1[c_2[a]]) & \quad \text{mts} \\
\text{end} & = c_{d_{s,c}}
\end{align*}

where

\begin{align*}
c'_1 & \triangleq c_1[\text{self}.m_2(a)/c_2[a]] \\
mts' & \triangleq mts'[c_2[a]/\text{self}.m_2(a)]
\end{align*}

\(a\) is the finite set of free variables of command \(c_2\), not including attributes of class \(A\);

provided

\begin{align*}
(\leftrightarrow) & \quad \text{Variables in } a \text{ have basic types;} \\
& \quad \text{Parameters in } pds_2 \text{ must have types the same types as those of variables in } a; \\
(\rightarrow) & \quad m_2 \text{ is not declared in } mts \text{ nor in any superclass or subclass of } A \text{ in } c_{d_{s,c}}; \\
(\leftarrow) & \quad m_2 \text{ is not recursively called; } \\
& \quad B.m_2 \text{ does not appear in } c_{d_{s,c}}, c \text{ nor in } mts, \text{ for any } B \text{ such that } B \leq A.
\end{align*}

4.2 Refactoring Rules

In this section we present some compositional refactoring rules. The first rule we present is used to extract or inline a method. We also present rules for moving methods and attributes. Another rule allows us to pull up or push down a method in a hierarchy. Finally, we present a rule for class extraction.

Each refactoring rule is presented along with an explanation of the rule itself and of its provisos. Then, we present its derivation.

4.2.1 Extract and Inline Method

Rule 4.1, when considered from left to right, coincides with the refactoring \textit{Extract Method} presented by Fowler [42] p. 110], whereas the application in the reverse direction corresponds to the refactoring \textit{Inline Method} [42] p. 117]. It turns a command \(c_2\), which is present in a method \(m_1\), into a new method \(m_2\). Occurrences of the command \(c_2\) in the original method \(m_1\) are replaced by calls to the newly introduced method.

The meta-variable \(a\) represents a list containing the free variables that appear in the command \(c_2\) of method \(m_1\) which are not attributes of class \(A\). On the left-hand side of this rule, \(c_1[c_2[a]]\) represents the command \(c_1\), which may have occurrences of the command \(c_2\), which in turn may
have occurrences of $a$. The class $C$ that appears in the `extends` clause, and in the others that follow, is present in the sequence of class declarations $cds$ or is the class `object`.

On the right-hand side of this rule, the method call `self.m2(a)` replaces the occurrences of the command $c_2[a]$ in the command $c_1$ of method $m_1$; the resulting command is $c'_1$. In the command $c_2$ of method $m_2$, the variables indicated by $a$ are replaced with the parameters $\alpha(pds)$. If a variable is only read in $c_2$, it could be passed as a value argument. A variable that is only written could be passed as result argument. Variables that are both read and written must be passed as both value and result arguments. The free variables, represented by $a$, that are passed as arguments in the call to $m_2$ may involve all arguments of method $m_1$ as well as local variables that appear in $c_2$. Of course, all the free variables that appear in $c_2$ must become parameters of $m_2$.

The method $m_2$ must be new: not declared in a superclass of $A$, in $A$ itself, nor in any of its subclasses. To apply this rule, we require the types of variables in $a$ to be basic. Also parameters in $pds_2$ must have the same types as those of $a$. Applying this rule from right to left replaces method calls to $m_2$ with the body of this method and removes $m_2$ from class $A$. To apply this rule in this direction, there must be no recursive calls in the method $m_2$ (note that this is not a limitation in practice because $c_2$ can be a recursive command of the form `rec X • c_3 end`). Also, the method $m_2$ cannot be called in $cds$, $c$ nor in $mts$.

The rule `Extract/Inline Method` improves the legibility and maintenance of a class. If a command is present in several methods of a class, this command can be extracted into a new method. The methods in which the command occurred can then just call this new method. Consequently, they become shorter than they were before the rule application; this improves legibility. Moreover, changing a command that appears in several methods can lead to inconsistency if one of the methods is not properly modified, whereas changing only one method is easier. This improves maintainability.

The rule `Extract/Inline Method` as we present here differs from the corresponding refactorings presented by Fowler [42], since we restrict the types of the variables passed as argument to be basic types. This limitation is a consequence of the laws used for introduction of parameterised commands from an arbitrary command. This is the only difference between our rule and Fowler’s one.

**Derivation.** We begin the derivation with the class $A$ that appears on the left-hand side. We assume that the required conditions for the application of this rule hold. However, it is not possible to have a fully general derivation, in the sense that we do not define a parameter for the method being extracted, rather we have to introduce a parameterised command along with a specific parameter passing mechanism in order to be able to conduct the proof. Moreover, in the derivation we consider that the set of variables $a$ has just one element, a variable named $a$. Also, we consider that such variable is only read, implying that the value parameter passing mechanism is the one applicable for the parameter to be defined in the extracted method. The derivation for a result
parameter is similar.

\[
\text{class } A \text{ extends } C
\]
\[
\text{ads;}
\]
\[
\text{meth } m_1 \triangleq (pds_1 \bullet c_1[c_2[a]])
\]
\[
\text{mts}
\]
\[
\text{end}
\]

We introduce a method named \( m_2 \) in class \( A \) by using the Law \( 90 \) (method elimination), from right to left. This requires that the method to be introduced is not declared in the superclass of the class to which the law is applied, in the class itself nor in any of its subclasses. As the refactoring rule requires the same conditions to be satisfied, we can apply Law \( 90 \) (method elimination).

\[
= \text{Law } 90 \text{ (method elimination) (r/l)}
\]
\[
\text{class } A \text{ extends } C
\]
\[
\text{ads;}
\]
\[
\text{meth } m_1 \triangleq (pds_1 \bullet c_1[c_2[a]]) \triangleleft
\]
\[
\text{meth } m_2 \triangleq (\text{val } \text{arg} : T \bullet c_2[\text{arg}] )
\]
\[
\text{mts}
\]
\[
\text{end}
\]

The command \( c_2 \) that is present in the method \( m_1 \) also appears in the method \( m_2 \). Our aim is to introduce a call to the method \( m_2 \). We introduce a parameterised command that is applied to the argument \( a \) by using Lemma \( 9 \) presented in Appendix \( C \). The occurrences of variable \( a \), in command \( c_2 \), are replaced with the parameter \( \text{arg} \). The symbol \( \triangleleft \) indicates the focus of our refinement. This lemma requires that the argument that is applied to the parameterised command is not a method call target. Consequently, there is no sense in having as argument an object on which we cannot call a method inside the parameterised command. For this reason we restricted \( a \) to have basic type.

\[
= \text{Lemma } 9
\]
\[
(\text{val } \text{arg} : T \bullet c_2[\text{arg}])(a)
\]
The class $A$ now is as follows.

```plaintext
class $A$ extends $C$

  ads;
  meth $m_1$ = $(pds_1 \cdot c_1[[\text{val} \ arg : T \cdot c_2[\text{arg}]](a)]) <$
  meth $m_2$ = $(\text{val} \ arg : T \cdot c_2[\text{arg}])

mts

end
```

The parameterised command that occurs in command $m_1$ is the same as that of method $m_2$. By using Lemma 1 (Appendix C), from left to right, we introduce in $m_1$ a call to $m_2$, obtaining the following class.

```plaintext
= Lemma 1

class $A$

  ads;
  meth $m_1$ = $(pds_1 \cdot c_1[[\text{self}.m_2(a)]])
  meth $m_2$ = $(\text{val} \ arg : T \cdot c_2[\text{arg}])

mts

end
```

This completes the derivation for a value parameter.

The derivation for an arbitrary number of parameters is similar, but it involves the application of Law 66 ($pcom\ merge$). Similarly to the inline method refactoring presented by Fowler [42, p. 117], we replace each call to the method we inline with the method body. Also, we remove the method definition.

The method `moreThanFiveLateDeliveries` is public and it is the target of refactoring `Inline Method` described by Fowler [42, p. 117]. However, there is no restrictions on calls to such method around the program. Maybe, this is consequence of the refactoring strategy that relies on program compilation for detecting errors such as calling a method that is not declared in the class of the target object. Similarly, the `Inline` refactoring implemented in the Eclipse IDE [37] does not give any alert to the user about calls to a public method that is target of this refactoring. Only in compilation, which is accomplished just after the refactoring application, errors are detected. During the refactoring application, there is no check that, when we inline a public method, it may lead to errors because such public method, for instance, may access attributes of the class in which it is declared. It is necessary to check this condition to allow a correct refactoring application.

### 4.2.2 Move Method

One of the most disseminated practices in the development of object-oriented programs is moving methods between classes. It is usually required when classes are highly coupled. Moving
methods between classes helps to make them simpler and clearer, improving legibility. The rule \(<\text{Move Method}\>\) (Rule 4.2) allows us to move a method that already indirectly reads and writes an attribute of the class to which the method is being moved.

On the left-hand side of this rule, the method \(m_1\) of \(A\) reads and writes to the attribute \(x\) of \(B\) by means of its get and set methods. The method calls occur inside the command \(c_1\) of the method \(m_1\) and have as target the attribute \(b\). The variable \(\text{aux}\) holds the result of the call to the method \(\text{getX}\) of \(B\). Such variable may occur on the right-hand side of an assignment to an expression \(le\). In the command \(c_1\), there may also be occurrences of the parameters that appear in \(pds\), which is indicated by \(\alpha(pds)\), where \(\alpha(pds)\) gives the list of parameter names declared in \(pds\). For instance, if \(pds\) is the declaration \((\text{val}\ x:\ T, \text{res}\ y:\ T)\), the list given by \(\alpha(pds)\) is \(x, y\). The class \(B\) on the left-hand side declares no method like \(m_1\) of the class \(A\).

On the right-hand side of this rule, the method \(m_1\) in \(A\) becomes just a delegating method: it calls the corresponding method of class \(B\) and passes the parameters in \(pds\) as arguments. The class \(B\) declares a method called \(m_1\) similar to the method \(m_1\) that appears on the left-hand side of the rule. The attribute \(x\) is accessed by means of the get and set methods declared in \(B\).

To apply this rule, we assume that \(b \neq \text{null}\) and that \(b \neq \text{error}\) along the lifetime of an object of type \(A\), otherwise a call to the method \(m_1\) of the class \(A\) would abort because this method calls methods of the class \(B\). So, the predicate \(b \neq \text{null} \land b \neq \text{error}\) is an invariant for class \(A\). We assume this based on the condition about the initialiser of class \(A\): the proviso requires that the attribute \(b\) is initialised with an object of type \(B\) and the initialiser terminates. Also, the parameters \(\text{arg}_1\) and \(\text{arg}_2\) must have basic types.

The notion of class invariant we use is similar to those of Meyer \([62]\) and Opdyke \([69]\). According to Meyer, an invariant for a class is a set of assertions that every instance of such a class will satisfy at all observable states, that is, after its creation, and before and after every call to a class method. In \texttt{ROOL}, we write invariants as predicates.

In order to apply this rule from left to right, the method \(m_1\) must not be declared in \(B\), nor in any of its superclasses or subclasses, and there should be no occurrences of \texttt{super} in \(m_1\). Also, attributes declared in \(A\) are not accessed in \(m_1\), and methods declared in \(A\) are not called inside \(m_1\). The reason for the last two provisos is that, to move a method in which there are occurrences of attributes or calls to methods of the source class (the one that introduces the method), it would be necessary to pass \texttt{self} as argument in the calls to the method of the target class (the class to where we move the method). In this way, the parameter declaration of the method in the target class would include an additional parameter: one whose type is the source class. Also, clients of the target class that call the method that was moved would also need to declare an object of the source class and pass such an object as argument in the method call. To avoid the additional parameter,
Rule 4.2 (Move Method)

class A extends C
  pri b : B; ads_a;
  meth m1 ≜ (pds •
    c1[var aux : T • self.b.getX(aux);
    le := exp1[aux] end,
    self.b.setX(exp2, α(pds))]
  new ≜ self.b := new B
  mts_a
end

class B extends D
  pri x : T; ads_b;
  meth getX ≜
    (res arg' : T • arg' := self.x)
  meth setX ≜
    (val arg' : T • self.x := arg')
  mts_b
end

where

α(pds) gives the names of parameters in pds;

provided

(→) \( b \neq \text{null} \land b \neq \text{error} \) is an invariant of A;

arg1 and arg2 have basic types;

(→) \( m_1 \) is not declared in mts_b nor in any superclass or subclass of B;

super does not appear in \( m_1 \);

self.a does not appear in \( m_1 \), for any \( a \) declared in ads_a;

self.p does not appear in \( m_1 \), for any method \( p \) in mts_a;

(←) \( N.m_1 \) does not appear in mts_b, cds, or c, for any \( N \) such that \( N \leq B \).

we restrict the method \( m_1 \) not to access attributes nor call methods declared in A.

For applying this rule from right to left, the method \( m_1 \) must not be called in mts_b, cds, or c, as this method is removed from B.
Derivation. We begin the derivation by introducing a method \( m_1 \) in class \( B \). Such method is similar to that of class \( A \), but it reads and writes the attribute \( x \) by means of the get and set methods already present in \( B \). In the derivation, it is not possible to deal with an arbitrary parameter declaration \( pds \) in method \( m_1 \), since \( pds \) may have any number of parameters with different parameter passing mechanisms. So, it is necessary to have a specific list of parameters. We define the list \( \text{val arg}_1 : T_1; \text{res arg}_2 : T_2 \) to be the parameters of such method. This does not mean a lack of abstraction, as the derivation for a longer parameter list is similar. We also change the visibility of attribute \( x \) from private to public.

\[
\begin{align*}
\text{class B extends D} \\
\text{pri x : T; adsb;} \\
\text{meth getX} \equiv (\text{res arg : T \bullet arg := self.x}) \\
\text{meth setX} \equiv (\text{val arg : T \bullet self.x := arg}) \\
mts_b \\
\end{align*}
\]

= Law 90 (method elimination) \((r/l)\), Law 85 (change visibility: from private to public) \((l/r)\)

\[
\begin{align*}
\text{class B extends D} \\
\text{pub x : T; adsb;} \\
\text{meth getX} \equiv (\text{res arg'} : T \bullet arg' := \text{self.x}) \\
\text{meth setX} \equiv (\text{val arg'} : T \bullet \text{self.x := arg'}) \\
\text{meth } m_1 \equiv (\text{val arg}_1 : T_1; \text{res arg}_2 : T_2 \bullet \\
\quad c_1[\text{var aux : T \bullet self.getX(aux); le := exp}_1[\text{aux}] \text{end, self.setX}(exp_2), \text{arg}_1, \text{arg}_2] \\
mts_b \\
\end{align*}
\]

In the method \( m_1 \) of class \( A \) we introduce the same parameterised command as the one that defines the method \( m_1 \) of class \( B \). For this, we first introduce a parameterised command with a formal result argument that is applied to \( \text{arg}_2 \).

\[
\begin{align*}
\text{class A extends C} \\
\text{pri b : B; adsa;} \\
\text{meth } m_1 \equiv (\text{val arg}_1 : T_1; \text{res arg}_2 : T_2 \bullet \\
\quad c_1[\text{var aux : T \bullet self.b.getX(aux); le := exp}_1[\text{aux}] \text{end, self.b.setX}(exp_2), \text{arg}_1, \text{arg}_2] \\
\quad \text{new } \equiv \text{self.b := new } B() \\
mts_a \\
\end{align*}
\]

end
We introduce a parameterised command with a formal value argument that is applied to the argument \( \text{arg}_1 \). Then we merge the two parameterised commands.

In class \( A \), we introduce the class invariant \( \{ b \neq \text{null} \land b \neq \text{error} \} \) as the last command of method \( m_1 \). Then we move this assumption to the beginning of \( m_1 \).
4.2. REFACTORIZING RULES

= Law [107] (introduce class invariant) (1/r)

\[
\begin{align*}
\text{class } A \ &\text{ extends } C \\
&\text{pri } b : B; \ ads_a; \\
&\text{meth } m_1 \triangleq (\text{val } \text{arg}_1 : T_1; \ \text{res } \text{arg}_2 : T_2 \bullet \\
&\quad (\text{val } \text{arg}_1 : T_1; \ \text{res } \text{arg}_2 : T_2 \bullet \\
&\quad c_1[\text{var } \text{aux} : T \bullet \text{self}.b.\text{getX}(\text{aux}); \ \text{le} := \text{exp}_{1[\text{aux}] \ \text{end}}, \\
&\quad \text{self}.b.\text{setX}(\text{exp}_2, \ \text{arg}'_1, \ \text{arg}'_2)])[\text{arg}_1, \ \text{arg}_2][b \neq \text{null} \land b \neq \text{error}] \\
&\quad \text{new } \triangleq \text{self}.b := \text{new } B() \\
&\text{mts}_a \\
\end{align*}
\]

Then, we eliminate the call to \text{getX} on \text{self}.b that appears in \text{c}_1.

= Law [95] (method call elimination) (1/r)

\[
\begin{align*}
\text{class } A \ &\text{ extends } C \\
&\text{pri } b : B; \ ads_a; \\
&\text{meth } m_1 \triangleq (\text{val } \text{arg}_1 : T_1; \ \text{res } \text{arg}_2 : T_2 \bullet \\
&\quad (\text{val } \text{arg}_1 : T_1; \ \text{res } \text{arg}_2 : T_2 \bullet \\
&\quad c_1[\text{var } \text{aux} : T \bullet \{\text{self}.b \neq \text{null} \land \text{self}.b \neq \text{error}\} \\
&\quad (\text{res } \text{arg}' : T \bullet \text{arg}' := \text{self}.b.\text{x}(\text{aux}); \ \text{le} := \text{exp}_{1[\text{aux}] \ \text{end}}, \\
&\quad \text{self}.b.\text{setX}(\text{exp}_2, \ \text{arg}'_1, \ \text{arg}'_2)])[\text{arg}_1, \ \text{arg}_2][b \neq \text{null} \land b \neq \text{error}] \\
&\quad \text{new } \triangleq \text{self}.b := \text{new } B() \\
&\text{mts}_a \\
\end{align*}
\]

= Law [53] (assumption before or after command) (r/l)

\[
\begin{align*}
\text{class } A \ &\text{ extends } C \\
&\text{pri } b : B; \ ads_a; \\
&\text{meth } m_1 \triangleq (\text{val } \text{arg}_1 : T_1; \ \text{res } \text{arg}_2 : T_2 \bullet \\
&\quad \{b \neq \text{null} \land b \neq \text{error}\}[\text{val } \text{arg}_1 : T_1; \ \text{res } \text{arg}_2 : T_2 \bullet \\
&\quad c_1[\text{var } \text{aux} : T \bullet \{\text{self}.b \neq \text{null} \land \text{self}.b \neq \text{error}\} \\
&\quad (\text{res } \text{arg}' : T \bullet \text{arg}' := \text{self}.b.\text{x}(\text{aux}); \ \text{le} := \text{exp}_{1[\text{aux}] \ \text{end}}, \\
&\quad \text{self}.b.\text{setX}(\text{exp}_2, \ \text{arg}'_1, \ \text{arg}'_2)])[\text{arg}_1, \ \text{arg}_2] \\
&\quad \text{new } \triangleq \text{self}.b := \text{new } B() \\
&\text{mts}_a \\
\end{align*}
\]

Now we introduce the call to method \text{getX} on \text{self}.b in command \text{c}_1.
= Law 95 \langle method call elimination \rangle (r/l)

class A extends C
  pri b : B; ads_a;
  meth m_1 \triangleq (val arg_1 : T_1; res arg_2 : T_2 •
    \{ b \neq \text{null} \land b \neq \text{error}\}(val arg_1 : T_1; res arg_2 : T_2 •
    c_1[var aux : T • self.b.getX(aux); le := \exp_1[aux] end,
    self.b.setX(\exp_2, \text{arg}_1', \text{arg}_2')(\text{arg}_1, \text{arg}_2)
    new \triangleq \text{self.b := new} B()
  mts_a
end

With the sequential composition constituted by the assumption and the parameterised command applied to the argument \text{arg}_1, we can introduce a call to method \text{m}_1 of class \text{B}. Notice that the parameterised command is the same as the one used in the definition of method \text{m}_1 of class \text{B}.

= Law 95 \langle method call elimination \rangle (r/l)

class A extends C
  pri b : B; ads_a;
  meth m_1 \triangleq (val arg_1 : T_1; res arg_2 : T_2 • self.b.m_1(\text{arg}_1, \text{arg}_2))
  new \triangleq \text{self.b := new} B()
  mts_a
end

Finally, by applying Law 85 \langle change visibility: from private to public \rangle, from right to left, we change the visibility of attribute \text{x} of class \text{b} back to private. This finishes the derivation of the refactoring rule \langle Move Method \rangle.

Notice that we have just considered one attribute in class \text{A}. But we could also deal with an arbitrary number of attributes, each of them with appropriate get and set methods. The proof for this situation is similar to the one we have presented here.

Roberts [74] states in the postcondition of his refactoring for moving methods between classes that the moved method can refer to attributes of its original class. For this, the method in the new class must take an additional parameter for objects of the original class. We have not presented this refactoring with such additional parameter because ROOL has a copy semantics. As a consequence, if \text{self} is an argument in the call to the moved method, any modification to attributes of the original class is not reflected in the object that originates the call to the moved method. Indeed, in our rule the types of the parameters of parameterised command we introduce in derivation are required to be basic.
4.2.3 Move Attribute

Moving attributes between classes is also a common activity during program development. If an attribute is used by methods of another class—through get and set methods—more than by those methods of the class in which it is declared, then such an attribute probably belongs to that other class; it is intrinsic to the concept described by another class.

We can apply rule \langle Move Attribute \rangle (Rule 4.3) for moving private attributes between classes. It moves an attribute \( x \) of a class \( A \) to a class \( B \), when applied from left to right. In \( A \), instead of \( x \), we end up with an attribute of type \( B \). When applied from right to left, we move \( x \) from \( B \) to \( A \).

The class \( A \), from which we move the attribute \( x \), after the application of this rule from left to right, has an attribute \( b \) of type \( B \) that cannot be declared in \( A \). This attribute is initialised in the method \texttt{new} of class \( A \) with an object of \( B \). The attribute to be moved cannot be already declared in class \( B \), the target class. Get and set methods are declared in \( B \) in order to allow indirect access to \( x \). These methods cannot be already declared in any superclass of \( B \), in \( B \) itself, nor in any of its subclasses. Commands in \( A \) that read and write the attribute \( x \) have to use the get and set methods introduced in \( B \). In order to call these methods in class \( A \), we use the attribute \( b \). Notice that it leads to changes in \( A \) as whole, the new methods are denoted by \( mts'_a \).

After the application of this rule from right to left, the class \( A \) has an attribute \( x \) of type \( T \). Class \( B \) has no declaration of \( x \), and methods \texttt{getX} and \texttt{setX} are removed from this class. The proviso requires \( x \) not to be declared in \( A \) nor in any superclass or subclass of \( A \). The methods \texttt{getX} and \texttt{setX} must also not be called inside \( B \), nor in \( cds \) or \( c \). Also, we require \( b \neq \texttt{null} \land b \neq \texttt{error} \) to be an invariant of class \( A \). This avoids program abortion due to calls to methods of \( B \).

**Derivation.** We begin with the left-hand side of the rule \langle Move Attribute \rangle. Our intention is to move the attribute \( x \) from class \( A \) to class \( B \).

```
class A
  pri x : T; ads_a;
  meth m_1 := (pds_1 • c[le := exp[self.x], self.x := exp])
  mts_a
end
```

```
class B
  ads_b;
  mts_b
end
```

The first step is to introduce an attribute \( x \) of type \( T \) in class \( B \). In other words, we introduce an attribute that has the same type as the one we want to move. As we required the attribute \( x \) must not to be declared in \( B \), so we can apply the law for attribute elimination, from right to left. Then we change the visibility of \( x \) from private to public.
### Rule 4.3 (Move Attribute)

<table>
<thead>
<tr>
<th>Class $A$ extends $C$</th>
<th>Class $A$ extends $C$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>pri</strong> $x : T$; $ads_a$;</td>
<td><strong>pri</strong> $b : B$; $ads_a$;</td>
</tr>
<tr>
<td><strong>meth</strong> $m_1 \doteq (pds_1 \bullet$</td>
<td><strong>meth</strong> $m_1 \doteq (pds_1 \bullet c'_1)$</td>
</tr>
<tr>
<td>$c_1[le \leftarrow exp]\langle self.x],$</td>
<td><strong>meth</strong> $getX \doteq$</td>
</tr>
<tr>
<td>$self.x := exp\rangle$</td>
<td>$(res \ arg : T \bullet \text{self}.b.getX(arg))$</td>
</tr>
<tr>
<td><strong>meth</strong> $getX \doteq$</td>
<td><strong>meth</strong> $setX \doteq$</td>
</tr>
<tr>
<td>$(res \ arg : T \bullet arg := self.x)$</td>
<td>$(val \ arg : T \bullet \text{self}.b.setX(arg))$</td>
</tr>
<tr>
<td><strong>meth</strong> $setX \doteq$</td>
<td>**new \doteq \text{self}.b := \text{new} B()$</td>
</tr>
<tr>
<td>$(val \ arg : T \bullet self.x := arg)$</td>
<td>$mts'_a$</td>
</tr>
<tr>
<td>$mts_a$</td>
<td><strong>end</strong></td>
</tr>
<tr>
<td><strong>class</strong> $B$</td>
<td><strong>class</strong> $B$</td>
</tr>
<tr>
<td><strong>ads$_b$;</strong></td>
<td><strong>pri</strong> $x : T$; $ads_b$</td>
</tr>
<tr>
<td><strong>mts$_b$</strong></td>
<td><strong>meth</strong> $getX \doteq$</td>
</tr>
<tr>
<td><strong>end</strong></td>
<td>$(res \ arg' : T \bullet arg' := self.x)$</td>
</tr>
<tr>
<td></td>
<td><strong>meth</strong> $setX \doteq$</td>
</tr>
<tr>
<td></td>
<td>$(val \ arg' : T \bullet \text{self}.x := arg')$</td>
</tr>
<tr>
<td></td>
<td>$mts_b$</td>
</tr>
</tbody>
</table>

where

$c'_1 \doteq c_1[\text{var} \ aux : T \bullet \text{self}.b.getX(aux); \ le := \exp[aux] \langle \text{end}, \text{self}.b.setX(\exp)/$

$\le := \exp[\text{self}.x], \text{self}.x := \exp]$  

$mts'_a \doteq mts_a[\text{var} \ aux : T \bullet \text{self}.b.getX(aux); \ le := \exp[aux] \langle \text{end}, \text{self}.b.setX(\exp)/$

$\le := \exp[\text{self}.x], \text{self}.x := \exp]$  

provided

$(\leftrightarrow)$ $b \neq \text{null} \land b \neq \text{error}$ is an invariant of $A'$

$x$ has basic type;

$(\rightarrow)$ $x$ is not declared in $ads_b$ nor in any superclass or subclass of $B$;

$b$ is not declared in $ads_a$ nor in any superclass or subclass of $A$;

the initialiser of class $B$ terminates;

$getX$ and $setX$ are not declared in $mts_b$ nor in any superclass or subclass of $B$;

$(\leftarrow)$ $x$ is not declared in $mts_a$ nor in any superclass or subclass of $A$ in $cds_s$;

$N.getX$ and $N.getX$ do not appear in $mts_b$, $cds$ or $c$, for any $N \leq B$.  

4.2. REFACTORING RULES

= Law \[83\] (attribute elimination) \((r/l)\), Law \[85\] (change visibility: from private to public) \((l/r)\)

\[
\text{class } B \\
\text{ pub } x : T; \ ads_b; \\
\text{ mts}_b \\
\text{ end}
\]

Also in class \(B\), using Law \[90\] (method elimination) twice, we introduce get and set methods for the new attribute \(x\).

= Law \[90\] (method elimination) \((r/l)(2x)\)

\[
\text{class } B \\
\text{ pri } x : T; \ ads_b; \\
\text{ meth } \textit{getX} \equiv (\text{res } arg : T \bullet arg := \text{self}.x) \\
\text{ meth } \textit{setX} \equiv (\text{val } arg : T \bullet \text{self}.x := arg) \\
\text{ mts}_b \\
\text{ end}
\]

Now we turn our attention to class \(A\).

\[
\text{class } A \\
\text{ pri } x : T; \ ads_a; \\
\text{ meth } m_1 \equiv (\text{pds}_1 \bullet c[le := \text{exp[\text{self}.x]}, \text{self}.x := \text{exp}]) \\
\text{ meth } \textit{getX} \equiv (\text{res } arg : T \bullet arg := \text{self}.x) \\
\text{ meth } \textit{setX} \equiv (\text{val } arg : T \bullet \text{self}.x := arg) \\
\text{ mts}_a \\
\text{ end}
\]

We apply the Law \[78\] (simple specification) so that we can apply the rules for data refinement later. This law is applied to assignments in which the variable we want to substitute appear on the right-hand side of the assignment. Notice that we want to substitute the attribute \(x\) of class \(A\) by attribute \(x\) of class \(B\).

= Law \[78\] (simple specification) \((l/r)\)

\[
\text{class } A \\
\text{ pri } x : T; \ ads_a; \\
\text{ meth } m_1 \equiv (\text{pds}_1 \bullet c[le := \text{exp[\text{self}.x]}, \text{self}.x := \text{exp}]) \\
\text{ meth } \textit{getX} \equiv (\text{res } arg : T \bullet arg := \text{arg} = \text{self}.x) \\
\text{ meth } \textit{setX} \equiv (\text{val } arg : T \bullet \text{self}.x, b.x := arg, arg) \\
\text{ mts}_a \\
\text{ end}
\]
In class $A$ we perform a data refinement: we introduce the private attribute $b$ of type $B$ along with the coupling invariant $self.x = b.x \land self.b \neq \text{null} \land self.b \neq \text{error}$. The application of this law changes specification, assignments, and guards involving the attribute $x$, which is considered an abstract variable in the context of this data refinement, in the whole class $A$. Here we concentrate on method $m_1$.

= Law [108] (private attribute-coupling invariant)

\[ CI \equiv self.x = b.x \land self.b \neq \text{null} \land self.b \neq \text{error} \]

class $A$

\begin{align*}
\text{pri } \ & b : B; \\
\text{pri } \ & x : T; \ ads_a; \\
\text{meth } \ & m_1 \equiv (\text{pds}_1 \land c[le, b.x : [le = \text{exp}[self.x] \land CI], self.x, self.b.x := \text{exp}, \text{exp}]) \\
\text{meth } \ & \text{getX} \equiv (\text{res } \text{arg} : T \land \text{arg} : [\text{arg} = self.x \land CI]) \\
\text{meth } \ & \text{setX} \equiv (\text{val } \text{arg} : T \land self.x, self.b.x := \text{arg}, \text{arg}) \\
\text{mts}_a \ & \text{new} \equiv self.b := \text{new } B() \\
\end{align*}

end

In the specification statement we substitute the expression $\text{self}.x$ with $b.x$, as their relationship is stated in the coupling invariant $CI$.

= Predicate calculus

\[ CI \equiv self.x = self.b.x \land b \neq \text{null} \land b \neq \text{error} \]

class $A$

\begin{align*}
\text{pri } \ & b : B; \\
\text{pri } \ & x : T; \ ads_a; \\
\text{meth } \ & m_1 \equiv (\text{pds}_1 \land c[le, b.x : [le = \text{exp}[self.b.x] \land CI], self.x, self.b.x := \text{exp}, \text{exp}]) \\
\text{meth } \ & \text{getX} \equiv (\text{res } \text{arg} : T \land \text{arg} : [\text{arg} = self.b.x \land CI]) \\
\text{meth } \ & \text{setX} \equiv (\text{val } \text{arg} : T \land self.x, self.b.x := \text{arg}, \text{arg}) \\
\text{mts}_a \ & \text{new} \equiv self.b := \text{new } B() \\
\end{align*}

end

We assume that the initialiser of class $B$ terminates, so that this invariant is established by the method $\text{new}$. It is also preserved by other methods because $b$ is an attribute recently introduced and it is not assigned in methods of $A$. So, we introduce the predicate $self.b \neq \text{null} \land self.b \neq \text{error}$ as a class invariant. We define $\text{classAInv}$ as an abbreviation for this predicate.
4.2. REFACTORING RULES

Law $[107]$ (introduce class invariant) $\langle l/r \rangle$, Law $[53]$ (assumption before or after command) $\langle r/l \rangle$

$\text{classAInv} \triangleq \text{self} . b \neq \text{null} \land \text{self} . b \neq \text{error}$

class $A$

$\text{pri } x : T; \text{pri } b : B; \text{ads}_a;$

$\text{meth } m_1 \triangleq (\text{pds}_1 \bullet \{ \text{classAInv} \};\ c[le, b.x : [le = exp[\text{self} . x] \land CI], \text{self} . x, \text{self} . b.x := exp, exp])$

$\text{meth } \text{getX} \triangleq (\text{res } \text{arg} : T \bullet \{ \text{classAInv} \} \ \text{arg} := \text{self} . b.x)$

$\text{meth } \text{setX} \triangleq (\text{val } \text{arg} : T \bullet \{ \text{classAInv} \} \ \text{self} . b.x := \text{arg})$

$\text{new } \triangleq \text{self} . b := \text{new } B()$

$\text{mts}_a$

end

By using Law $[69]$ (assignment), we refine the specification $le, b.x : [le = exp[b.x] \land CI]$ to the assignment $le := exp[b.x]$. The assignment $\text{self} . x, \text{self} . b.x := exp, exp$ becomes $\text{self} . b.x := exp$, by using Law $[73]$ (diminish assignment).

$\ll$ Law $[69]$ (assignment), Law $[73]$ (diminish assignment)

$CI \triangleq \text{self} . x = b.x \land b \neq \text{null} \land b \neq \text{error}$

class $A$

$\text{pri } b : B;$

$\text{pri } x : T; \text{ads}_a;$

$\text{meth } m_1 \triangleq (\text{pds}_1 \bullet \{ \text{classAInv} \};\ c[le := exp[\text{self} . b.x], \text{self} . b.x := \text{exp}])]$

$\text{meth } \text{getX} \triangleq (\text{res } \text{arg} : T \bullet \text{arg} := \text{self} . b.x)$

$\text{meth } \text{setX} \triangleq (\text{val } \text{arg} : T \bullet \text{self} . b.x := \text{arg})$

$\text{mts}_a$

$\text{new } \triangleq \text{self} . b := \text{new } B()$

end

As already said, the attribute $b$ was not present in class $A$ before the derivation, and we assign to $b$ an object of class $B$ in the method $\text{new}$. Besides that, we only assign to attributes of the object denoted by $b$, not to $b$ itself. Therefore, after the data refinement, a $\text{null}$ object is not assigned to $b$. So, we can assume in all points in method $m_1$ that $b$ is a non-null object. So, the assumption we introduced above is innocuous because the predicate $b \neq \text{null} \land b \neq \text{error}$ is true at all points of $m_1$ and, consequently, the assumption behaves as $\text{skip}$, which does not affect the behaviour of $m_1$. We can distribute such assumption in the command $c$ of $m_1$.
We introduce the local variable \( aux \) to which we assign the value of the attribute \( x \) of the object denoted by \( b \). This assignment is introduced at the end of the variable block.

\[
\begin{align*}
\text{class } A & \quad \text{pri } x : T; \text{ pri } b : B; \text{ ads}_a; \\
\text{meth } m_1 \triangleq (pds_1 \bullet c[\{classAInv\} le := \exp[\text{self}.b.x], \quad \langle \text{classAInv} \rangle \text{self}.b.x := \exp]) \quad (i) \\
\text{meth } getX \triangleq (\text{res arg} : T \bullet \{\text{classAInv}\} \text{arg} := \text{self}.b.x) \\
\text{meth } setX \triangleq (\text{val arg} : T \bullet \{\text{classAInv}\} \text{self}.b.x := \text{arg}) \\
\text{new } \triangleq \text{self}.b := \text{new } B() \\
m_{ts_a}
\end{align*}
\]

end

We introduce the local variable \( aux \) to which we assign the value of the attribute \( x \) of the object denoted by \( b \). This assignment is introduced at the end of the variable block.

\[
\begin{align*}
\text{var } aux : T \bullet \{\text{classAInv}\} \text{le} := \exp[\text{self}.b.x]; \text{aux} := \text{self}.b.x \textbf{ end}
\end{align*}
\]

The variable \( aux \) is new and, for this reason, we can move the assignment to it inside the variable block. This assignment does not influence other commands.

\[
\begin{align*}
\text{var } aux : T \bullet \{\text{classAInv}\} \text{aux} := \text{self}.b.x; \text{le} := \exp[\text{self}.b.x] \textbf{ end}
\end{align*}
\]

The expression \( b.x \) that occurs on the right-hand side of the assignment to \( aux \) also occurs in expression \( \exp \). The assignment to \( aux \) occurs just before the command \( \text{le} := \exp[\text{self}.b.x] \), thus in this command we can replace \( \text{self}.b.x \) with \( aux \).

\[
\begin{align*}
\text{var } aux : T \bullet \{\text{classAInv}\} \text{aux} := \text{self}.b.x; \text{le} := \exp[\text{aux}] \textbf{ end}
\end{align*}
\]

The command \( aux := \text{self}.b.x \) reads the attribute \( x \) of the object denoted by the attribute \( b \). We can use the method \( getX \) of class \( B \) to get the value of such attribute. To introduce a call to \( getX \), we need a parameterised command. First, we introduce a variable block that is equivalent to a parameterised command with a result parameter as \( getX \). The variable \( aux \) is replaced with the new local variable. Recall that we defined \( classAInv \) to be the predicate \( \text{self}.b \neq \text{null} \land \text{self}.b \neq \text{error} \).
4.2. REFACTORING RULES

= Law \text{[63]} (\text{var block-res}) (1/r)

\[
\begin{align*}
\text{var aux : T • } & \{\text{self.b \neq null \land self.b \neq error}\} \\
& \text{var } p\_aux : T \bullet \\
& \quad p\_aux := \text{self.b.x} ; \\
& \quad aux := p\_aux \\
& \quad \text{end} ; \\
& \quad le := \exp[aux]
\end{align*}
\]

Then, we write the parameterised command equivalent to the variable block just introduced.

= Law \text{[65]} (\text{pcom elimination-res}) (r/1)

\[
\begin{align*}
\text{var aux : T • } & \{\text{self.b \neq null \land self.b \neq error}\} \text{ (res arg’ : T • arg’ := self.b.x)(aux)} \\
& \text{le := } \exp[aux]
\end{align*}
\]

Using the Law \text{[95]} (\text{method call elimination}), from right to left, we introduce a call to the method \text{getX} of class \text{B}. Notice that we have the required sequential composition of an assumption with a parameterised command in order to apply such law.

= Law \text{[95]} (\text{method call elimination}) (r/1)

\[
\begin{align*}
\text{var aux : T • } \\
& \text{self.b.getX(aux)} \\
& \text{le := } \exp[aux]
\end{align*}
\]

The command \text{self.b.x := exp} is an assignment to the attribute \text{x} of the object denoted by \text{b}. We can use the method \text{setX} of class \text{B} to indirectly assign a value to this attribute. We follow the same steps as we have done in the case of method \text{getX} above. However, here we introduce a parameterised command with a parameter passed by value.

(i) = Law \text{[62]} (\text{var block-val}) (1/r)

\[
\begin{align*}
\{\text{self.b \neq null \land self.b \neq error}\} \\
\text{var } p\_arg’ : T \bullet \\
& \quad p\_arg’ := \exp ; \\
& \quad \text{self.b.x := p\_arg’}
\end{align*}
\]
\[\text{Law 64 (pcom elimination-val) (r/l)}\]
\[
\{\text{self.} b \neq \text{null} \land \text{self.} b \neq \text{error}\} (\text{val } \text{arg}' : T \bullet \text{self.} b.x := \text{arg}'(\text{exp}))
\]

\[\text{Law 95 (method call elimination) (r/l)}\]
\[
\text{self.} b.\text{setX}(\text{exp})
\]

Now the classes \(A\) and \(B\) are as follows.

```plaintext
class A
  pri b : B;
  pri x : T; ads_a;
  meth m_1 \equiv (pds_1 \bullet
c[\text{var aux} : T \bullet \text{self.} b.\text{getX}(\text{aux});
  le := \text{exp}[\text{aux}] \text{end},
  \text{self.} b.\text{setX}(\text{exp}))

  meth \text{getX} \equiv (\text{res } \text{arg} : T \bullet \{\text{self.} b \neq \text{null} \land \text{self.} b \neq \text{error}\} \text{arg} := \text{self.} b.x)
  meth \text{setX} \equiv (\text{val } \text{arg} : T \bullet \text{self.} x := \text{arg})
  new \equiv \text{self.} b := \text{new } B()
mts_a'
end
```

We proceed in the same way to introduce calls to the methods \(\text{getX}\) and \(\text{setX}\) of class \(B\) in the methods \(\text{getX}\) and \(\text{setX}\) of class \(A\). After that, the attribute \(x\) of class \(A\) is no longer used inside this class, and we can remove it by applying the Law 83 (attribute elimination), from left to right. Direct accesses to attribute \(x\) that might exist in \(mts_a\) are replaced with calls to get and set methods related to attribute \(x\) of class \(B\), similarly to the changes in method \(m_1\). This is denoted by \(mts_a'\).

\[\text{Law 83 (attribute elimination) (l/r)}\]

```plaintext
class A
  pri b : B; ads_a;
  meth m_1 \equiv (pds_1 \bullet
c[\text{var aux} : T \bullet \text{self.} b.\text{getX}(\text{aux});
  le := \text{exp}[\text{aux}] \text{end},
  \text{self.} b.\text{setX}(\text{exp}))

  meth \text{getX} \equiv (\text{res } \text{arg} : T \bullet \{\text{self.} b \neq \text{null} \land \text{self.} b \neq \text{error}\} \text{arg} := \text{self.} b.x)
  meth \text{setX} \equiv (\text{val } \text{arg} : T \bullet \text{self.} b.\text{setX}(\text{arg}))
  new \equiv \text{self.} b := \text{new } B()
mts_a'
end
```
4.2. REFACTORING RULES

By using Law \[85\](change visibility: from private to public), from right to left, we privatise the attribute \(x\) of class \(B\) as it is not directly accessed from outside class \(B\). We finish the derivation with this step.

\[= \text{Law} \[85\] \langle \text{change visibility: from private to public} \rangle \ (r/l)\]

\[
\begin{align*}
\text{class } B
\text{pri } x : T; \text{ ads } b; \\
\text{meth } \text{getX} \equiv (\text{res } \text{arg'} : T \bullet \text{arg} := \text{self}.x) \\
\text{meth } \text{setX} \equiv (\text{val } \text{arg'} : T \bullet \text{self}.x := \text{arg}) \\
\text{mts}_b \\
\text{end}
\end{align*}
\]

The proof of this rule from right to left is similar to the one we presented here. First we change the visibility of \(x\) from private to public in \(B\). Then, we eliminate calls to \(\text{getX}\) and \(\text{setX}\) in \(A\). We prepare class \(A\) for data refinement and introduce the attribute \(x\) of type \(T\) with the coupling invariant we used for the proof from left to right. After this, we proceed with diminishing specification statements, assignments and guards, so that there are no references to \(b\) in \(A\). We eliminate \(b\) from \(A\), \(x\) and methods \(\text{getX}\) and \(\text{setX}\) from \(B\). As both sides are refinement of each other, we conclude that they are equal.

\[\square\]

If the visibility of the attribute we want to move is public, we should first apply the refactoring rule \(\langle \text{Encapsulate Field} \rangle\) (presented in Chapter 5), and then apply the rule \(\langle \text{Move Attribute} \rangle\) presented here. The rule \(\langle \text{Encapsulate Field} \rangle\) is not compositional, consequently, moving a public attribute from one class to another is a non-compositional transformation. Differently from Fowler’s refactoring, our rule requires that the attribute that we move from one class to another to have basic type. This is a consequence of restrictions of laws used to introduce parameterised commands, a means to introduce method calls.

4.2.4 Pull Up and Push Down Method

In a class hierarchy, methods defined by the same parameterised command, in different branches of the hierarchy, should not be duplicated. Changes to one method may not be applied to the other, resulting in inconsistent methods. In order to avoid this situation, we should move the methods to a common superclass, obtaining just one definition. This is the purpose of the refactoring \(\langle \text{Pull Up/Push Down Method} \rangle\)(Rule 4.4). When applying this rule from left to right, we move methods with the same definition to the superclass of the classes in which these definitions were present. In practice, the most common use of this rule involves its application for pulling up methods. On the other hand, if a method is called just on objects of a particular subclass, we can push down a method, and then remove it from classes whose objects are not target of calls to such
### Rule 4.4 (Pull Up/Push Down Method)

<table>
<thead>
<tr>
<th>Class A extends D</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ads_a );</td>
</tr>
<tr>
<td>( mts_a )</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>Class B extends A</td>
</tr>
<tr>
<td>( ads_b );</td>
</tr>
<tr>
<td>( \text{meth } m \equiv (pds_m \bullet c') )</td>
</tr>
<tr>
<td>( mts_b )</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>Class C extends A</td>
</tr>
<tr>
<td>( ads_c );</td>
</tr>
<tr>
<td>( \text{meth } m \equiv (pds_m \bullet c') )</td>
</tr>
<tr>
<td>( mts_c )</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class A extends D</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ads_a );</td>
</tr>
<tr>
<td>( \text{meth } m \equiv (pds_m \bullet c') )</td>
</tr>
<tr>
<td>( mts_a )</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>Class B extends A</td>
</tr>
<tr>
<td>( ads_b );</td>
</tr>
<tr>
<td>( mts_b )</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>Class C extends A</td>
</tr>
<tr>
<td>( ads_c );</td>
</tr>
<tr>
<td>( mts_c )</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>

provided

\((\leftarrow)\) super and private attributes do not appear in \( c' \);

\((\rightarrow)\) \( m \) is not declared in \( mts_a \), and can only be declared in a class \( N \), for any \( N \leq A \), if it has parameters \( pds_m \);

\( m \) is not declared in any superclass of \( A \) in \( cds \);

\((\leftarrow)\) \( m \) is not declared in \( mts_b \) or \( mts_c \);

super.\( m \) does not appear in \( mts_b \) or \( mts_c \) nor in any subclass of \( A \), which are not subclasses of \( B \) or \( C \);

\( N.m \), for any \( N \leq A \) and \( N \not< B \) or \( N \not< C \), does not appear in \( cds \), \( c \), \( mts_a \), \( mts_b \) or \( mts_c \).

In order to apply this rule, super and private attributes must not appear in \( c' \). For applying this rule from left to right, we require that any other subclass of \( A \), different from \( B \) and \( C \), which declares a method named \( m \), must define it with the same parameters as the method \( m \) that is pulled up. We also require that the method \( m \) is not declared in \( A \).

In order to apply this rule from right to left, the method \( m \) must not be already declared in the subclasses to which we push down the method. We also require that super.\( m \) must not appear in the classes to which we push \( m \) down nor in any subclass of \( A \). Also, there must be no calls to \( m \).
that have as target expressions which are not subclasses of the classes to which we push down \( m \), appearing in \( cds \) or \( c \).

Notice that in \( m \), the method being pulled up, there should be no accesses to attributes declared in the class in which \( m \) is declared. Also, inside \( m \) there should be no calls to methods declared in the class in which \( m \) is declared. This is a consequence of having command \( c' \) in the definitions of method \( m \) in \( B \) and \( C \).

**Derivation.** We assume that the provisos are valid and begin the derivation with the left-hand side. As we want to move the method \( m \), of classes \( B \) and \( C \), to \( A \), we have to apply laws that deal with moving methods up in a hierarchy. The Law [91](move original method to superclass), which moves an original method to a superclass, and the one for moving a redefined method to a superclass (Law [89](move redefined method to superclass)) require that occurrences of \( \text{self} \) in methods to be moved are cast. So, our first step is to introduce casts in occurrences of \( \text{self} \) in the method \( m \) of classes \( B \) and \( C \). We sequentially compound the command : \([\text{true}, \text{true}]\) with commands in which there are occurrences of \( \text{self} \). This composition is equivalent to the sequential composition of \text{skip} with commands in which \( \text{self} \) occurs. The specification statement : \([\text{true}, \text{true}]\) is equivalent to the assumption \( \{\text{true}\} \). According to the Law [101](is test true), from right to left, occurrences of assumption \( \{\text{true}\} \) can be written as \( \{\text{self is } A\} \). By applying Law [98](eliminate cast of expressions), from right to left, we cast \( \text{self} \) to \( A \) in expressions. The same changes are applied to class \( C \). The result is denoted by \( c'' \).

\[
\begin{align*}
\text{class } A & \text{ extends } D \\
ads_a; & \\
mts_a & \\
end & \\
\text{class } B & \text{ extends } A \\
ads_b; & \\
\text{meth } m \equiv (pds \bullet c'') & \\
mts_b & \\
end & \\
\text{class } C & \text{ extends } A \\
ads_c; & \\
\text{meth } m \equiv (pds \bullet c'') & \\
mts_c & \\
end
\end{align*}
\]

By applying the Law [91](move original method to superclass), we move the method \( m \) from \( B \) to \( A \), as \( m \) is not declared in \( mts_a \). The provisos of this law are also present in the provisos of the refactoring rule. The classes \( A \), \( B \), and \( C \) are now as follows.

\[
\begin{align*}
\text{class } A & \text{ extends } D \\
ads_a; & \\
\text{meth } m \equiv (pds \bullet c'') & \\
mts_a & \\
end & \\
\text{class } B & \text{ extends } A \\
ads_b; & \\
mts_b & \\
end & \\
\text{class } C & \text{ extends } A \\
ads_c; & \\
\text{meth } m \equiv (pds \bullet c'') & \\
mts_c & \\
end
\end{align*}
\]

Now, there is already a definition of the method \( m \) in the class \( A \). In order to move the method \( m \) from \( C \) to \( A \) we apply the Law [89](move redefined method to superclass). This applica-
tion introduces an alternation in the method $m$ declared in $A$, yielding the following.

$$= \text{Law} \, \text{89 (move redefined method to superclass)} \, (l/r)$$

$$\begin{align*}
\text{class } A \text{ extends } D & \\
ads_a; & \\
\text{meth } m & \equiv (pds \bullet \\
\text{if } \neg (\text{self is } C) \rightarrow c'' \\
\text{fi}) & \\
mts_a & \text{end} & \\
\text{class } B \text{ extends } A & \\
ads_b; & \\
mts_b & \text{end} & \\
\text{class } C \text{ extends } A & \\
ads_c; & \\
mts_c & \text{end} &
\end{align*}$$

The disjunction of the guards of the alternation that appears in the method $m$ is true, and the same command is guarded in both branches of the alternation. In this way, regardless of the guard that is evaluated to true, the same command is executed. To simplify this alternation we apply Law \, \text{44 (if identical guarded commands)}. We obtain the original command $c'$ by applying Law \, \text{97 (introduce trivial cast in expressions)}, from right to left.

$$= \text{Law} \, \text{44 (if identical guarded commands)} \, (l/r),$$

\text{Law} \, \text{97 (introduce trivial cast in expressions)} \, (r/l)

$$\begin{align*}
\text{class } A \text{ extends } D & \\
ads_a; & \\
\text{meth } m & \equiv (pds \bullet c') & \\
mts_a & \text{end} & \\
\text{class } B \text{ extends } A & \\
ads_b; & \\
mts_b & \text{end} & \\
\text{class } C \text{ extends } A & \\
ads_c; & \\
mts_c & \text{end} &
\end{align*}$$

With this step we finish the derivation of the rule \, \langle \text{Pull Up/Push Down Method} \rangle. \quad \Box$

The derivation of a rule that considers an arbitrary number of subclasses is similar. First, casts must be introduced and then methods must be pulled up. This results in an alternation whose branches are defined by the same command. So the command to be executed, in spite of evaluation of guards, is the same. By Law \, \text{44 (if identical guarded commands)}, we simplify the alternation to the guarded command.

The refactoring we presented here is similar to the refactoring \text{Pull Up Method} presented by Fowler [42, p.322], as he moves methods with the same name and bodies from two subclasses to their common superclass. On the other hand, our definition slightly differ from the one presented by Fowler for refactoring \text{Push Down Method} [42, p. 328], as a method is pushed down to a specific subclass. In fact, our refactoring rule is more general than Fowler’s refactoring, in the sense that we push down a method from a superclass to all its immediate subclasses. After this,
4.2. REFACTORING RULES

if a method is called on object of just one of the subclasses, we can remove such method from the subclasses whose objects are not target of calls to the method we push down.

The Eclipse IDE [37] implements a refactorings that are useful for moving methods and attributes up (PULL UP...) and down (PUSH DOWN...) a hierarchy. The refactoring PULL UP... restricts the access to an attribute in a class from a method that is moved to the superclass of such class: this access cannot use casts. A warning is given to the Eclipse user about the impossibility of accessing the attribute, which is in the subclass, from the superclass even via an expression with a cast. If the warning is ignored and the refactoring is completed, the refactored code compiles without problems and the resulting program behaves as the one before refactoring. This situation is a consequence of an implementation decision for the refactoring that does not take casts into account in refactorings [84]. In fact, casts may not improve a program design, but it should be possible to refactor a program with casts, otherwise it should be clear that programmers should not write programs that use casts.

An inconsistency appears in a refactored program, in the Eclipse IDE [37], when we move methods with the same name, but different code, from different classes to their common superclass of such classes. This happens, for instance, when the methods we move to the common superclass is different from each other only as a consequence of the occurrence of casts to the classes in which the methods are declared. These casts appear in expressions that access attributes declared in the subclasses. During the refactoring, only a warning concerning the access of the attributes declared in the subclasses from the method that we move to the superclass is given to the user. There is no warning concerning the differences between method bodies, so it seems that there is no problem moving methods with the same names, but with different bodies, from classes to the common superclass of them. This leads to a break in program behaviour.

In the case of pushing down a method from a superclass to subclasses, the Eclipse IDE [37] does not give any warning when we push down a method to subclasses, excluding a subclass whose instances are target of calls to the method we are pushing down. This leads to a compilation error.

Although Opdyke [69] does not present a refactoring as the one presented here, when he presents the use of refactorings for creating a superclass, the methods to be abstracted have the same bodies. For methods that have the same body but different names, renaming is a solution. In this case, it is necessary to apply the rule (Rename Method) (Rule 5.6), presented in Chapter 5, as many times as necessary, before applying (Pull Up/Push Down Method). The refactoring rule (Pull Up/Push Down Method) still remains compositional; it is just preceded by an application of a contextual refactoring. On the other hand, if we assume in (Pull Up/Push Down Method) that methods may have different names, the first step in the derivation is the application of (Rename Method). Consequently, (Pull Up/Push Down Method) would be contextual. From a strict point of view, this would not be (Pull Up/Push Down Method).
### Rule 4.5 (Replace Parameter with Method)

<table>
<thead>
<tr>
<th>Class $A$ extends $C$</th>
<th>Class $A$ extends $C$</th>
</tr>
</thead>
<tbody>
<tr>
<td>class $A$ extends $C$</td>
<td>class $A$ extends $C$</td>
</tr>
<tr>
<td>$\textit{ads}_a$;</td>
<td>$\textit{ads}_a$;</td>
</tr>
<tr>
<td>$\textit{meth} m_1 \triangleq (pds_1 \bullet c_1);$</td>
<td>$\textit{meth} m_1 \triangleq (pds_1 \bullet c_1);$</td>
</tr>
<tr>
<td>$\textit{var} x, y : T_1, T_2 \bullet c_3[x];$</td>
<td>$\textit{var} y : T_2 \bullet \textit{self}.m_2(y)$;</td>
</tr>
<tr>
<td>self.$m_2(x, y)$ end)</td>
<td>end)</td>
</tr>
<tr>
<td>$\textit{meth} m_2 \triangleq$</td>
<td>$\textit{meth} m_2 \triangleq$</td>
</tr>
<tr>
<td>(val arg$_1 : T_1; \textit{res} arg$_2 : T_2 \bullet$</td>
<td>(res arg$_2 : T_2 \bullet$</td>
</tr>
<tr>
<td>$c_2[\textit{arg}_1, \textit{arg}_2]$)</td>
<td>$\textit{var} x : T_1 \bullet \textit{self}.m_3(\textit{arg})$;</td>
</tr>
<tr>
<td>$\textit{mts}_a$ end</td>
<td>$c_2[\textit{x}, \textit{arg}_2]$ end)</td>
</tr>
<tr>
<td>$\textit{mts}_a$ end</td>
<td>$\textit{meth} m_3 \triangleq (\textit{res} \textit{arg} : T_1 \bullet c_3[\textit{arg}])$</td>
</tr>
</tbody>
</table>

Provided

- The types of variables $x$ and $y$ are basic;
- $m_3$ is not declared in $\textit{mts}_a$ nor in any subclass or superclass of $A$;
  - $N.m_2$ does not appear in $\textit{cds}$ or $c$, for any $N$ such that $N \leq A$;
  - $\alpha(pds_1)$ are not free in $c_3$;
  - The local variable $y$ does not occur free in $c_3$
- $N.m_3$ does not appear in $\textit{cds}$ or $c$, for any $N$ such that $N \leq A$.

#### 4.2.5 Replace Parameter with Method

In Rule 4.5, the method $m_1$, on the left-hand side of the rule, calls the method $m_2$ passing $x$, a locally declared variable, as a value argument. After application of this rule, the method $m_1$ still calls $m_2$. The major changes occurs in $m_2$. The command $c_3$ in which $x$ occurs in the original $m_1$ is extracted into the method $m_3$, which has a result argument $\textit{arg}$ corresponding to $x$. Instead of passing the result of the call to $m_3$, as an argument, in the call to $m_2$, the method $m_1$ just calls $m_2$, which is responsible for calling $m_3$ with argument $x$. Consequently, the number of parameters of $m_2$ is smaller than it was originally. The local variable $x$ that is present in the method $m_2$, on the right-hand side, replaces the parameter $\textit{arg}_1$ in the original $m_2$.

To apply this rule, the types of $x$ and $y$ must be basic. In order to apply this rule, from left to right, the method $m_3$ cannot be declared in any superclass of $A$, in $A$ itself, nor in any of its subclasses. Method $m_2$ cannot be called outside class $A$. Also, parameters of $m_1$ must not appear in $c_3$, otherwise we could not turn this command into a method independent of $m_1$. For applying this rule from right to left, method $m_3$ must not be called in $\textit{cds}$ or $c$, since it is removed.
Derivation. The derivation involves only class \( A \).

\[
\text{class } A \\
\text{ads_a;}
\]

\[
\text{meth } m_1 \equiv (\text{pds}_1 \bullet c_1; \\
\text{var } x, y : T_1, T_2 \bullet c_3[x]; \text{self.m2}(x,y)\text{end})
\]

\[
\text{meth } m_2 \equiv (\text{val } \text{arg}_1 : T_1; \text{res } \text{arg}_2 : T_2 \bullet c_2[\text{arg}_1, \text{arg}_2])
\]

\[
\text{end}
\]

The first step is to introduce the method \( m_3 \) defined by a parameterised command with a result parameter and body \( c_3 \).

\[
\text{meth } m_3 \equiv (\text{res } \text{arg} : T_1 \bullet c_3[\text{arg}])
\]

\[
\text{end}
\]

= Law[90] (method elimination) (r/l)

\[
\text{class } A \\
\text{ads_a;}
\]

\[
\text{meth } m_1 \equiv (\text{pds}_1 \bullet c_1; \\
\text{var } x, y : T_1, T_2 \bullet c_3[x]; \text{self.m2}(x,y)\text{end})
\]

\[
\text{meth } m_2 \equiv (\text{val } \text{arg}_1 : T_1; \text{res } \text{arg}_2 : T_2 \bullet c_2[\text{arg}_1, \text{arg}_2])
\]

\[
\text{meth } m_3 \equiv (\text{res } \text{arg} : T_1 \bullet c_3[\text{arg}])
\]

\[
\text{end}
\]

In method \( m_1 \), we eliminate the call to \( m_2 \) having \text{self} as target. This results in the parameterised command that defines the method \( m_2 \).

\[
\text{meth } m_1 \equiv (\text{pds}_1 \bullet c_1; \\
\text{var } x, y : T_1, T_2 \bullet c_3[x]; \\
(\text{val } \text{arg}_1 : T_1; \text{res } \text{arg}_2 : T_2 \bullet c_2[\text{arg}_1, \text{arg}_2])(x,y)\text{end})
\]

\[
\text{meth } m_2 \equiv (\text{val } \text{arg}_1 : T_1; \text{res } \text{arg}_2 : T_2 \bullet c_2[\text{arg}_1, \text{arg}_2])
\]

\[
\text{meth } m_3 \equiv (\text{res } \text{arg} : T_1 \bullet c_3[\text{arg}])
\]

\[
\text{end}
\]

= Lemma[1] (l/r)
= Law 61 (var dec separation) (l/r)

\[
\text{class } A \\
\text{ ads } a; \\
\text{ meth } m_1 \overset{\approx}{=} (pds_1 \cdot c_1; \\
\text{ var } y : T_2 \cdot \\
\text{ var } x : T_1 \cdot c_3[x]; \\
(\text{val arg}_1 : T_1; \text{ res arg}_2 : T_2 \cdot c_2[\text{arg}_1, \text{arg}_2])(x, y) \\
\text{ end } \\
\text{ end}) \\
\text{ meth } m_2 \overset{\approx}{=} (\text{val arg}_1 : T_1; \text{ res arg}_2 : T_2 \cdot c_2[\text{arg}_1, \text{arg}_2]) \\
\text{ meth } m_3 \overset{\approx}{=} (\text{res arg} : T_1 \cdot c_3[\text{arg}]) \\
\text{ end }
\]

From the command \( c_3 \) that appears inside the method \( m_1 \), we can introduce a call to the method \( m_3 \), passing \( x \) as a result argument: since there is no assignment to \( x \) before \( c_3 \), only the result of \( x \) in \( c_3 \) is of interest. If a value were assigned to \( x \), we should use both value and result mechanisms.

\( c_3[x] \)

= Lemma 10

\( (\text{res arg} : T_1c_3[\text{arg}])(x) \)

The method \( m_1 \) is now as follows.

\[
\text{ meth } m_1 \overset{\approx}{=} (pds_1 \cdot c_1; \\
\text{ var } y : T_2 \cdot \\
\text{ var } x : T_1 \cdot (\text{res arg} : T_1c_3[\text{arg}])(x); \\
(\text{val arg}_1 : T_1; \text{ res arg}_2 : T_2 \cdot c_2[\text{arg}_1, \text{arg}_2])(x, y) \\
\text{ end } \\
\text{ end})
\]

As inside \( m_1 \) appears the same parameterised command that defines the method \( m_3 \), we introduce a call to such method.
4.2. REFACTORING RULES

\[ \text{Lemma 1} \ (r/l) \]

\[
\text{meth } m_1 \triangleq \text{ (pds}_1 \bullet c_1; \\
\text{ var } y : T_2 \bullet \\
\text{ var } x : T_1 \bullet \text{ self.m}_3(x); \\
\text{ (val arg}_1 : T_1; \text{ res arg}_2 : T_2 \bullet c_2[\text{arg}_1, \text{arg}_2])(x, y) \\
\text{ end} \\
\text{ end) }
\]

We separate the declarations of the parameters \( \text{arg}_1 \) and \( \text{arg}_2 \) into two parameterised commands applied to the arguments \( x \) and \( y \).

\[ \text{Law 66} \ (\text{pcom merge}) \ (r/l) \]

\[
\text{meth } m_1 \triangleq \text{ (pds}_1 \bullet c_1; \\
\text{ var } y : T_2 \bullet \\
\text{ var } x : T_1 \bullet \text{ self.m}_3(x); \\
\text{ (val arg}_1 : T_1; \text{ (res arg}_2 : T_2 \bullet c_2[\text{arg}_1, \text{arg}_2])(y))(x) \\
\text{ end} \\
\text{ end) }
\]

We eliminate the parameterised command in which the command \( c_2 \) appears. With this elimination, the parameters \( \text{arg}_1 \) and \( \text{arg}_2 \) are replaced with the arguments \( x \) and \( y \), respectively.

\[ \text{Lemma 10} \ (r/l), \text{ Lemma 9} \ (r/l) \]

\[
\text{meth } m_1 \triangleq \text{ (pds}_1 \bullet c_1; \\
\text{ var } y : T_2 \bullet \\
\text{ var } x : T_1 \bullet \text{ self.m}_3(x); \\
\text{ c}_2[x, y] \\
\text{ end} \\
\text{ end) }
\]

In the class \( A \) we remove the method \( m_2 \). This is allowed by the fact the \( m_2 \) is no longer called inside or outside \( A \). Then we introduce a definition for a new method \( m_2 \).

\[ \text{Law 90} \ (\text{method elimination}) \ (l/r), \text{ Law 90} \ (\text{method elimination}) \ (r/l) \]

\[
\text{meth } m_2 \triangleq \text{ (res arg}_2 : T_2 \bullet \\
\text{ var } x : T_1 \bullet \text{ self.m}_3(x); \\
\text{ c}_2[x, \text{arg}_2] \\
\text{ end) }
\]
Notice that the above definition of $m_2$ is similar to the command that appears inside the block that declares the variable $y$ in the method $m_1$. We introduce a call to $m_2$ inside $m_1$. First we introduce a parameterised command from the command in the body of the block that declares $y$. This parameterised command is defined with a result parameter, and is applied to the argument $y$.

$$\text{meth } m_1 \triangleq (pds_1 \bullet c_1; \var y : T_2 \bullet (\var x : T_1 \bullet self.m_3(x); c_2[x, y] \text{ end}) \text{ end})$$

$= \text{ Lemma[10]} (1/r)$

$$\text{meth } m_1 \triangleq (pds_1 \bullet c_1; \var y : T_2 \bullet (\text{res arg}_2 : T_2 \bullet \var x : T_1 \bullet self.m_3(x); c_2[x, \text{arg}_2] \text{ end})(y) \text{ end})$$

As the parameterised command introduced is equal to the parameterised command that defines the method $m_2$, we introduce a call to such method.

$= \text{ Lemma[11]} (r/l)$

$$\text{meth } m_1 \triangleq (pds_1 \bullet c_1; \var y : T_2 \bullet \text{self.m}_2(y) \text{ end})$$

The final class $A$ is as follows.

$$\text{class } A$$
$$ads\_A;$$
$$\text{meth } m_1 \triangleq (pds_1 \bullet c_1; \var y : T_2 \bullet \text{self.m}_2(y) \text{ end})$$
$$\text{meth } m_2 \triangleq (\text{res arg}_2 : T_2 \bullet \var x : T_1 \bullet \text{self.m}_3(x); c_2[x, \text{arg}_2] \text{ end})$$
$$\text{meth } m_3 \triangleq (\text{res arg} : T_1 \bullet c_3[\text{arg}])$$
$$\text{end}$$

This finishes the derivation of rule $\langle \text{Replace Parameter with Method.} \rangle$.

This refactoring was proposed by Fowler [42, p. 292]. Our rule is more strict than Fowler’s, as we require the type of parameters to be basic ones. Also, we require not to exist calls to $m_2$ in the program, just inside class $A$. This avoids looking for calls to $m_2$ around the program to check if parameter $\text{arg}_1$ is used uniformly, i.e., if it is preceded by the command $c_3$. The strategy proposed by Fowler would lead to a contextual refactoring, in our terminology. As we require $m_2$ not to be called outside $A$, our refactoring is compositional.
4.2.6 Extract Class

A common practice in object-oriented software development is the partitioning of a class into several ones. This can be done by using the rule \langle Extract Class \rangle (Rule 4.6) [42, p. 149]. This rule creates a new class \( B \) with some attributes and methods from an original class \( A \), which ends with an attribute of type \( B \). In \( A \), original direct accesses to attributes that are now in \( B \) are replaced with calls to the get and set methods of \( B \). Original methods of \( A \) that act on the attributes that were moved to \( B \) are moved to \( B \) as well; in \( A \) we keep delegating methods: they just call the corresponding methods of \( B \).

To apply this rule, we require the types of parameters \( \text{arg} \) and those in \( \text{pds} \) to be basic. In an application from left to right, the attribute \( b \) cannot be already declared in \( A \), nor in its superclasses or subclasses. Also, attributes of \( A \), except \( x \), cannot be present in expressions \( \text{le} \) and \( \text{exp}_2 \), otherwise it would be necessary to pass \text{self} as argument. The attribute \( x \) of \( A \) is moved into \( B \) along with get and set methods. An attribute \( b \) of type \( B \) is introduced in class \( A \) and initialised in the \textbf{new} method with an object of \( B \). This attribute is the target of calls to the get and set methods of class \( B \). Original get and set methods that act on attributes that are declared in \( B \) must use the corresponding methods of class \( B \). In this way, there is no impact on clients of \( A \) that use the get and set methods. Methods declared in \( A \) that act on attributes present in \( B \) use the get and set methods present in \( A \) itself. The method \( m_1 \) acts only on the attribute \( x \), indicating that it is related to the concept introduced by the attribute \( x \). Therefore, it is provided by \( B \). On the right-hand side of the rule \langle Extract Class \rangle, the method \( m_1 \) of class \( B \) is similar to the definition of \( m_1 \) in \( A \), but the attribute \( x \) is read and written by means of get and set methods declared in \( B \) itself. In order to get the value of \( x \), we declare a variable that is passed as argument in the call to \text{getX}. We use \text{setX} to write to \( x \).

To apply this rule from left to right, the class \( B \), which is being extracted from class \( A \), cannot be declared in the sequence of class declarations \( \text{cds} \). In order to apply this rule from right to left, class \( B \) cannot be used in \( \text{cds} \) or \( c \): it is not superclass of any class in \( \text{cds} \), it is not type of any attribute, parameter, or local variable in \( \text{cds} \) or \( c \).

This refactoring improves reuse and extensibility. Classes should describe single concepts of the real world. Describing different concepts in one class mingles attributes and methods that are intrinsic to distinct concepts. This can result in complex classes that are hard to reuse and extend. Extracting a class from a complex class simplifies the original class and favours reuse and extensibility of the resulting classes. The refactoring rule we present here is similar to Fowler’s refactoring for class extraction [42, p.149].

Derivation. The starting point for the derivation is the left-hand side of this rule. Extracting a class from an existing one is equivalent to moving attributes and methods necessary to describe a concept that is different from that presented in the original class into a new class. Initially we have just the class \( A \).
### Rule 4.6 (Extract Class)

<table>
<thead>
<tr>
<th>A \text{ extends } C</th>
<th>\text{class } A \text{ extends } C</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{pri } x : T; \text{ ads}_a;</td>
<td>\text{pri } b : B; \text{ ads}_a;</td>
</tr>
<tr>
<td>\text{meth } \text{getX } \cong \begin{align*} (\text{res } \text{arg} : T \bullet \text{arg} := \text{self}.x) \end{align*}</td>
<td>\text{meth } \text{getX } \cong \begin{align*} (\text{res } \text{arg} : T \bullet \text{self}.b.\text{getX}(\text{arg}) \end{align*}</td>
</tr>
<tr>
<td>\text{meth } \text{setX } \cong \begin{align*} (\text{val } \text{arg} : T \bullet \text{self}.x := \text{arg}) \end{align*}</td>
<td>\text{meth } \text{setX } \cong \begin{align*} (\text{val } \text{arg} : T \bullet \text{self}.b.\text{setX}(\text{arg}) \end{align*}</td>
</tr>
<tr>
<td>\text{meth } m_1 \cong \text{pds} \bullet \begin{align*} c[\text{le} := \exp_1[\text{self}.x], \text{self}.x := \exp_2, \alpha(\text{pds})]] \end{align*}</td>
<td>\text{meth } m_1 \cong \begin{align*} (\text{pds} \bullet \text{self}.b.m_1(\alpha(\text{pds}))) \end{align*}</td>
</tr>
<tr>
<td>\text{mts}_a</td>
<td>\text{new } \cong \text{self}.b := \text{new } B() \end{align*}</td>
</tr>
<tr>
<td>\text{end}</td>
<td>\text{mts}_a' \end{align*}</td>
</tr>
</tbody>
</table>

where

\[
\text{mts}_a' \cong \text{mts}_a[\text{var aux} : T \bullet \text{self}.b.\text{getX}(\text{aux}); \text{le} := \exp[\text{aux}] \text{ end}, \text{self}.b.\text{setX}(\exp_2), \alpha(\text{pds})]] \end{align*}

\[
\text{le} := \exp[\text{self}.x], \text{self}.x := \exp\]

provided

(\leftrightarrow) The types of parameters \text{arg} and those in \text{pds} are basic;

(\rightarrow) The class \text{B} is not declared in \text{cds}:

- The attribute \text{b} is not declared in \text{ads}_a nor in any subclass or superclass of \text{A}.
- Attributes of \text{A}, except \text{x}, cannot be present in \text{le} or \text{exp}_2;

(\leftarrow) The class \text{B} is not used in \text{cds} or \text{c}

---

As we have done in the derivation of rule \langle Move Method \rangle (Rule 4.2), we do not deal with arbitrary parameters \text{pds}, but with clearly defined parameters. For method \text{m}_1, we define parameters
4.2. REFACTORING RULES

val arg₁ : T₁; res arg₂ : T₂.

class A
  pri x : T; adsₐ;
  meth getX ≡ (res arg : T • arg := self.x)
  meth setX ≡ (val arg : T • self.x := arg)
  meth m₁ ≡ (val arg₁ : T₁; res arg₂ : T₂ • c₁[le := exp₁[self.x], self.x := exp₂, arg₁, arg₂])
mtsₐ
end

We introduce an empty class B to which we move attributes and methods.

= Law \text{82 (class elimination)} (r/l)

class A
  pri x : T; adsₐ;
  meth getX ≡ (res arg : T • arg := self.x)
  meth setX ≡ (val arg : T • self.x := arg)
  meth m₁ ≡ (val arg₁ : T₁; res arg₂ : T₂ • c₁[le := exp₁[self.x], self.x := exp₂, arg₁, arg₂])
mtsₐ
end class B
end

By applying rule \text{(Move Attribute)} (Rule \text{4.3}), we move the attribute x from A to B, in which get and set methods are declared. In A, an attribute b of class B is declared and initialised. Occurrences of previous accesses to x are replaced by calls to get and set methods of class B, having b as call target.

= (Move Attribute)(Rule \text{4.3})(1/r)

class A
  pri b : B; adsₐ;
  meth getX ≡ (res arg : T • b.getX(arg))
  meth setX ≡ (val arg : T • b.setX(arg))
  meth m₁ ≡ (val arg₁ : T₁; res arg₂ : T₂ •
              c₁[var aux : T • self.b.getX(aux); le := exp[aux] end,
              self.b.setX(exp, arg₁, arg₂)]
  new ≡ self.b := new B()
mtsₐ
end
class B
  pri x : T;
  meth getX ≡ (res arg : T • arg := self.x)
  meth setX ≡ (val arg : T • self.x := arg)
end

Finally, we move the method \(m_1\) from \(A\) to \(B\), since this method essentially uses get and set methods of \(B\) in order to indirectly access the attribute \(x\).

\[\langle \text{Move Method} \rangle \text{ (Rule 4.2)}(l/r)\]

class A
  pri b : A; ads_a;
  meth getX ≡ (res arg : T • self.b.getX(arg))
  meth setX ≡ (val arg : T • self.b.setX(arg))
  meth \(m_1\) ≡ (val arg1 : T_1; res arg2 : T_2 • self.b.m1(arg1, arg2))
  new ≡ self.b := new B()
  mts_a
end

class B
  pri x : T;
  meth getX ≡ (res arg : T • arg := self.x)
  meth setX ≡ (val arg : T • self.x := arg)
  meth \(m_1\) ≡ (val arg1 : T_1; res arg2 : T_2 •
    \(c_1[\text{var } aux : T • \text{self}.getX(aux); le := exp_1[aux]} \) end,
    self.setX(exp_2), arg_1, arg_2)
end

This finishes the derivation of the rule \(\langle \text{Extract Class} \rangle\). \(\square\)

The derivation was based on the application of rules \(\langle \text{Move Attribute} \rangle\) (Rule 4.3) and \(\langle \text{Move Method} \rangle\) (Rule 4.2), a derivation strategy similar to the strategy proposed by Fowler [42]. In this way, we have just considered one attribute and one method; the derivation of a rule that considers an arbitrary number of attributes with get and set methods, as well as an arbitrary number of methods that must be moved to a newly extracted class, is similar to the one we presented here. Our rule presents a limitation when requiring the type of parameters to be basic, because one of the laws used for the introduction of parameterised commands requires variables not to be method call targets. These variables in a latter derivation step are changed to be arguments. There is no sense in passing to a method an object that is not a method call target.

The similarity between \(\langle \text{Extract Class} \rangle\) and \(\langle \text{Move Attribute} \rangle\) comes from the view that data (attributes) determine abstractions to be modelled by classes. Consequently, as we are defining
4.3. NEW REFACTORINGS

a new class, the attributes that we perceive to describe another class must be moved from their current class to such new class.

4.3 New refactorings

In this section we present two new refactorings which do not appear in refactoring catalogs [69, 42]. They deal basically with the relationship between a class and its clients.

4.3.1 Clientship Elimination

For the derivation of some refactoring rules, it is necessary to eliminate a clientship relation between two classes. This is useful when it is necessary to change the client of a class, and it is the aim of the refactoring rule ⟨Clientship Elimination⟩ (Rule 4.7).

On the left-hand side of this rule, class A is a client of B; it declares the attribute b of type B and initialise it with an object of B. In method m of A, we assume that there are occurrences of calls to a method n of B. The class B declares the attribute x of type T and attributes ads_b. In method n of B, there might be occurrences of the expression self.x.

On the right-hand side, the class A does not declare an attribute of type B, but the attribute x that is also declared in B. The method n is also declared in A, and the call to the method n of B, that occurred in method m on the left-hand side, is replaced with a call to n on self.

To apply this rule, super must not appear in method n that is called inside A, otherwise we could not move such method. We require that b is not null or error avoiding program abortion. Also, method n of class B must not access any attribute declared in ads_b. To apply this rule from left to right, we require that n refers only to attribute x. This is because we cannot establish a coupling invariant between inherited attributes of a class and attributes of its clients. We also require that inside n there are no calls to methods declared in mts_b. There must be no references to attribute b in methods in mts_a, otherwise we cannot remove it from class A. The method m of A calls the method n of B with arguments represented by a_n and b as target. The method n must not be declared in mts_a nor in any subclass or superclass of A. Similarly, attribute x must not be declared in ads_a nor in any superclass or subclass of A. Methods in mts_a must not refer to the attribute b.

In order to apply this rule from right to left, b must not be declared in ads_a nor in any subclass or superclass of A. As we are interested in the refinement of the method m, we assume that methods in mts_a do not refer to the attribute x. The method n must be called only inside method m, because it will be removed from A. Finally, the method n of class A must not refer to any attribute in ads_a.

Derivation. Below we present the derivation of ⟨Clientship Elimination⟩, from left to right, i.e., in the direction of elimination of the client relation between A and B.
CHAPTER 4. COMPOSITIONAL REFACTORINGS

Rule 4.7 (Clientship Elimination)

provided

(←)  
(1) super does not appear in n; (2) \( b \neq \text{null} \land b \neq \text{error} \) is an invariant of A;
(3) self.b does not appear in n of class B, for any attribute b that is declared in ads\( _a \);

(→)  
(1) self.a does not appear in n, for any public or protected attribute a that is declared by D or by any of its superclasses; (2) self.p does not appear in n, for any method p declared in mts\( _b \) or in any superclasses of B; (3) self.b does not appear in mts\( _a \); (4) n is not declared in mts\( _a \) nor in any subclass or superclass of A in eds; (5) x is not declared in ads\( _a \) nor in any superclass or subclass of A;

(←)  
(1) b is not declared in ads\( _a \) nor in any superclass or subclass of A; (2) self.x does not appear in mts\( _a \); (3) D.n does not appear in mts\( _a \), eds or c, for any D such that D \( \leq \) A; (4) self.a, for any a in ads\( _a \), does not appear in method n of class A.

First, we apply Law 85 (change visibility: from private to public), from left to right, to change the visibility of attribute x of class B. Then, we eliminate the call to method n of class B that
occurs inside method $m$ of $A$. This yields a sequential composition of an assumption about $\texttt{self}.b$, the target of the method call, and the parameterised command that defines the method $n$ in which occurrences of $\texttt{self}$ are replaced with $\texttt{self}.b$. Every access to $\texttt{self}.x$ that appears inside the parameterised command that defines the method $n$ is replaced with $\texttt{self}.b.x$. This parameterised command is applied to the arguments represented by $a_n$.

$$= \text{Law } 85 \langle \text{change visibility: from private to public} \rangle \ (l/r), \text{ Law } 95 \langle \text{method call elimination} \rangle \ (l/r)$$

\[
\begin{align*}
\text{class } A \text{ extends } C & \quad \text{class } B \\
\text{pri } b : B; \ ads_a & \quad \text{pub } x : T; \ ads_b \\
\text{meth } m \equiv (pds_m \bullet c_m) \{\texttt{self}.b \neq \texttt{null} \land \texttt{self}.b \neq \texttt{error}\} (pds_m \bullet c_m)[\texttt{self}.b/\texttt{self}](a_n)) & \quad \text{meth } n \equiv (pds_n \bullet c_n) \\
\text{mts}_a & \quad \text{mts}_b \\
\text{new } \hat{=} \texttt{self}.b := \texttt{new } B & \quad \text{end}
\end{align*}
\]

Now we prepare the class $A$ for a class refinement. We apply the Law 78 $\langle$simple specification$\rangle$ exhaustively to assignments of the form $t := \texttt{self}.b$, transforming them into the specification statement $t : [\texttt{true}, t = \texttt{self}.b]$. As one condition for applying $\langle$Clientship Elimination$\rangle$, from left to right, is that $\texttt{self}.b$ does not appear in methods in $\text{mts}_a$, these methods are not changed by the application of the Law 78 $\langle$simple specification$\rangle$. So, only the method $m$ is changed; its new body is denoted by writing $c_m^{\text{spec}}$.

$$= \text{Law } 78 \langle \text{simple specification} \rangle \ (r/l)$$

\[
\begin{align*}
\text{class } A \text{ extends } C & \quad \text{class } B \\
\text{pri } b : B; \ ads_a & \quad \text{pub } x : T; \ ads_b \\
\text{meth } m \equiv (pds_m \bullet c_m^{\text{spec}}) \{\texttt{self}.b \neq \texttt{null} \land \texttt{self}.b \neq \texttt{error}\} (pds_m \bullet c_m)[\texttt{self}.b/\texttt{self}](a_n)) & \quad \text{end} \\
\text{mts}_a & \quad \text{mts}_b \\
\text{new } \hat{=} \texttt{self}.b := \texttt{new } B & \quad \text{end}
\end{align*}
\]

We then apply the Law 108 $\langle$private attribute-coupling invariant$\rangle$, introducing the attribute $x$ in $A$. The coupling invariant $CI$, relating $x$ with the attribute $x$ of class $B$, is $\texttt{self}.x = \texttt{self}.b.x$. It states that the new attribute $x$ in $A$ represents the attribute $x$ of class $B$. In terms of data refinement, the new attribute $x$ is a concrete variable.

The application of $CI$ changes guards and commands in the methods $m$ and in those in $\text{mts}_a$. In fact, the application of $CI$ to $\text{mts}_a$ does not affect such methods because one of the conditions of the rule $\langle$Clientship Elimination$\rangle$ is that $\texttt{self}.b$ does not appear in $\text{mts}_a$. Consequently,
only the method \( m \) is modified. For this reason, in what follows, instead of writing \( CI(mts_a) \), we write just \( mts_a \). Guards and commands of the method \( m \) are changed according to the laws of data refinement. Guards are augmented to assume the coupling invariant. Each new guard can be just the conjunction of the old guard with the coupling invariant. In specification statements, the frame is expanded with the concrete variable and the coupling invariant is conjoined with the existing pre- and postconditions. The specification statement \( t : [\text{true}, t = \text{self}.b.x] \) becomes \( x, t : [CI, t = \text{self}.b.x \land CI] \). Assignments of the form \( \text{self}.b.x := \text{exp} \) are augmented to \( \text{self}.x, \text{self}.b.x := \text{exp}, \text{exp} \).

\[ \preceq \text{Law 108 (private attribute-coupling invariant)} \]

\[
CI \triangleq \text{self}.x = \text{self}.b.x
\]

\[
\text{class } A \text{ extends } C
\]

\[
\begin{align*}
\text{pri } x & : T; \\
\text{pri } b & : B; \text{ ads}_a \\
\text{meth } m & \triangleq (pds_m \cdot CI(c_{spec}^{m}[\{\text{self}.b \neq \text{null} \land \text{self}.b \neq \text{error}\} \land CI]) \\
& \quad (pds_n \cdot c_n)[\text{self}.b/self[\text{self}(a_n)]) \\
mts_a & \\
\text{new } \triangleq \text{self}.b := \text{new } B
\end{align*}
\]

The next step is the elimination of occurrences of \( \text{self}.b.x \) in the method \( m \). Guards must be algorithmically refined. Specification statements like \( x, t : [CI, t = \text{self}.b.x \land CI] \) are refined, by using Law 69 (assignment), to the assignment \( t := \text{self}.x \). An assignment \( \text{self}.x, \text{self}.b.x := \text{exp}, \text{exp} \) becomes \( \text{self}.x := \text{exp} \). We write \( c_m' \) to represent the command \( c_m \) of the method \( m \) in which we replace occurrences of \( \text{self}.b.x \) with \( \text{self}.x \). For this reason, now we do not write the substitution \( [\text{self}.b/self] \) for the parameterised command \( (pds_n \cdot c_n) \).

\[
\text{class } A \text{ extends } C
\]

\[
\begin{align*}
\text{pri } x & : T; \\
\text{pri } b & : B; \text{ ads}_a \\
\text{meth } m & \triangleq (pds_m \cdot c_m')[\{\text{self}.b \neq \text{null} \land \text{self}.b \neq \text{error}\}(pds_n \cdot c_n)(a_n)] \\
mts_a & \\
\text{new } \triangleq \text{self}.b := \text{new } B
\end{align*}
\]

The next step is to remove the assumption about the attribute \( b \) that appears in the method \( m \). This is possible since a proviso in the rule already states that \( b \neq \text{null} \) and \( b \neq \text{error} \). We finish with just one reference to such an attribute, the one in the method \( \text{new} \).
4.3. NEW REFACTORINGS

⊆ Law [77] (remove assumption), Law [16] (skip unit)

class A extends C
pri x : T;
pri b : B; ads_a
meth m ≜ (pds_m • c'_m[(pds_n • c_n)(a_n)])
mts_a
new ≜ self.b := new B
end

We use the parameterised command (pds_n • c_n) to define a new method in A. We name such method as n.

= Law [90] (method elimination) (r/l)

class A extends C
pri x : T;
pri b : B; ads_a
meth m ≜ (pds_m • c'_m[(pds_n • c_n)(a_n)])
meth n ≜ (pds_n • c_n)
lmtnsa
new ≜ self.b := new B
end

As the parameterised command that defines the method n is applied in the command c_m to the arguments a_n, we can introduce a call to n in the command c_m.

= Lemma [1] (r/l)

class A extends C
pri x : T;
pri b : B; ads_a
meth m ≜ (pds_m • c'_m[exp_n])
meth n ≜ (pds_n • c_n)
lmtnsa
new ≜ self.b := new B
end

We remove the method new by applying Law [90] (method elimination), from left to right. Then, we apply Law [83] (attribute elimination), from left to right, for removing the attribute b from class A.
Law \(90\) \textit{(method elimination)} (l/r), Law \(83\) \textit{(attribute elimination)} (l/r)

\begin{verbatim}
class A extends C
  pri x : T;.ads
  meth m ≜ (pds_m • c_m′[self.n(exp_n)])
  meth n ≜ (pds_n • c_n)
  mts_a
end
\end{verbatim}

Finally, we apply Law \(85\) \textit{(change visibility: from private to public)}, from right to left, changing the visibility of attribute \(x\) of class \(B\) back to private. This finishes the derivation of the refactoring rule \(\textit{(Clientship Elimination)}\), from left to right. The proof of the reverse direction is similar and also constitutes a refinement between the programs involved. Because both sides are refinement of each other, we conclude that they are equal.

Notice that this rule is only valid in a language with a copy semantics. When we apply this rule from left to right, we copy an attribute an the method that accesses such attribute to another class. In the class to which we copied the attribute and the method, there is no assignment involving the attribute in this class and the one of the source class. So, we have these attributes refer to different objects.

For simplicity, in this rule we considered just one attribute and one method. However, clients may call distinct methods that use different attributes. In this situation, we should follow the same steps for the proof. After eliminating method calls and preparing client classes for data refinement, we introduce new attributes that are related with those attributes used in the methods called. After this, we continue with data refinement and introduction of method calls. In the case of inherited attributes, its is not possible, for now, to couple such attributes with attributes of clients of the class that inherits the attributes by means of an invariant.

This rule, in fact, does not appear in any of the existing list of refactorings. Indeed, it is easy to argue that it does not lead to an improvement of design of code; eliminating clientship is not a good practice, since we can possibly mix concepts described in different classes. On the other hand, introducing clientship justifies this rule as a refactoring, even though finding a class that has similar attributes and methods to another one is not so easy in practice. Moreover, this rule is justified as a step in the derivation of other rules. From this point of view, we can classify it to be a transformational rule: it preserves behaviour.

### 4.3.2 Delegation Elimination

A particular case of Rule 4.7 is the rule \(\textit{(Delegation Elimination)}\) (Rule 4.8). On the left-hand side of this rule, class \(A\) is a delegating class. Any call to the method \(m\) of class \(A\) is forwarded to the class \(B\). The class \(A\) also declares the attribute \(b\) of type \(B\) and initialises it with an object of \(B\).
4.4 Further Compositional Refactoring Rules

In this section we present refactoring rules related with commands, without their corresponding derivations. For their derivations see Appendix A.

Rule 4.8 (Delegation Elimination)

Given the following classes:

- \texttt{class A extends C}
- \texttt{class B extends D}

The method \texttt{n} of class \texttt{B} declares the attribute \(x\) of type \(T\) and attributes \(ads_b\). In the method \(n\) of \(B\) there might be occurrences of the expression \(\text{self}.x\).

On the right-hand side, the class \(A\) does not declare an attribute of type \(B\), but the attribute \(x\) that is also declared in \(B\). The method \(m\) of \(A\) is defined by the same parameterised command that defines the method \(n\) of \(B\).

Notice that in Rule 4.7, we require, for application from right to left, that method \(n\) of class \(A\) is not called in the program. This condition is not necessary for Rule 4.8, as we do not remove any method in class \(A\) when applying this law from right to left.

The class \(B\) declares the attribute \(x\) of type \(T\) and attributes \(ads_b\). In the method \(n\) of \(B\) there might be occurrences of the expression \(\text{self}.x\).

On the right-hand side, the class \(A\) does not declare an attribute of type \(B\), but the attribute \(x\) that is also declared in \(B\). The method \(m\) of \(A\) is defined by the same parameterised command that defines the method \(n\) of \(B\).

Notice that in Rule 4.7, we require, for application from right to left, that method \(n\) of class \(A\) is not called in the program. This condition is not necessary for Rule 4.8, as we do not remove any method in class \(A\) when applying this law from right to left.
Rule 4.9 (Inline Class)

\begin{align*}
\text{class } & A \text{ extends } C \\
\text{pri } & b : B; \text{.ads}_a; \\
\text{meth } & \text{getX } \triangleq \\
& (\text{res } \text{arg} : T \bullet \text{self.b.getX(\text{arg}))} \\
\text{meth } & \text{setX } \triangleq \\
& (\text{val } \text{arg} : T \bullet \text{self.b.setX(\text{arg}))} \\
\text{meth } & m \triangleq (\text{pds} \bullet) \\
& \text{self.b.m(\alpha(\text{pds}))} \\
\text{new } & \triangleq \text{self.b := new } B(); \\
\text{mts}_a \end{align*}

\begin{align*}
\subseteq_{cds, c}
\text{class } & A \text{ extends } C \\
\text{pri } & x : T; \text{ads}_a; \\
\text{meth } & \text{getX } \triangleq \\
& (\text{res } \text{arg} : T \bullet \text{arg} := \text{self.x}) \\
\text{meth } & \text{setX } \triangleq \\
& (\text{val } \text{arg} : T \bullet \text{self.x := arg'}) \\
\text{meth } & m \triangleq (\text{pds} \bullet) \\
& c_m[\text{var aux} : T \bullet \text{self.getX(aux);} \\
& \text{le := exp1[aux]end,} \\
& \text{self.setX(exp2, } \alpha(\text{pds}))] \\
\text{end} \end{align*}

provided

1. \( b \neq \text{null} \land b \neq \text{error} \) is an invariant of \( A \); 2. The types of parameters \( \text{arg} \) and those in \( \text{pds} \) are basic; 3. \( x \) is not declared \( \text{ads}_a \), in any of its superclasses or subclasses; 4. \( \text{self.b} \) does not appear in \( \text{mts}_a \), except in target of method calls; 5. \( n \) is not declared in \( \text{mts}_a \), in any subclass or superclass of \( A \).

4.4.1 Inline Class

The purpose of a class in a system must be well-justified. When the role of a class is not clear, the first step in order to remove it is to inline it: all clientship relations with respect to such class are eliminated from the whole program. This is the purpose of the rule (Inline Class) (Rule 4.9). If the class that is inlined is not used as a type of any kind of variable in the program nor it is a superclass of any class, we can remove it. In this case, this rule is exactly the reverse of
4.4. FURTHER COMPOSITIONAL REFACTORING RULES

rule \langle Extract Class \rangle.

The class $A$ on the left-hand side of this rule is a client of $B$. It declares the attribute $b$ of type $B$. The methods $getX$ and $setX$ just call the corresponding methods of class $B$. Also, the method $m$ just calls the method $m$ of $B$. The class $B$ declares an attribute $x$ of type $T$ along with get and set methods for it. Also, $B$ declares a method $m$ in which accesses to $x$ are carried out by means of the get and set methods. We assume that all other clients of $B$ are in the sequence of class declaration denoted by $cds_1$. This sequence is empty if $A$ is the only client of $B$. The sequence denoted by $cds_2$ contains classes that are not clients of $B$.

On the right-hand side of this rule, the class $A$ declares an attribute $x$ of type $T$. The get and set methods directly use this attribute instead of calling get and set methods of $B$. The method $m$ of $A$ now, instead of calling the method $m$ of $B$, has a copy of the command $c_m$ that is present in the method $m$ of $B$. However, accesses to the attribute $x$ are direct, and not by means of get and set methods.

To apply this law, we require that $b$ is not null or error avoiding program abortion. Also, parameters $arg$ and those in $pds$ must have basic types. The attribute $x$ must not be declared in $A$ nor in any of its subclasses or superclasses. As we want to remove the attribute $b$ from $A$ and other clients of $B$, the expression $self.b$ must not appear in $mts_a$, except as method call target. The method $m$ of $B$ that is copied to $A$ must not be declared in $mts_a$, in any subclass or superclass of $A$.

In the refactoring \langle Inline Class \rangle presented by Fowler, after eliminating clientship, if $B$ is not used as type of any local variable in the whole program, it is not the type of any method parameter nor superclass of any class in $cds$, we can eliminate $B$. In this case, the refactoring rule \langle Inline Class \rangle is the inverse of rule \langle Extract Class \rangle (Rule 4.6). Another difference between our rule and Fowler’s refactoring is that we restrict parameters to have basic types.

4.4.2 Self Encapsulate Field

Accessing attributes of a superclass from a subclass is only allowed if the attributes are declared as protected or public. In order to allow subclasses—and all other classes—to have access to private attributes already declared in a superclass, get and set methods have to be declared. The rule \langle Self Encapsulate Field \rangle (Rule 4.10) introduces get and set methods for an attribute $x$ declared in the class $A$.

In method $m_1$ on the left-hand side of the rule, the attribute $x$ appears in the expression $exp_1$ and there is also an assignment to this attribute. On the right-hand side of the rule, the occurrence of $self.x$ in expression $exp_1$ is replaced by the local variable $aux$ declared in $m_1$. This variable receives the result of the call to method $getX$. The assignment is accomplished by a call to method $setX$, passing by value the expression $exp_2$. These changes also occur in $mts_a$.

To apply this rule from left to right, the methods $getX$ and $setX$ cannot be declared in the superclass of $A$, in $A$ itself, nor in any of its subclasses. To apply this rule in the reverse direction,
RULE 4.10 (Self Encapsulate Field)

\[
\begin{align*}
\text{class } A & \text{ extends } C \\
\text{pri } x : T; \ ad_{a} \\
\text{meth } m_{1} & \triangleq (p_{d_{1}} \bullet \\
& c_{1}[l_{1} := \text{exp}_{1}[\text{self}.x], \\
& \text{self}.x := \text{exp}_{2}]) \\
\text{mts}_{a} & \end{align*}
\]

where
\[
\begin{align*}
c_{1}' & \triangleq c_{1}[\text{var } \text{aux} : T; \bullet \text{self}.getX(\text{aux})]; \ l_{1} := \text{exp}_{1}[\text{aux}] \end{align*}
\]

\[
\begin{align*}
\text{mts}_{a}' & \triangleq \text{mts}_{a}[\text{var } \text{aux} : T; \bullet \text{self}.getX(\text{aux})]; \ l_{1} := \text{exp}_{1}[\text{aux}] \\
\end{align*}
\]

provided
\[
\begin{align*}
(\rightarrow) & \text{getX is not declared in any superclass or subclass of } A \text{ in } \text{cds}; \\
\text{setX is not declared in any superclass or subclass of } A \text{ in } \text{cds}; \\
(\leftarrow) & \text{le}.getX \text{ and } \text{le}.setX \text{ do not appear in } m_{t_{a}}', \text{cds} or c, \text{ for any } le \text{ such that } le \leq A.
\]

the methods getX and setX cannot be called in cd, c, or A.

In this rule, we considered just one attribute. To encapsulate fields inside a class implies in the application of this rule the same number of times as the number of attributes to be encapsulated.

4.4.3 Decompose Conditionals

Conditionals are one of the causes of long methods. It is possible to write conditionals to do various tasks depending on different conditions, which in ROOL are written as guards. This leads to long methods, decreasing legibility. The refactoring rule (Decompose Conditional) (Rule 4.11) simplifies a conditional by extracting methods for each guarded command. Free variables that appear in these commands must be passed as arguments in the method calls. Also, the boolean guards are extracted into a new method that is called before reaching the conditional. In ROOL, we use the term alternation for conditionals.

On the left-hand of this rule, the alternation that appears inside the method \( m_{1} \) presents two branches. One of the branches is guarded by the expression \( \text{exp} \) and the corresponding guarded command is \( c_{m_{1}}[a_{i}] \), in which free variables represented by \( a_{i} \) may appear. The other branch is guarded by the expression \( \neg \text{exp} \); the corresponding guarded command in this branch is \( c_{m_{1}}'[a_{j}] \) in which the free variables \( a_{j} \) may appear.
4.4. FURTHER COMPOSITIONAL REFACTORING RULES

Rule 4.11 (Decompose Conditional)

<table>
<thead>
<tr>
<th>class A</th>
</tr>
</thead>
<tbody>
<tr>
<td>ads_a</td>
</tr>
<tr>
<td>meth m_1 \equiv (pds_1 \bullet</td>
</tr>
<tr>
<td>[ \neg exp \rightarrow c'_{m_1}[a_j] ]</td>
</tr>
<tr>
<td>fi)</td>
</tr>
<tr>
<td>mts_a</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>class A</th>
</tr>
</thead>
<tbody>
<tr>
<td>ads_a</td>
</tr>
<tr>
<td>meth m_1 \equiv (pds_1 \bullet</td>
</tr>
<tr>
<td>\var b : bool \bullet self.m_2(b);</td>
</tr>
<tr>
<td>if b \rightarrow self.m_3(a_i)</td>
</tr>
<tr>
<td>[ \neg b \rightarrow self.m_4(a_j) ]</td>
</tr>
<tr>
<td>fi</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>meth m_2 \equiv (res \ arg : bool \bullet \ arg := exp)</td>
</tr>
<tr>
<td>meth m_3 \equiv (pds_3 \bullet c_{m_1})</td>
</tr>
<tr>
<td>meth m_4 \equiv (pds_4 \bullet c'_{m_1})</td>
</tr>
<tr>
<td>mts_a</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>

provided

(\leftarrow) \quad \bullet a_i and a_j have basic types;

(\rightarrow) \quad a_i \subseteq FV(c_m) and a_j \subseteq FV(c'_m);

m_2 is not declared in mts_a nor in any superclass or subclass of A;

m_3 is not declared in mts_a nor in any superclass or subclass of A;

m_4 is not declared in mts_a nor in any superclass or subclass of A;

(\leftarrow) \quad N.m_2, N.m_3, and N.m_4 do not appear in mts_a, cds or c, for any N \leq A;

On the right-hand side, we declare in the method m_1 the boolean variable b. We also introduce in the class A the method m_2 with a result parameter of type bool. To such a parameter is assigned the value of the expression exp that appears in one of the branches of the alternation on the left-hand side. We also introduce the methods m_3 and m_4. The method m_3 is defined by a parameterised command with parameters pds that appear in the command c_{m_1}. This command appears in the alternation on the left-hand side guarded by the expression exp. Similarly, the method m_4 is defined by a parameterised command with the command c'_{m_1}.

In the method m_1 on the right-hand side, the variable b holds the result of the call to the method m_2, which is a boolean value corresponding to the evaluation of the expression exp. In the guards, the variable b replaces the expression exp. The guarded commands are substituted with calls to methods m_3 and m_4. Free variable that appeared in the commands c_{m_1} and c'_{m_1} may be passed as arguments in the method call.
### Rule 4.12 (Introduce Explaining Variable)

\[ \text{cds, } C \triangleright c[e] = \text{cds, } C \triangleright \textbf{var } x : T \cdot x := e; c[x] \textbf{ end} \]

provided

\[ (\leftrightarrow) \text{cds, } C \triangleright e : T \]

\(x\) is not free in \(c\) and \(e\)

\(c\) does not assign free variables in \(e\)

### Rule 4.13 (Consolidate Conditional Expressions)

\[ \text{cds, } C \triangleright \textbf{if } \psi_1 \rightarrow c [] \psi_2 \rightarrow c [] (\psi_i \rightarrow c_i) \textbf{ fi} = \textbf{if } (\psi_1 \lor \psi_2) \rightarrow c [] ([] i \cdot \psi_i \rightarrow c_i) \textbf{ fi} \]

provided

\[ (\leftrightarrow) i \text{ ranges over } 3..n \]

To apply this rule, the variables \(a_i\) and \(a_j\) must have basic type. In order to apply this rule from left to right, the methods \(m_2, m_3\) and \(m_4\) must not be declared in \(A\) nor in any of its subclasses or superclasses. The variables \(a_i\) and \(a_j\) represent free variables that may appear in the commands \(c_{m_1}\) and \(c'_{m_1}\), respectively. Their types are given by \(T_3\) and \(T_4\), respectively. To apply this rule from right to left, the methods \(m_2, m_3\) and \(m_4\) must not be called inside \(A\) nor in \(cds\) or \(c\).

### 4.4.4 Introduce Explaining Variable

In order to simplify an expression and make clear its use, we can introduce a variable and assign to it the whole expression. This improves the readability of a program due to the simplification of the expressions and the use of a variable which should have a name that explains the meaning of the expression. We model this refactoring by using local (ordinary) variables (Rule 4.12), because constant declarations like Java’s `final` is not available in ROOL.

On the left-hand side of this rule, in class \(C\), we have the command \(c\) in which the expression \(exp\) appears. On the right-hand side, we declare the variable \(x\) to which we assign the expression \(exp\); the variable \(x\) replaces the occurrence of \(exp\) in the command \(c\).

In order to apply this rule, the variable \(x\) must not occur free in the command \(c\) nor in the expression \(e\). We assume that the type of \(e\) is \(T\).

### 4.4.5 Consolidate Conditional Expression

An alternation that presents different guards whose corresponding guarded commands are the same can be consolidated by combining the guards using the logic connective or. This is expressed in

\(^1\)This modifier establishes that the variable value cannot be changed after object initialisation.
4.4. Further Compositional Refactoring Rules

<table>
<thead>
<tr>
<th>Rule 4.14 (Consolidate Duplicate Conditional Fragments)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c_{ds}, C \Rightarrow \text{if } [i \bullet \psi_i \rightarrow (c_i; c)] = \text{if } [i \bullet \psi_i \rightarrow c_i \text{ fi}; c] ) provided</td>
</tr>
</tbody>
</table>

\[ (\leftrightarrow) \quad i \text{ ranges over } 1..n \]

the Rule 4.13.

On the left-hand side of this rule, command \( c \) appears in the branches guarded by \( \psi_1 \) and \( \psi_2 \). On the right-hand side, these guards appears as a disjunction in one of the branches whose guarded command is \( c \). In fact, this refactoring corresponds to the application of the Law 8 (\( \text{if} \lor \text{distrib1} \)) of ROOL (presented in Appendix D), which joins two branches of the same guarded command.

4.4.6 Consolidate Duplicate Conditional Fragments

When a command appears as the last command of every branch of an alternation, we can remove such a command from the branches of the alternation and place it just after the alternation. This is the purpose of the rule (Consolidate Duplicate Conditional Fragments) (Rule 4.14).

On the left-hand side of this rule, command \( c \) appears as the last command of every branch of the alternation. On the right-hand side, command \( c \) appears just after the alternation. In fact, this refactoring corresponds to the application of the Law 20 (\( ; \text{-if left dist} \)) of ROOL, which deals with the distribution of a command over an alternation, from right to left.

4.4.7 Substitute Algorithm

Replacing an algorithm with another one is a refactoring that aims at making a program better, since we are expected to introduce a more efficient algorithm. This is the practical view of algorithm substitution. In ROOL, besides this view, substitution of algorithms can be done with the intention of refining a program in order to reduce nondeterminism, for instance. So, an algorithm that is a refinement of the original one takes its place. The refactoring (Substitute Algorithm) is presented in Rule 4.15; it relies on the simple property of monotonicity of parameterised commands with respect to command refinement.

On the left-hand side, the method \( m_1 \) is defined by the parameterised command \( (pds \bullet c_1) \). On the right-hand side, the same method is defined by the parameterised command \( (pds \bullet c'_1) \); we require \( c'_1 \) to refine \( c_1 \).

We restricted this refactoring to deal only with methods, but the refinement relation between commands \( c \sqsubseteq c' \) in ROOL allows us to substitute an algorithm in any context. Methods are a particular case of algorithm substitution.
CHAPTER 4. COMPOSITIONAL REFACTORINGS

Rule 4.15 (Substitute Algorithm)

\[
\begin{array}{c|c}
\text{class } A & \text{class } A \\
ads_a & ads_a \\
\text{meth } m_1 \triangleq (pds_1 \bullet c_1) & \text{meth } m_1 \triangleq (pds_1 \bullet c'_1) \\
mts_a & mts_a \\
\end{array}
\]

\[\subseteq \text{cds}, c\]

provided

\[c_1 \subseteq c'_1\]

4.5 Conclusions

In this chapter we presented 12 refactorings based on the work of Fowler [42]. These are concerned mainly with the structure of classes and methods. Some of the refactorings that deal with data organisation were not formalised because they involve references, class attributes and methods, and collections, for instance. These features are not available in rool. However, in this chapter we presented a rule for self encapsulating attributes (refactoring SELF ENCAPSULATE FIELD [42, p. 171])

The refactorings have constraints that must be satisfied in order to allow their application. Moreover, the refactorings are presented, in its majority, as algebraic rules. The application of a rule in different directions (from left to right or from right to left) usually correspond to different refactorings.

We can classify some of the refactorings to be basic, as they can be used in the derivation of other refactorings too, and do not have a derivation that relies on other refactorings. Rules ⟨Extract/Inline Method⟩ (Rule 4.1), ⟨Move Method⟩ (Rule 4.2), and ⟨Move Attribute⟩ (Rule 4.3) are basic refactorings. This is evidenced by the fact that the derivation of ⟨Extract Class⟩ uses ⟨Move Attribute⟩ and ⟨Move Method⟩.

We presented the refactorings as rules that are derived by means of object-oriented programming laws, assuring their correctness. This allows us to reason about what underlies changes in an object-oriented program. We can observe, for instance, the role of data refinement in refactorings.

We also presented in this chapter some new refactoring rules that help in the derivation of other refactoring rules. For instance, when it is necessary to break the clientship relation between classes, the application of the rule ⟨Clientship Elimination⟩ avoids explicit data refinement of the class that is currently the client of another class. We actually believe that these rules are useful for object-oriented program transformation in general and not only to refactoring. Recall that an application of ⟨Clientship Elimination⟩ may not lead to code improvement, one of the characteristics of refactorings, but can transform an object-oriented program into another leaving its behaviour...
unchanged.

The practical methodology adopted for refactoring object-oriented programs relies on cycles of test and compilation \[42\]. Tests deal with the detection of functional errors that may be introduced in the refactoring process; compilation aims at detecting type errors that also may be introduced while refactoring. In summary, test and compilation are used to guarantee that refactored code is valid and equivalent to the original one. In this practical approach, the conditions that must be satisfied for the application of a refactoring are not clearly stated for a developer.

In our refactoring approach, the application of programming laws leads to programs that are correct by construction, so that compilation and tests are not compulsory. In fact, we should rely on the use of tools for checking if the conditions of a refactoring rule are satisfied, since non-automated checks are error-prone.

Opdyke presents a set of small refactorings that serve as a basis for the definition of more complex refactorings. Associated with each small refactoring, there are a set of preconditions that, when satisfied, assure that the refactoring does not change the meaning of a program. In our rules, the preconditions of a refactoring are presented in the provisos, according to the direction in which the rule is applied.

Besides preconditions, Roberts \[74\] presents the postconditions of refactorings. However, instead of just presenting the refactored code as a result, he writes the preconditions as assertions using the same language that is used to specify the preconditions. So, he moves the discourse from programs to the language used for writing assertions. This allows calculating the conditions that must be satisfied in order to apply a sequence of refactorings: a chain, to use Roberts’ terminology. In our approach, we relate program fragments directly.

As observed by Roberts, the conditions for a chain of refactorings is not determined by simply conjoining the preconditions of all refactorings in a chain. The reason is that some preconditions of a refactoring are satisfied due to the application of another previous refactoring. Calculating the whole precondition for a sequence of refactorings implies in the evaluation of the conditions for each refactoring against the program obtained from the application of a previous refactoring rule. The conditions for applying a refactoring may be satisfied by the program that results from a refactoring rule application. Those conditions not satisfied should be part of the conditions of the whole chain of refactorings.

As the programming laws related to object-oriented constructs have, in general, the same purpose of the basic refactorings presented by Opdyke \[69\], some of the conditions for the refactoring rule we presented in this chapter are the same as conditions of the laws used in the derivation of refactoring rules. For instance, a method we want to move to the superclass of the class in which it is declared must not exist in the superclass. This condition appears in Law \[91\] (move original method to superclass) as well as in the refactoring rule. However, the refactoring rule does present any condition about casts. This is a consequence that the bodies of the methods, which we move from classes to the common superclass, must be the same. In this way,
a programming law may present restrictions that do not appear in refactoring rules, even if law is used in the derivation of a refactoring rule.

A refactoring chain could be expressed as a single transformation rule along with the provisos that must be satisfied for the rule application using tactics for refinement [67]. The work in [68] defines a language that can be suitably extended.
Chapter 5

Contextual Refactorings

In this chapter we present refactoring rules that change the context in which the class being refactored is present. In this way, refactoring a class leads to changes in other classes and, possibly, in the main program. Consequently, differently from the derivation of compositional refactorings, whose changes do not affect the auxiliary class declarations and the main program, for the derivation of contextual refactorings, we have to consider modification in classes, other than those in which we are applying the refactoring rule, and in the main program.

This chapter is organised as follows. First, we present a refactoring rule that is used in the derivation of other refactorings. After this, we present a list of refactoring rules, based on the informal work of Fowler [42], along with their proofs. Those proofs not presented in this chapter can be found in Appendix B. Finally, we summarise our results.

The notation we use to express refactoring rules that change the context is similar to that used in Chapter 4. In addition, we write $cds[c]$ to indicate that the command $c$ may appear in classes declared in the sequence of class declarations $cds$; we can also indicate a list of commands that may appear in $cds$ by writing $cds[c_1, c_2, ..., c_n]$. We write $cds[c'/c]$ to express that the command $c'$ replaces every occurrence of the command $c$ in $cds$; in the case of a list of commands, we write $cds[c'_1, ..., c'_n/c_1, ..., c_n]$ to indicate that commands $c'_1, ..., c'_n$ replace commands $c_1, ..., c_n$, respectively, in $cds$. Moreover, we express the fact that a sequence of classes $cds'$ is defined from a sequence of classes $cds$ with some substitutions by writing $cds' \equiv cds[c'/c]$, where $c'$ may be a new command that replaces $c$. In some of the rules we present in this chapter, the program on the right-hand side refines that on left-hand side, rather than being equivalent.

5.1 Refactoring Rules

In this section we present some contextual refactoring rules. One rule changes the clientship between classes in a hierarchy. We also present a rule that moves a common attribute of subclasses to their superclass. Another rule deals with extraction of a superclass from two classes; another one allows us to remove a class from a class hierarchy. We also present rules for method renaming, method
parametrisation, and attribute encapsulation.

5.1.1 Changing clientship between classes in a hierarchy

Rule (Change clientship: from subclass to superclass) (Rule 5.1) changes clients of a class to be clients of its superclass. On the left-hand side of this rule, clients of a class $C$ call its method $p$, which accesses attribute $x$. On the right-hand side, the attribute $x$ and the method $p$ are declared in class $B$, the superclass of $C$. Previous clients of $C$ are now clients of $B$. Notice that in this rule, we do not modify the inheritance hierarchy. This rule is new; it is not presented in refactoring catalogs [69, 74, 42].

The application of this rule, from left to right, turns clients of $C$ into clients of $B$, implying in changes to attributes and methods. Attributes of type $B$ replace former attributes of type $C$. As expected, attributes of type $B$ are initialised with objects of $B$, substituting initialisation of attributes of type $C$. Previous calls to $p$ on attributes of type $C$ are substituted with calls on objects of type $B$. All these changes are expressed in the where clause.

For the application of this rule in any direction, super must not appear in the method $p$. Also, the method $p$ must not be declared by any superclass of $B$ in $cds$, avoiding the introduction of method redefinition. To apply this rule from left to right, the attribute $x$ must not be declared by any subclass of $B$ other than $C$. Also, if subclasses of $B$, excluding $C$, declare a method with name $p$, this method must have the same parameter declaration $p_{dS}$. The class $B$ cannot declare a method with name $p$. As we introduce an attribute named $b$ of type $B$ in clients of $C$, when applying this rule from left to right, we require such attribute not to be declared in these clients.

We consider clients of $C$ that call just the method $p$, which accesses only the attribute $x$. We require $p$ not to call any other method available in $C$ nor to access any attribute of $C$ or of any of its superclasses, because we are just dealing with one method and one attribute. In clients of $C$, attributes of type $C$ appear only in methods that call $p$, so that we can replace such attributes with attributes of type $B$. Also, there must be no parameters or local variables of type $C$ in methods of clients of $C$, otherwise it would not be possible to eliminate the clientship with $C$.

To apply this rule from right to left, the attribute $x$ must not be accessed by any expression of strict type $B$ in the whole program. Also, there must be no calls to $p$ with targets of type $B$. Finally, the new attribute of type $C$, which is the target of calls to $p$, must not be declared in any superclass or subclass of clients of $B$. 

### Rule 5.1 *(Change clientship: from subclass to superclass)*

<table>
<thead>
<tr>
<th>Class $B$ extends $A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ads_b$</td>
</tr>
<tr>
<td>$mts_b$</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>Class $C$ extends $B$</td>
</tr>
<tr>
<td>$pri x : T$; $ads_c$</td>
</tr>
<tr>
<td>$meth p \triangleq (pds_p \cdot c_p[self.x])$</td>
</tr>
<tr>
<td>$mts_c$</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>$cds \cdot c$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class $B$ extends $A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ads_b$; $pri x : T$</td>
</tr>
<tr>
<td>$mts_b$; $meth p \triangleq (pds_p \cdot c_p[self.x])$</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>Class $C$ extends $B$</td>
</tr>
<tr>
<td>$ads_c$</td>
</tr>
<tr>
<td>$mts_c$</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>$cds' \cdot c$</td>
</tr>
</tbody>
</table>

where

$$cds' \triangleq cds[pri b : B, new \triangleq self.b := new B, meth m \triangleq (pds_m \cdot c'_m[self.b.p(exp_p)])]$$

$$pri at : C, new \triangleq self.at := new C, meth m \triangleq (pds_m \cdot c'_m[self.at.p(exp_p)])]$$

provided

\[ \rightarrow \]
1. *super* does not appear in $p$; (2) $p$ is not declared by any superclass of $B$ in $cds$;

\[ \rightarrow \]
1. The attribute name $x$ is not declared by the subclasses of $B$ in $cds$; (2) $p$ is not declared in $mts_b$, and can only be declared in a class $D$, for any $D \leq B$ and $D \not\leq C$, if it has the same parameters as $pds_p$; (3) $b$ is not declared by any client of $C$ nor by any superclass or subclass of these clients; (4) *self*. $a$ does not appear in $p$, for any protected attribute $a$ that is declared by $B$ or by any of its superclasses, nor is declared in $ads_c$; (5) *self*. $m$ does not appear in $p$, for any method $m$ declared in $mts_c$; (6) *self*. $w$ does not appear in methods of clients of $C$ that do not call $p$, for any $w : C$; (7) there are no parameters or local variables of type $C$ in methods of clients of $C$;

\[ \leftarrow \]
1. *self*. $x$ does not appear $mts_b$; (2) $p$ is not declared in $mts_c$; (3) $D.p$ appears in $cds'$, $c$, $mts_b$, and $mts_c$ only for a $D$ such that $D \leq C$ (4) $at$ is not declared by clients of $B$ nor by any of their superclasses or subclasses.
Derivation. For the derivation of the rule (Change clientship: from subclass to superclass), we introduce a client of class $C$, in order to make explicit what happens to clients of $C$.

```
class B extends A
    ads_b
    mts_b
end
class C extends B
    pri x : T; ads_c
    meth p \hat{=} (pds_p \bullet c_p[x])
    mts_c
end
class D extends E
    pri at : C; ads_d
    meth m \hat{=} (pds_m \bullet c'_m[\text{self}.at.p(exp_p)])
    new \hat{=} \text{self}.at := \text{new} C
end
```

For each client of class $C$ we apply the rule (Clientship Elimination) (Rule [4.7]), from left to right. In this example, we have an explicit client of $C$, the class $D$. However, the changes we present here are also valid to other clients of $C$.

\[
\begin{align*}
&= \text{(Clientship Elimination) (1/r)} \\
\text{class } B \text{ extends } A \\
&\text{class } D \text{ extends } E \\
&\text{class } C \text{ extends } B \\
\end{align*}
\]

```
class B extends A
    ads_b
    mts_b
end
class C extends B
    pri x : T; ads_c
    meth p \hat{=} (pds_p \bullet c_p)
    mts_c
end
class D extends E
    pri x : T; ads_d
    meth m \hat{=} (pds_m \bullet c'_m[\text{self}.p(exp_p)])
    meth p \hat{=} (pds_p \bullet c_p)
end
```

By applying the Law [85] (change visibility: from private to public), from left to right, we change the visibility of the declaration of $x$ in $C$ to public. Then we move the attribute $x$ from $C$ to its superclass $B$. 

```
class B extends A
    ads_b
    mts_b
end
class C extends B
    pri x : T; ads_c
    meth p \hat{=} (pds_p \bullet c_p)
    mts_c
end
class D extends E
    pri x : T; ads_d
    meth m \hat{=} (pds_m \bullet c'_m[\text{self}.p(exp_p)])
    meth p \hat{=} (pds_p \bullet c_p)
end
```
5.1. REFACTORING RULES

= Law 85 (change visibility: from private to public) (l/r),
Law 87 (move attribute to superclass) (l/r)

\[
\begin{align*}
\text{class } B & \text{ extends } A \\
& \text{pub } x : T; \text{ ads}_b \\
& \text{mts}_b \\
& \text{end}
\end{align*}
\]

\[
\begin{align*}
\text{class } C & \text{ extends } B \\
& \text{ads}_c \\
& \text{meth } p \ddoteq (\text{pds}_p \bullet c_p) \\
& \text{mts}_c \\
& \text{end}
\end{align*}
\]

\[
\begin{align*}
\text{class } D & \text{ extends } E \\
& \text{pri } x : T; \text{ ads}_d \\
& \text{meth } m \ddoteq (\text{pds}_m \bullet c'_m[\text{self}.p(\exp_p)]) \\
& \text{meth } p \ddoteq (\text{pds}_p \bullet c_p) \\
& \text{end}
\end{align*}
\]

As we require the method \( p \) not to be declared in \( \text{mts}_b \), we apply the law that moves an original method from a class to its superclass.

\[
\begin{align*}
\text{class } B & \text{ extends } A \\
& \text{ads}_b; \text{ pub } x : T \\
& \text{meth } p \ddoteq (\text{pds}_p \bullet c_p) \\
& \text{mts}_b \\
& \text{end}
\end{align*}
\]

\[
\begin{align*}
\text{class } C & \text{ extends } B \\
& \text{ads}_c \\
& \text{mts}_c \\
& \text{end}
\end{align*}
\]

\[
\begin{align*}
\text{class } D & \text{ extends } E \\
& \text{pri } x : T; \text{ ads}_d \\
& \text{meth } m \ddoteq (\text{pds}_m \bullet c'_m[\text{self}.p(\exp_p)]) \\
& \text{meth } p \ddoteq (\text{pds}_p \bullet c_p) \\
& \text{end}
\end{align*}
\]

= Law 91 (move original method to superclass) (l/r)

The attribute \( x \) and the method \( p \) that were originally in class \( C \) are in \( B \), the superclass of \( C \). We privatise such attribute by applying an adequate law. Then, we introduce a clientship relation between classes that were clients of class \( C \), in the beginning of the derivation, with class \( B \). As a consequence, class \( D \) is made a client of \( B \). This also holds for all classes that were originally clients of \( C \).
In this rule, we considered that clients of $C$ call just method $p$, which accesses only attribute $x$. This simplification allows us to apply rule $\langle \text{Clientship Elimination} \rangle$ (Rule 4.7). Dealing with an arbitrary number of attributes and methods is also possible; it would imply in the application of $\langle \text{Clientship Elimination} \rangle$ according to the number of attributes and methods. Again, we consider pairs of methods and attributes that are accessed. If the method called by a client calls other methods, it is necessary to eliminate such calls. The rule $\langle \text{Clientship Elimination} \rangle$ should deal with methods that possibly access more than one attribute in order to be applied in this situation. We have already discussed such a situation in the description of $\langle \text{Clientship Elimination} \rangle$.

For simplicity, we have also considered that no superclass of $B$ introduces a definition of $p$. With a definition of $p$ in any superclass of $B$, when moving $p$ from $C$ to $B$ we are moving a redefined method to $B$, the superclass, since $B$ inherits a definition of $p$.

As with rule $\langle \text{Clientship Elimination} \rangle$, the rule $\langle \text{Change clientship: from subclass to superclass} \rangle$ is more adequately classified as a rule for program transformation rather than as an actual refactoring rule. It can be used, for instance, when a method and an attribute associated with such a method should be part of the concept described by a superclass, not a subclass.

### 5.1.2 Pull Up and Push Down Field

Subclasses developed independently can have attributes (fields) with the same purpose. These attributes may have different names, but surely they have the same type. Also, classes combined through refactoring may end with similar attributes. If attributes are used in a similar way, they can be unified, eliminating duplication. This is the purpose of rule $\langle \text{Pull Up/Push Down Field} \rangle$ (Rule 5.2).

On the left-hand side of the rule, the classes $B$ and $C$ are subclasses of $A$. The class $B$ declares the attribute $x$, whereas the class $C$ declares an attribute named $y$. Both attributes have the same type. The sequence of class declaration $\text{cds}_1$ contains the subclasses of $B$ and $C$ that refer to the attributes $x$ and $y$.

On the right-hand side of the rule, the class $A$ declares a protected attribute $z$ with the same type.
as $x$ and $y$. However, these attributes are not declared by these classes any more. Occurrences of $x$ and $y$ in methods of $B$ and $C$, respectively, are replaced with $z$. This is indicated by the notation $mts_b[z/x]$ and $mts_c[z/y]$. Also, occurrences of $x$ and $y$ in the subclasses of $B$ and $C$ that refer to such attributes are replaced with $z$. We indicate this by the notation $cds_1[z, z/x, y]$.

To apply this rule from left to right, the attribute $z$ must be new: not declared in the class $A$, $B$, $C$, nor in any subclass or superclass of $A$ that is present in $cds$. In order to apply this rule in the reverse direction, the attributes $x$ and $y$ must not be already declared in $A$. Moreover, the attribute $x$ must not be declared in $B$, or in any subclass of $B$ in $cds_1[z, z/x, y]$, nor in any superclass of $B$ in $cds$. Similar restrictions apply to attribute $y$.

Derivation. Here we prove the derivation of this refactoring from left to right. We first apply law Law 87 (move attribute to superclass), from left to right, twice to move the attributes $x$ and $y$ of classes $B$ and $C$ to their common superclass $A$; at this point $x$ and $y$ are public as required.

\[
\begin{align*}
\text{class } A & \text{ extends } D \\
& \quad \text{ads}_a; \\
& \quad \text{mts}_a \\
& \text{end} \\
\text{class } B & \text{ extends } A \\
& \quad \text{pub } x : T; \text{ ads}_b; \\
& \quad \text{mts}_b \\
& \text{end} \\
\text{class } C & \text{ extends } A \\
& \quad \text{pub } y : T; \text{ ads}_c; \\
& \quad \text{mts}_c \\
& \text{end} \\
\end{align*}
\]

= Law 87 (move attribute to superclass) $(1/r)$ $(2x)$

\[
\begin{align*}
\text{class } A & \text{ extends } D \\
& \quad \text{pub } x : T; \text{ pub } y : T; \\
& \quad \text{ads}_a; \\
& \quad \text{mts}_a \\
& \text{end} \\
\text{class } B & \text{ extends } A \\
& \quad \text{ads}_b; \\
& \quad \text{mts}_b \\
& \text{end} \\
\text{class } C & \text{ extends } A \\
& \quad \text{ads}_c; \\
& \quad \text{mts}_c \\
& \text{end} \\
\end{align*}
\]

For simplicity, we omit $cds_1$ in the derivation because modifications to the methods of classes in $cds_1$ are similar to those done to $mts_b$ and $mts_c$.

The next step is to prepare $A$ and its subclasses for a data refinement in which $x$ and $y$ are the abstract attributes, and $z$ is the concrete attribute. This preparation consists of the exhaustive application of Law 78 (simple specification) 64. It transforms assignments of the form $t := \text{self}.x$ into the specification statement $t : [\text{true}, t = \text{self}.x]$, where $t$ is a local variable or a method parameter. This transformation should be carried out in all subclasses of $A$ in which there are occurrences of $x$ and $y$ in assignments. We denote the operations of classes $A$, $B$, and $C$, after these changes, by $mts_a^{spec}$, $mts_b^{spec}$, and $mts_c^{spec}$, respectively.
**Rule 5.2 (Pull Up/Push Down Field)**

```plaintext
class A extends D
  ads_a;
  mts_a
end
class B extends A
  pub x : T; ads_b;
  mts_b
end
class C extends A
  pub y : T; ads_c;
  mts_c
end
cds_1

class A extends D
  pub z : T; ads_a;
  mts_a
end
class B extends A
  ads_b;
  mts_b
end
class C extends A
  ads_c;
  mts_c
end
cds_1'
```

where

- \( mts'_b \triangleq mts_b[z/x] \)
- \( mts'_c \triangleq mts_c[z/y] \)
- \( cds'_1 \triangleq cds_1[M.z, N.z/M.x, N.y] \), for any \( M \leq_{cds} B \) and \( N \leq_{cds} C \)

provided

- (\( \leftarrow \)) \( cds \) contains no subclasses of \( B \) and \( C \) in which there are references to \( x \) and \( y \);
- \( N.x \), for any \( N \leq_{cds} B \), does not appear in \( cds \) or \( c \), and \( N.y \), for any \( N \leq_{cds} C \), does not appear in \( cds \) or \( c \);
- \( cds_1 \) contains only subclasses of \( B \) and \( C \) in which there are references to \( x \) and \( y \);
- (\( \rightarrow \)) The attribute name \( z \) is not declared in \( ads_a \), \( ads_b \), \( ads_c \), nor in any subclass or superclass of \( A \) in \( cds \);
- \( N.z \), for any \( N \leq_{cds} A \), \( N \not\leq_{cds} N \) and \( N \not\leq_{cds} C \), does not appear in \( cds \) or \( c \);
- (\( \leftarrow \)) \( x \) (\( y \)) is not declared in \( ads_a \), \( ads_b \) (\( ads_c \)), nor in any subclass or superclass of \( B \) (\( C \)) in \( cds \) and \( cds'_1 \)
5.1. REFACTORING RULES

= Law [78] (simple specification) (multiple application)

\[\text{class } A \text{ extends } D \]
\[\text{pub } x : T; \text{ pub } y : T; \]
\[\text{ads}_a; \]
\[\text{mts}_a^{s\text{pec}} \]
\[\text{end} \]

\[\text{class } B \text{ extends } A \]
\[\text{ads}_b; \]
\[\text{mts}_b^{s\text{pec}} \]
\[\begin{align*}
\text{CI} & (\text{mts}_b^{s\text{pec}}) \\
\text{end} \end{align*} \]

\[\text{class } C \text{ extends } A \]
\[\text{ads}_c; \]
\[\text{mts}_c^{s\text{pec}} \]
\[\begin{align*}
\text{CI} & (\text{mts}_c^{s\text{pec}}) \\
\text{end} \end{align*} \]

We then apply Law [109] (superclass attribute-coupling invariant), introducing the attribute \(z\) in \(A\). The predicate \((\text{self is } B) \Rightarrow z = x) \land ((\text{self is } C) \Rightarrow z = y)\) is the coupling invariant \(CI\), relating \(z\) with \(x\) and \(y\). It states that \(z\) represents \(x\) in objects of \(B\), and \(y\) in objects of \(C\). The result is shown below.

\[\text{class } A \text{ extends } D \]
\[\text{pub } z : T; \]
\[\text{pub } x : T; \text{ pub } y : T; \]
\[\text{ads}_a; \]
\[\text{CI} (\text{mts}_a^{s\text{pec}}) \]
\[\text{end} \]

\[\text{class } B \text{ extends } A \]
\[\text{ads}_b; \]
\[\text{CI} (\text{mts}_b^{s\text{pec}}) \]
\[\text{end} \]

\[\text{class } C \text{ extends } A \]
\[\text{ads}_c; \]
\[\text{CI} (\text{mts}_c^{s\text{pec}}) \]
\[\text{end} \]

The application of \(CI\) changes guards and commands of classes \(A\), \(B\), and \(C\) according to the laws of data refinement the way already described in the derivation of the Rule [4.7]. Since the attributes \(x\) and \(y\) are new in class \(A\), there are no occurrences of them in \(\text{mts}_a^{s\text{pec}}\). Consequently, we can reduce \(\text{CI}(\text{mts}_a^{s\text{pec}})\) to \(\text{mts}_a\) by using refinement laws from [64]. The classes are now as follows.

\[\text{class } A \text{ extends } D \]
\[\text{pub } z : T; \]
\[\text{pub } x : T; \text{ pub } y : T; \]
\[\text{ads}_a; \]
\[\text{mts}_a \]
\[\text{end} \]

\[\text{class } B \text{ extends } A \]
\[\text{ads}_b; \]
\[\text{CI} (\text{mts}_b^{s\text{pec}}) \]
\[\text{end} \]

\[\text{class } C \text{ extends } A \]
\[\text{ads}_c; \]
\[\text{CI} (\text{mts}_c^{s\text{pec}}) \]
\[\text{end} \]

The next step is the elimination of occurrences of \(x\) and \(y\) in the subclasses of \(A\). We proceed with diminishing assignments of the form \(\text{self}.x, \text{self}.z := \text{exp}, \text{exp}\) to \(\text{self}.z := \text{exp}\) using the Law [73] (diminish assignment) [64].

In order to simplify the conjunction of the coupling invariant in specification statements of the form \(t, z : [\text{CI}, t = \text{self}.x \land \text{CI}]\), we apply Law [101] (is test true) and Law [102] (is test false). Inside \(B\), Law [101] (is test true) states that the test \(\text{self is } B\) is true. On the other hand, the test \(\text{self is } N\), for a class \(N\) that is not a superclass or a subclass of \(B\) is false inside \(B\), according to Law [102] (is test false). Consequently, the coupling invariant is simplified to the predicate \((\text{true } \Rightarrow z = x) \land (\text{false } \Rightarrow z = y)\) which is equivalent to \(z = x\). The specification statement \(t, z : [z = x, t = \text{self}.x \land z = x]\), at this moment, is refined to the assignment \(t := \text{self}.z\); this can
be written as \( t := \text{self}.x[z/x] \), a renaming of the original code in \( mts_b \).

We proceed in the same way with the commands of \( C \). The methods in the classes \( B \) and \( C \) that we obtain are the same as the original, except that all occurrences of \( x \) and \( y \) in the commands are replaced with \( z \).

```plaintext
class A extends D
  pub z : T;
  pub x : T; pub y : T;
  ads a;
  mts a
end

class B extends A
  ads b;
  mts.b[z/x]
end

class C extends A
  ads c;
  mts.c[z/x]
end
```

Since the abstract attributes \( x \) and \( y \) are no longer read or written in \( B \) or \( C \) and their subclasses, where they were originally declared, we can remove them from \( A \). First we apply Law \( 85 \langle \text{change visibility: from private to public} \rangle \), from right to left, in order to change the visibility of these attributes to private, since they are not read or written outside the class in which they are declared. Then we apply Law \( 83 \langle \text{attribute elimination} \rangle \), from left to right, that allows us to remove a private attribute that is not read or written inside the class where it is declared. We proceed in the same way for \( y \).

= Law \( 85 \langle \text{change visibility: from private to public} \rangle \) (r/l) (2x), Law \( 83 \langle \text{attribute elimination} \rangle \) (l/r) (2x)

```plaintext
class A extends D
  pub z : T;
  ads a;
  mts a
end

class B extends A
  ads b;
  mts.b
end

class C extends A
  ads c;
  mts.c
end
```

Notice that we replace \( mts.b[z/x] \) and \( mts.c[z/x] \) with \( mts'_b \) and \( mts'_c \), respectively, because the last ones are abbreviations of the former, as stated in the rule.

This finishes the proof of the refactoring rule \( \langle \text{Pull Up/Push Down Field} \rangle \), from left to right. The proof of the reverse direction is similar to the one presented here. Equality is a consequence of refinement in both directions. We have considered just two subclasses. The derivation for a greater number of subclasses is similar to the one presented here.

\( \square \)

In this refactoring, we considered attributes with different names. In this sense, our refactoring is more general than the one presented by Fowler [42] and the one implemented in Eclipse [37]. Also, Roberts [74] consider that variables that are moved to the superclass have the same name. The language ROOL does not allow attributes with the same name in different classes in a hierarchy. If subclasses of a common superclass have attributes with the same name, it is necessary to rename these attributes so that they end up with different names, otherwise, it is not possible to move any of the attributes up. This is the opposite of the mechanics proposed by Fowler [42], because
his refactorings are written for Java, which allows attributes with the same name in a hierarchy. Type and visibility changes, if necessary, can be made by using Law $\text{Law 86} (\text{change attribute type})$, and the adequate laws for visibility change (Law $\text{Law 84} (\text{change visibility: from protected to public})$ or Law $\text{Law 85} (\text{change visibility: from private to public}))$, in a similar way as the suggested by Opdyke [69 p. 93], when discussing refactoring for creation of an abstract superclass.

5.1.3 Extract Superclass

The rule $\langle \text{Extract Class} \rangle$ (Rule 4.6) is adequate for delegation. Inheritance is adequate when two classes share behaviour. The rule $\langle \text{Extract Superclass} \rangle$ (Rule 5.3) extracts attributes and methods common to two classes $A$ and $B$, declaring them in a new class $C$ that is declared to be the superclass of $A$ and $B$. In the case of methods in distinct classes that have the same parameters and one is a refinement of the other one, the application of rule $\langle \text{Substitute Algorithm} \rangle$ (Rule 4.15) permits obtaining just one definition for both methods.

On the left-hand side of this rule, class $B$ extends $\text{object}$ and declares the attribute $x$ of type $T$, and a method $m_1$, which is defined by the parameterised command $(pds_1 \bullet c_1)$. This class also initialises the attribute $x$. Similarly, class $C$ extends $\text{object}$ and declares the attribute $y$ of type $T$, and a method $m_1$ defined by the parameterised command $(pds_1 \bullet c_1)$. It also initialises the attribute $y$.

On the right-hand side, classes $B$ and $C$ extends class $A$, which declares the attribute $z$ and the method $m_1$ defined by the parameterised command $(pds_1 \bullet c_1)$. Accesses to attributes $x$ and $y$ in classes $B$ and $C$, respectively, are replaced with accesses to $z$. Also, subclasses of $B$ and $C$ accesses $z$ instead of $x$ and $y$. This is indicated by $cds'[z, z/x, y]$.

In order to apply this rule, attributes declared in $ads_b$ or in $ads_c$ must not appear in $c_1$. To apply this rule from left to right, the class $A$ must not be declared in $cds$ or $cds'$; an attribute with name $z$ must not be declared. Also, no expressions of type $B$ or of any subtype of $B$ are used to access the attribute $x$ of class $B$ in $cds$ or $c$. The same applies to expressions of type $C$ or of any subtype of $C$. To apply this rule from right to left, the method $m_1$ must not be declared in $mts_b$ or $mts_c$. The expression $\text{super}$ must not appear in methods of classes $B$ and $C$; the method $m_1$ must not be called on expression of type $A$ that are not of type $B$ or $C$; the attribute $z$ is not accessed by expressions of type $A$ or of any subtype of $A$ in $cds$ or in methods declared by classes $B$ and $C$. The attributes $x$ and $y$ are not declared in classes $B$ and $C$, respectively, nor in any of their subclasses. Casts involving classes $B$ and $C$ must not appear in the program. Finally, type tests with class $A$ involving attributes, variables or parameters declared to be of types $B$ and $C$ or of any of their subtypes must not appear in the whole program.
CHAPTER 5. CONTEXTUAL REFACTORINGS

Rule 5.3 \(\langle\text{Extract Superclass}\rangle\)

\[
\begin{align*}
\text{class } B \quad & \text{pub } x : T; \ ads_b \\
\text{meth } m_1 & \equiv (\textit{pds}_1 \bullet \textit{c}_1) \\
\text{new } & \equiv (\textit{val } \textit{arg} : T \bullet \textit{self}.x := \textit{arg}) \quad mts_b \\
\text{end} \\
\text{class } C \quad & \text{pub } y : T; \ ads_c \\
\text{meth } m_1 & \equiv (\textit{pds}_1 \bullet \textit{c}_1) \\
\text{new } & \equiv (\textit{val } \textit{arg} : T \bullet \textit{self}.y := \textit{arg}) \quad mts_c \\
\text{end} \\
\text{cds}' &= \text{cds}, \\
\text{class } A \quad & \text{pub } z : T; \\
\text{meth } m_1 & \equiv (\textit{pds}_1 \bullet \textit{c}_1) \\
\text{new } & \equiv (\textit{val } \textit{arg} : T \bullet \textit{self}.z := \textit{arg}) \quad mts_b[z/x] \\
\text{end} \\
\text{class } B \text{ extends } A \quad & \text{ads}_b \\
\text{mts}_b[z/x] \\
\text{end} \\
\text{class } C \text{ extends } A \quad & \text{ads}_c \\
\text{mts}_c[z/y] \\
\text{end} \\
\text{cds}'[N.z, M.z/N.x, M.y]
\end{align*}
\]

where

\[N \leq B \text{ and } M \leq C\]

provided

\((\rightarrow)\) (1) Attributes declared in \(ads_b\) or \(ads_c\) do not appear in \(c_1\):

(2) \(cds\) contains no subclasses of \(B\) or \(C\) in which there are references to \(x\) or \(y\);

\((\leftarrow)\) (1) The class \(A\) is not already declared in \(cds\) or \(cds'\); (2) an attribute named \(z\) is not declared by \(ads_b\) or \(ads_c\); (3) \(N.x\), for any \(N \leq B\), does not appear in \(cds\) or \(c\), and \(N.y\), for any \(N \leq C\), does not appear in \(cds\) or \(c\);

\((\leftrightarrow)\) (1) \(m_1\) is not declared in \(mts_b\) or \(mts_c\); (2) \textbf{super}.\(m_1\) does not appear in \(mts_b[z/x]\) or \(mts_c[z/y]\); (3) \textit{N}.\(m_1\), for any \(N \leq A\) and \(N \notin B\) or \(N \notin C\), does not appear in \(cds\), \(c\), \(mts_a\), \(mts_b\) or \(mts_c\); (4) \textit{N}.\(z\), for any \(N \leq A\), \(N \leq B\), and \(N \leq C\), does not appear in \(cds\) or \(c\); (5) \(x\) (\(y\)) is not declared in \(mrs_b\) (\(mrs_c\)), nor in any subclass of \(B\) (\(C\)) in \(cds\) or \(cds'\); (6) Casts involving \(B\) and \(C\) do not appear in \(mts_b[z/x]\), \(mts_c[z/y]\), \(c\), \(cds\) or \(cds'\); (7) Type tests with class \(A\) involving attributes, variables or parameters declared to be of types \(B\) and \(C\) or of any of their subtypes do not appear in \(mts_b[z/x]\), \(mts_c[z/y]\), \(c\), \(cds\) or \(cds'\).
5.1. REFACTORIZATION RULES

Derivation. We begin the derivation with the set of classes on the left-hand side. First we introduce class $A$, so that it is used as the superclass of classes $B$ and $C$.

```
class B
  pub x : T; ads_a
  meth m_1 \equiv (pds_1 \bullet c_1)
  new \equiv (val arg : T \bullet self.y := arg)
  mts_a
end

class C
  pub y : T; ads_a
  meth m_1 \equiv (pds_1 \bullet c_1)
  new \equiv (val arg : T \bullet self.y := arg)
  mts_b
end
```

$\Rightarrow$ Law $82$ (class elimination) $(r/l)$

```
class A
end
```

As classes $B$ and $C$ have object as superclass, we can change their superclass from object to $A$ as this class is a subclass of object and is empty.

$\Rightarrow$ Law $105$ (change superclass: from object to any class) $(2x)$

```
class A
end
```

```
class B extends A
  pub x : T; ads_a
  meth m_1 \equiv (pds_1 \bullet c_1)
  new \equiv (val arg : T \bullet self.z := arg)
  mts_a
end

class C extends A
  pub y : T; ads_a
  meth m_1 \equiv (pds_1 \bullet c_1)
  new \equiv (val arg : T \bullet self.z := arg)
  mts_b
end
```

Now we move to $A$ what $B$ and $C$ have in common. First, we move common attributes. We consider attributes with different names, but this is not a drawback as they could be renamed to a common name. We apply refactoring rule $\langle$Pull Up/Push Down Field$\rangle$ (Rule $5.2$), from left to right, resulting in the following classes.

$\Rightarrow$ $\langle$Pull Up/Push Down Field$\rangle$ $(1/r)$

```
class A
  pub z : T;
end
```
class \(B\) extends \(A\)
\[
\begin{align*}
ads_a & \quad \text{meth} \quad m_1 \triangleq (pds_1 \cdot c_1[z/x]) \\
new & \quad \triangleq (\text{val} \ arg : T \cdot \text{self}.z := \text{arg}) \\
mts_{a}[z/x] & \quad \text{end}
\end{align*}
\]

class \(C\) extends \(A\)
\[
\begin{align*}
ads_b & \quad \text{meth} \quad m_1 \triangleq (pds_1 \cdot c_1[z/y]) \\
new & \quad \triangleq (\text{val} \ arg : T \cdot \text{self}.z := \text{arg}) \\
mts_{b}[z/y] & \quad \text{end}
\end{align*}
\]

The next step moves the common methods of classes \(B\) and \(C\) to \(A\). The common methods of these classes are \(m_1\) and \(\text{new}\). Recall that in ROOL the initialiser \(\text{new}\) is just a method. Here, we consider it to have the same body in both classes. Of course, if it were defined by distinct bodies, it would not be possible to move it to \(A\).

\[
= \langle \text{Pull Up/Push Down Method} \rangle \ (1/r) \ (2x)
\]

class \(A\)
\[
\begin{align*}
\text{pub} \ & \ z : T; \\
\text{meth} \ & \ m_1 \triangleq (pds_1 \cdot c_1) \\
\text{new} & \quad \triangleq (\text{val} \ arg : T \cdot \text{self}.z := \text{arg}) \\
& \quad \text{end}
\end{align*}
\]

This finishes the derivation of the rule \(\langle \text{Extract Superclass} \rangle\).

We considered the case of moving to the superclass two attributes and one method. This is a special case of moving an arbitrary number of attributes and methods. In this case, we should apply the laws for moving attributes and methods up to the superclass as many times as the number of attributes and methods.

Notice that the conditions of rule \(\langle \text{Extract Superclass} \rangle\) are not exactly the union of the conditions of rules \(\langle \text{Pull Up/Push Down Method} \rangle\) and \(\langle \text{Pull Up/Push Down Field} \rangle\). Because we introduce class \(A\) in the derivation, and this is not an already existing class, as required by one of the conditions of \(\langle \text{Extract Superclass} \rangle\). As a consequence, it is not necessary to require \texttt{super} not to appear in \(m_1\), since in \(\langle \text{Extract Class} \rangle\) this method is always located in classes that have \texttt{object} as superclass.

The proof of this refactoring is similar to the steps adopted by Fowler [42] for its application. We do not consider, as presented by [69], heuristics that could be applied in order to make the method signatures compatible by renaming, retyping and reordering parameters, and changing visibility of
attributes that may be used by the methods. In the case of a tool that implements such heuristics, suggestions of possible changes that could allow more methods to be abstracted might be given for a developer. Opdyke takes the position that primitive refactoring can be performed to make methods more structurally similar. For each heuristics, the name of a corresponding refactoring is presented. Similarly, we can change the type of a parameter by applying a law, according to the parameter passing mechanism, just to mention a case.

5.1.4 Collapse Hierarchy

After moving attributes and methods up or down in a hierarchy, a class may end up not playing an important role; it may not add any valuable feature. Such a class can be merged with another class, resulting in an empty class that can then be removed. This is the purpose of refactoring Collapse Hierarchy [42, p. 344]. We can merge a class with a superclass or a subclass. We treat these cases separately. Rule ⟨Collapse Hierarchy - Superclass⟩ (Rule 5.4) merges a class with its superclass, whereas rule ⟨Collapse Hierarchy - Subclass⟩ (Rule 5.5) merges a class with one of its subclasses.

Collapse Hierarchy - Superclass

Rule ⟨Collapse Hierarchy - Superclass⟩ (Rule 5.4) removes a class C from a class hierarchy; its only attribute and its only method are moved to its superclass B. Consequently, attributes of type C, on the left-hand side, that are initialised with objects of type C, are turned into objects of type B, and properly initialised with objects of B. Variables and parameters of type C are turned into type B. Subclasses of C are made subclasses of B, and variables initialised with objects of C are, now, assigned with objects of B. Local variables in the main command of type C are changed to type B. Also, occurrences of ‘new C’ are changed to ‘new B’.

In order to apply this rule, there must be no occurrences of super in p, as we move this method from C to B. The name x cannot be used by any subclass of B that is not C. Class C cannot be used in types test in the program, as we cannot remove them. The method p cannot be declared in mts_b and, if declared by any subclass of B that is not C, it must have the same parameters as p. The attribute name b is not used by superclasses or subclasses of clients of C, as we introduce such attribute in clients of C. We also require references to public attributes or protected attributes of superclasses of C not to appear in p; only x can occur in p. The reason is that we are dealing with just one attribute and one method. A possible generalisation is discussed later. There must be no calls to methods inherited by C inside p. We also require that attributes of type C that are target of calls to p are not null or error avoiding program abortion. As we change the type of parameters from C to B, it is required that actual parameters associated to formal result parameters of type C are of type B or of supertype of B.
Rule 5.4 (Collapse Hierarchy - Superclass)

<table>
<thead>
<tr>
<th>Class B extends A</th>
</tr>
</thead>
<tbody>
<tr>
<td>ads _b</td>
</tr>
<tr>
<td>mts _b</td>
</tr>
<tr>
<td>END</td>
</tr>
<tr>
<td>Class C extends B</td>
</tr>
<tr>
<td>pri ( x : T )</td>
</tr>
<tr>
<td>meth ( p \equiv (pds _p \cdot c_p) )</td>
</tr>
<tr>
<td>END</td>
</tr>
<tr>
<td>( cds _1 ) ( cds _2 ) ( c )</td>
</tr>
</tbody>
</table>

where

\( cds \_1 \equiv cds \_1[visib \ b : B, new \ B, var \ v : B, par \ p : B, new \ B, extends \ B \] /[visib \ c : C, new \ C, var \ v : C, par \ p : C, new \ C, extends \ C], \)

where \( visib \in \{pri, prot, pub\} \) and \( par \in \{val, res\} \)

\( cds \_2 \equiv cds \_2[\] \)

\( c \equiv c[val \ x : B, new \ B/var \ x : C, new \ C] \)

provided

(1) \textit{super} does not appear in \( p \); (4) the attribute name \( x \) is not declared by any subclass of \( B \) in \( cds \_2 \); (5) \( C \) is not used in type tests in \( cds \_1, c \) or \( mts \_b \); (6) \( p \) is not declared in \( mts \_b \), and can only be declared in a class \( D \), for any \( D \leq B \) and \( D \nsubseteq C \), if it has the same parameters as \( pds \_p \); (7) \( b \) is not declared by any client of \( C \) nor by any superclass or subclass these clients in \( cds \_1 \); (8) \textit{self.a} does not appear in \( p \), for any public or protected attribute \( a \) that is declared by \( B \) or by any of its superclasses; (9) \textit{self.t} does not appear in \( p \), for any method \( t \) inherited by \( C \); (10) attributes of type \( C \), which are target of calls to \( p \), are not \texttt{null} or \texttt{error} are part of an invariant of \( C \);

(11) every actual parameter associated with formal result parameters of type \( C \) in \( cds \_1 \) and \( c \) are of type \( B \) or any supertype of it. (12) \( cds \_1 \) only contains clients of \( C \) and subclasses of \( C \), and \( cds \_2 \) contains no clients of \( C \) or subclasses of \( C \);

Derivation. For the derivation of (Collapse Hierarchy - Superclass), we follow the steps below. In order to make clear what happens to clients of class \( C \), in the derivation we introduce class \( D \), a client of \( C \).
5.1. REFACTORIZATION RULES

1. For all clients of $C$, eliminate clientship with $C$;

2. Eliminate every declaration of a local variable of type $C$, while introducing a variable of type $B$;

3. Introduce clientship with class $B$;

4. Change types of attributes, and value and result parameters from $C$ to $B$.

5. Eliminate casts with $C$. Change expressions ‘new $C$’ to ‘new $B$’;

6. Eliminate class $C$.

**Step 1.** We begin the derivation by eliminating the client dependence between clients of $C$ and such class. This is done for all clients of $C$.

```plaintext
class B extends A
  ads_b
  mts_b
end
class C extends B
  pri x : T
  meth p \triangleq (pds_p \bullet c_p[sf.p.(exp_p)])
end
```

= (Clientship Elimination) (1/r)

```plaintext
class B extends A
  ads_b
  mts_b
end
class C extends B
  pri x : T
  meth p \triangleq (pds_p \bullet c_p[sf.x])
end
```

**Step 2.** As we remove class $C$ in this refactoring, local variables of type $C$ must not appear in the program at the end of this derivation. We data refine every variable block that introduces variables of type $C$. Consider a local variable block like the following. Notice that objects of class $C$ can only call method $p$, as it is the unique method of this class, and $x$ is a private attribute.

```plaintext
var c : C •
c := new C;
c.p(e);
...
end
```
For the data refinement of this variable block, we have to eliminate the call to method \( p \). As the attribute \( x \) in class \( C \) is private, we apply Law \text{85} \langle \text{change visibility: from private to public} \rangle, from left to right, to change its visibility to public.

After the elimination of the call to \( p \), by an application of Law \text{95} \langle \text{method call elimination} \rangle, we obtain the following variable block.

\[
\text{var } c : C \bullet \\
\quad c := \text{new } C; \\
\quad \{ c \neq \text{null} \land c \neq \text{error} \}; \ (pds_p \bullet c_p[c.x])(e); \\
\ldots \\
\text{end}
\]

We apply Law \text{87} \langle \text{move attribute to superclass} \rangle, from left to right, to move the attribute \( x \) from class \( C \) to class \( B \). We can apply this law as it is required that subclasses of \( B \), other than \( C \), do not declare \( x \). Also, by using Law \text{91} \langle \text{move original method to superclass} \rangle, from left to right, we move method \( p \) from \( C \) to \( B \).

Now we prepare the variable block for data refinement by applying Law \text{78} \langle \text{simple specification} \rangle, from left to right.

\[
\text{var } c : C \bullet \\
\quad c := \text{new } C; \\
\quad \{ c \neq \text{null} \land c \neq \text{error} \}; \ (pds_p \bullet c_p[c.x])(e); \\
\ldots \\
\text{end}
\]

We use a data refinement law, Law \text{81} \langle \text{Data refinement—variable blocks with initialisation} \rangle, with coupling invariant \( b = c \), in which \( b \) is a concrete variable of type \( B \). By the application of this law, we obtain a variable block as follows.

\[
\text{var } b : B \bullet \\
\quad b := \exists c : C \bullet b = c \land c = \text{new } C; \\
\quad \{ b \neq \text{null} \land b \neq \text{error} \}; \ (pds_p \bullet c_p[b.x])(e); \\
\ldots \\
\text{end}
\]

By the predicate calculus (one-point rule) and Law \text{78} \langle \text{simple specification} \rangle, from right to left, we obtain the following variable block.

\[
\text{var } b : B \bullet \\
\quad b := \text{new } C; \\
\quad \{ b \neq \text{null} \land b \neq \text{error} \}; \ (pds_p \bullet c_p[b.x])(e) \\
\ldots \\
\text{end}
\]
We can introduce a call to method \( p \) of class \( B \) from the sequential composition formed by the assumption and the parameterised command that is the same as the one that defines method \( p \) in class \( B \).

\[
\text{var } b : B \bullet \\
b := \text{new } C; \\
b.p(e); \\
\ldots \\
\text{end}
\]

We change the visibility of attribute \( x \) of class \( B \) back from public to private by applying Law 85 (change visibility: from private to public), from right to left.

**Step 3.** Previous clients of \( C \) (like the class \( D \) that we are using in this derivation) can now be made clients of class \( B \). We apply rule (Clientship Elimination), from right to left.

\[
= (\text{Clientship Elimination}) \ (r/l)
\]

\[
\begin{align*}
\text{class } B & \text{ extends } A \\
& \quad \text{ads}_a; \ tri x : T \\
& \quad \text{meth } p \triangleq (pds_p \bullet c_p[\text{self}.x]) \\
& \quad \text{mts}_b \\
\text{end} \\
\text{class } C & \text{ extends } B \\
\text{end}
\end{align*}
\]

\[
\begin{align*}
\text{class } D & \text{ extends } E \\
& \quad \text{tri } b : B; \ tri d; \ ads_d \\
& \quad \text{meth } m \triangleq (pds_m \bullet c_m[\text{self}.b.p(exp_p)]) \\
& \quad \text{new } \triangleq \text{self}.b := \text{new } B \\
\text{end}
\end{align*}
\]

**Step 4.** In the previous steps, we changed every client of \( C \) to be a client of \( B \). Also, local variables originally of type \( C \) are now of type \( B \). Value and result parameters of type \( C \) must have their types changed from \( C \) to \( B \). First, we apply Law 97 (introduce trivial cast in expressions) for introducing casts to \( C \) in every non-assignable occurrence of local variables, and value and result parameters of type \( C \). In order to change the type of local variables, value and result parameters of type \( C \), we use Lemma 7 (Lemma Type changes in program, see Appendix C), from left to right.

**Step 5.** By Lemma 8 (Lemma Program cast elimination, see Appendix C), we remove casts to \( C \) and assumptions with type tests that appear in a program. We have already moved the attribute \( x \) and methods from \( C \) to \( B \). Thus, class \( C \) is empty and we can change occurrences of the expression \( \text{new } C \) to \( \text{new } B \) without affecting the program. For this we apply Law 103 (new superclass), from left to right. Notice that as a consequence of the application of this law, the variable block
we presented in this derivation is now as follows.

\[
\text{var } b : B \star \\
b := \text{new } B; \\
b.p(e) \\
\ldots \\
\text{end}
\]

**Step 6.** By applying law \(\text{⟨change superclass: from an empty class to immediate superclass⟩}\), from left to right, every subclass of \(C\) is changed to be a subclass of \(B\). We can apply this law because there are no type casts or test involving \(C\), and we have changed the type of attributes, variables and parameters from \(C\) to \(B\). Consequently, there are no use of \(C\) in the program, and we can remove it from the program by applying Law 82 (class elimination), from left to right.

\[
\text{class } B \text{ extends } A \\
\text{ads}_b; \text{ pri } x : T \\
\text{meth } p \triangleq (\text{pds}_p \bullet \text{c}_p[\text{self}.x]) \\
\text{mts}_b \\
\text{end}
\]

\[
\text{class } D \text{ extends } E \\
\text{pri } b : B; \text{ ads}_d \\
\text{meth } m \triangleq (\text{pds}_m \bullet \text{c}_m[\text{self}.b.p(exp_p)]) \\
\text{new } \triangleq \text{self}.b := \text{new } B \\
\text{end}
\]

This finishes the derivation of rule \(\text{⟨Collapse Hierarchy - Superclass⟩}\).

We considered the class being removed to have just one attribute and one method. This simplification allows us to use rule \(\text{⟨Clientship Elimination⟩}\) (Rule 4.7). Dealing with more attributes and methods requires a rule that allows changing a clientship involving more than one attribute and one method.

**Collapse Hierarchy - Subclass**

Rule \(\text{⟨Collapse Hierarchy - Subclass⟩}\) (Rule 5.5) is similar to rule \(\text{⟨Collapse Hierarchy - Superclass⟩}\), but it removes a class from a class hierarchy; its attribute and method are moved to its subclass in such hierarchy. Clients of the class being removed are made clients of its subclass; subclasses of the class being removed have their superclasses changed to be the subclass of class being removed. Types of attributes, variables, and parameters are changed from \(B\) to \(C\).

To apply this rule, the method \(p\) must not be declared in \(\text{mts}_b\) as we move it from \(B\) to \(C\). The method \(p\) must not be called by expressions of strict type \(B\), as this method, after the application of this rule, is located in class \(C\). Expressions that are of subtype of \(B\), but are not of subtypes of \(C\), must not be target of method calls, as we move \(p\) from \(B\) to \(C\). The application of this refactoring rule remove class \(B\). So, this class cannot be used in type tests. All the other conditions for the application of this refactoring rule, except the last one, are related to changing types of attributes, local variables, value and result parameters.
Rule 5.5 (Collapse Hierarchy - Subclass)

\[
\begin{array}{l}
\text{class } B \text{ extends } A \\
\quad \text{pri } x : T \\
\quad \text{meth } p \triangleq (pds_p \cdot c_p) \\
\text{end}
\end{array}
\]

\[
\begin{array}{l}
\text{class } C \text{ extends } B \\
\quad \text{ads}_{c} \\
\quad \text{mts}_{c} \\
\text{end}
\end{array}
\]

\[
\begin{array}{l}
\text{class } C \text{ extends } A \\
\quad \text{pri } x : T; \text{ ads}_{c} \\
\quad \text{meth } p \triangleq (pds_p \cdot c_p) \\
\quad \text{mts}_{c}; \\
\text{end}
\end{array}
\]

\[
\begin{array}{l}
\quad \text{cds}_{1} \cds_{2} \bullet c' \\
\end{array}
\]

where

\[
\begin{array}{l}
\text{cds}_{1}' \triangleq \text{cds}_{1}[\text{visib } x : C, \text{ new } C, \text{ var } v : C, \text{ par } p : C, \text{ extends } C/ \\
\quad \text{visib } c : B, \text{ new } B, \text{ var } v : B, \text{ par } p : B, \text{ extends } B], \\
\end{array}
\]

\[
\begin{array}{l}
\text{where visib } \in \{\text{pri, prot, pub}\} \text{ and par } \in \{\text{val, res}\}
\end{array}
\]

\[
\begin{array}{l}
c' \triangleq c[\text{new } C/\text{new } B]
\end{array}
\]

provided

1. \(p\) is not declared in \(\text{mts}_{c}\); (2) \(D.p\), for any \(D \leq B\) and \(D \nleq C\), does not appear in \(\text{cds}_{1}\), \(\text{cds}_{2}\), \(c\), \(\text{mts}_{b}\) or \(\text{mts}_{c}\); (3) \(B\) is not used in type tests in \(\text{cds}, c\) or \(\text{mts}_{b}\); (4) every non-assignable occurrences of attributes, local variable, value and result parameters of type \(B\) in expressions are cast with \(C\) or any subclass of \(C\); (5) every actual parameter associated to a formal value parameter of type \(B\) is of type \(C\) or of any subtype of \(C\); (6) every expression assigned to a value parameter of type \(B\) is of type \(C\) or of any subtype of \(C\); (7) every use of value argument of type \(B\) as a result argument is for a corresponding formal parameter of type \(C\) or of any subclass of \(C\); (8) every expression assigned to a result parameter of type \(B\) is of type \(C\) or any subclass of \(C\); (9) every use of a result parameter of type \(B\) as a result argument is for a corresponding formal parameter of type \(C\) or of any subclass of \(C\); (10) every expression assigned to a variable of type \(B\) is of type \(C\) or of any subclass of \(C\); (11) every use of a variable of type \(B\) is for a corresponding formal result parameter of type \(C\) or any subclass of \(C\); (12) \(\text{cds}_{1}\) only contains clients of \(C\) and subclasses of \(B\), and \(\text{cds}_{2}\) contains no clients of \(C\) or subclasses of \(B\);
Derivation. We follow the steps below.

1. For all clients of $B$, change clientship from $B$ to $C$;
2. Change types of attributes, local variables, value and result parameter from $B$ to $A$;
3. Eliminate casts with $B$;
4. Change ‘new $B$’ to ‘new $C$’, and superclasses from $B$ to $A$;
5. Eliminate class $B$.

**Step 1.** We begin the derivation by changing clients of $B$ to be clients of $C$.

```plaintext
class A extends F
  ads_a
  mts_a
end
class B extends A
  pri x : T
  meth p \triangleq (pds_p \bullet c_p)
end
class C extends B
  ads_c
  mts_c
end
```

We remove any occurrence of super in the method $p$ by using the Law [94](eliminate super), from left to right, resulting in the command $c'_p$. It may be necessary to apply Lemma [4](Eliminate super in Hierarchy) to method $p$ of class $B$. We eliminate the clientship relation with respect to class $B$ and introduce clientship with $C$, using rule (Change clientship: from subclass to superclass), from right to left.

= Law [94](eliminate super) (l/r), (Change clientship: from subclass to superclass) (r/l)

```plaintext
class A extends F
  ads_a
  mts_a
end
class D extends E
  pri b : B; ads_d
  meth m \triangleq (pds_m \bullet c_m[self.b.p(exp_p)])
  new \triangleq self.b := new B
end
class B extends A
end
class C extends B
  pri c : C; ads_c
  meth m \triangleq (pds_m \bullet c_m[self.c.p(exp_p)])
  meth new \triangleq self.c := new C
end
```
5.1. REFACTORIZATION RULES

Step 2. We change type of attributes, local variables, and parameters from \( B \) to \( C \), by using Lemma 7 (Lemma Type changes in program, see Appendix C), from right to left.

Step 3. For removing type casts involving class \( B \), we use Lemma 8 (Lemma Program cast elimination, see Appendix C).

Step 4. We change new \( B \) to new \( C \), by applying law Law 104 (new subclass), from left to right. Also, we change the superclass of classes from \( B \) to \( A \). In this step, we also introduce calls on super in the method \( p \), by applying Law 94 (eliminate super), from right to left, resulting in the original command \( c_p \). It may be necessary to apply Lemma 5 (Eliminate super and Trivial Methods in Hierarchy) to method \( p \) of class \( C \).

\[
\text{Step 5. Finally, we eliminate class } B \text{ as there are no references to it in the program.}
\]

\[
\text{Law 82 (class elimination) (l/r)}
\]

This finishes the derivation of (Collapse Hierarchy - Subclass).
In the derivation of \( \langle \text{Collapse Hierarchy - Superclass} \rangle \) we applied data refinement in order to change the type of local variables of type \( C \), the class that is removed from the hierarchy, to variables of type \( B \), the superclass of \( C \). However, in the derivation of \( \langle \text{Collapse Hierarchy - Subclass} \rangle \), the type of local variables were changed from \( B \) to \( C \) by the application of a lemma in which these changes are realised by programming laws. The reason for this is the impossibility of applying data refinement from an abstract variable of a superclass to a concrete variable of a subclass. This is evidenced by the fact that we cannot assign an object of a class to a variable whose type is a subclass of such a class, as would arise as a consequence of the data refinement of assignment.

The data refinement of local blocks in the derivation of \( \langle \text{Collapse Hierarchy - Superclass} \rangle \) is not responsible for changing occurrences of ‘\text{new } C’ to ‘\text{new } B’’. In the derivation, the application of Law 103 \( \langle \text{new superclass} \rangle \) changes the expressions involving \( C \) to that involving \( B \). Strictly, this change is not possible from a data refinement stand point, because it is not possible to obtain ‘\text{new } B’ from ‘\text{new } C’’. Also, the coupling invariant relating \( b \), the concrete variable of type \( B \), and \( c \), the abstract variable of type \( C \), cannot include an expression like ‘\text{new } B’, since this does not maintain the coupling invariant.

As expected from an informal approach, which is based on tests and compilation, Fowler [42] does not make clear the restrictions for the application of this refactoring, specially those related to type changes as one class of a hierarchy is removed. Also, he does not make clear what happens to clients of the removed class.

We have presented a rule that deals with removing a class that has just one attribute and one method. The main reason for this is that the rule that we use to change clientship from a subclass to a superclass also deals with just one attribute and one method. As discussed in that rule, we could deal with more attributes and more methods. Certainly, this situation is more appropriate in a context in which a tool gives support to refactorings, as it could deal more easily with an arbitrary number of attributes and methods. The fact that it is possible to deal with one attribute and one method shows that we can deal with an arbitrary number of them.

5.1.5 Rename Method

Method names play an important role in object-oriented systems. They should be used to communicate the intention behind the method code. If we have a method whose name does not express its purpose, renaming a method is necessary in order to clarify the purpose of such method. This is the aim of the refactoring rule \( \langle \text{Rename Method} \rangle \) (Rule 5.6).

On the left-hand side of this rule, there is a method named \( m \). This method can be called at any point in the whole program. Applying this rule renames the method \( m \) to \( n \).

Renaming a method inside a class affects not only the clients of such class, but also the classes in the same hierarchy in which the method is present. In the rule \( \langle \text{Rename Method} \rangle \) (Rule 5.6), we use \( \text{cds}_1 \) to indicate the subclasses of the topmost class in the hierarchy of \( A \) that first introduces a method named \( m \). In all these classes, method \( m \) must be renamed to \( n \).
### 5.1. REFACTORIZING RULES

**Rule 5.6 (Rename Method)**

\[
\begin{align*}
\text{class } A & \text{ extends } C \\
ads_a; & \\
\text{meth } m \doteq (pds \bullet cmd) & \\
mts_a & \\
\text{end} & \\
cds_1 cds_2 \bullet c
\end{align*}
\]

\[
\begin{align*}
\text{class } A & \text{ extends } C \\
ads_a; & \\
\text{meth } n \doteq (pds \bullet cmd) & \\
mts_a & \\
\text{end} & \\
cds'_1 cds_2 \bullet c'
\end{align*}
\]

where

For all \( le : D \), such that \( D \leq A \), we have:

\[
\begin{align*}
cds'_1 & \equiv cds_1[\text{meth } n \doteq (pds \bullet c_1), le.n(a_n)/\text{meth } m \doteq (pds \bullet c_1), le.m(a_n)] \\
c' & \equiv c[le.n(a_n)/le.m(a_n)]
\end{align*}
\]

provided

\((\rightarrow)\)

1. \( cds_1 \) contains all subclasses of the topmost class in the hierarchy of \( A \) that introduces a method named \( m \), including that class, and classes in which there are calls to \( m \); \( cds_2 \) contains all classes other than those in \( cds_1 \); (2) \( n \) is not declared in any class in the whole hierarchy of the topmost superclass of \( A \) that first introduces a definition of \( m \);

\((\leftarrow)\)

1. \( cds'_1 \) contains all subclasses of the topmost class in the hierarchy of \( A \) that first introduces a definition of \( n \), and classes in which there are calls to \( n \); \( cds_2 \) contains all classes other than those in \( cds'_1 \); (2) \( m \) is not declared in any class in the whole hierarchy of the topmost class of \( A \) that first introduces a definition of \( n \);

In order to apply this law from left to right, a method named \( n \) must not be already declared in the whole hierarchy of class \( A \). Applying this rule from right to left requires similar conditions to be satisfied.

**Derivation.** The derivation, from left to right, follows the strategy below. The derivation from right to left is similar. In the following steps we use the class name \( Z \) to refer to the topmost class in the hierarchy of class \( A \) that first introduces a definition of \( m \).

1. Moving of all definitions of method \( m \) to \( Z \);
2. Elimination of calls \( le.m \), such that \( le : N \) for \( N \leq Z \);
3. Introduction of a new method named \( n \) with the same body as the method \( m \) in \( Z \);
4. Elimination of method \( m \) from \( Z \);
5. Introduction of calls to the new method \( n \);
6. Moving of the method \( n \) down from \( Z \) to its subclasses.

**Example.** Let us consider the following class declarations. We assume the conditions for the application of the refactoring rule (Rename Method) are satisfied. In this example, we consider a method with a value parameter. The argument \( a \) denotes an expression of type \( T \).

```latex
\begin{align*}
\text{class } D \\
ads_c \\
\mathbf{meth} \; m &\triangleq (\mathbf{val} \; x : T \bullet c_d) \\
mts_c[\mathbf{self}.m(a)] \\
\text{end} \\
\text{class } B \mathbf{extends } D \\
ads_b \\
\mathbf{meth} \; m &\triangleq (\mathbf{val} \; x : T \bullet c_b) \\
mts_b[\mathbf{self}.m(a)] \\
\text{end} \\
\text{class } A \mathbf{extends } D \\
ads_a \\
\mathbf{meth} \; m &\triangleq (\mathbf{val} \; x : T \bullet c_a) \\
mts_a[\mathbf{self}.m(a)] \\
\text{end} \\
\text{class } C \mathbf{extends } A \\
ads_c \\
\mathbf{mts}_c[\mathbf{self}.m(a)] \\
\text{end}
\end{align*}
```

First, we move all definitions of \( m \) to class \( D \) using Lemma 2 (Pull Up Method in the Whole Hierarchy), see Appendix C. This lemma moves all definitions of a method to the topmost class in the hierarchy that first introduces a definition for it.

= Lemma 2

```latex
\begin{align*}
\text{class } D \\
ads_c \\
\mathbf{meth} \; m &\triangleq (\mathbf{val} \; x : T \bullet \\
&\text{if } \mathbf{self} \text{ is } B \rightarrow c_b \\
&\quad \text{[} \neg (\mathbf{self} \text{ is } B) \rightarrow \text{if } \neg (\mathbf{self} \text{ is } A) \rightarrow c_d \\
&\quad \quad \text{[} \mathbf{self} \text{ is } A \rightarrow \text{if } (A)\mathbf{self} \text{ is } C \rightarrow c_c' \\
&\quad \quad \quad \text{[} \neg ((A)\mathbf{self} \text{ is } C) \rightarrow c_a' \quad \text{fi} \\
&\quad \text{fi} \\
&\quad \text{mts}_d[\mathbf{self}.m(a)] \\
\text{end}
\end{align*}
```
We eliminate every method call $le.m(a)$ in $cds_2$ and $c$. We also eliminate the parameterised commands that result from the method call elimination. Let us consider such command.

\[ le.m(a) \]

\[ = \text{Law 95 (method call elimination) (1/r)} \]

\[
\{ le \neq \text{null} \land le \neq \text{error} \} \\
(\text{val } x : T \bullet \text{if } le \text{ is } B \rightarrow c'_b) \\
\text{fi} \\
\]

Notice that, for every method call $le.m(a)$ that was present in the original program, we have now a sequential composition of an assumption about the method call target and an alternation similar to the one that defines the method $m$ of class $D$. Similarly, we eliminate calls to the method $m$ that have $self$ as target in the hierarchy of class $D$. This yields sequential compositions like the one above, but with assumptions about $self$.

We introduce a method called $n$ in class $D$ with the same parameterised command that defines $m$. For that, we apply Law 90 (method elimination), from right to left. We assume that $n$ is not declared by any superclass or subclass of $D$; this is a condition for the application of the refactoring rule. We eliminate the method $m$, because it is not called by any client of $D$ nor within $D$,
by applying Law \[90\] (method elimination), from left to right.

In class \(D\), the method \(n\) is defined with the same parameterised command that resulted from the elimination of calls to the previously existing method \(m\). Throughout the program, this parameterised command, due to method call elimination, is preceded by an assumption about the method call target. So, for each sequential composition like the one of Step 6, we can introduce a call to the method \(n\) of class \(D\), by applying Law \[95\] (method call elimination), from right to left, yielding the following method call.

\[le.n(a)\]

The introduction of calls to \(n\) with target \(self\) is also carried out, yielding calls \(self.n(a)\).

We can move \(n\) down to the subclasses of \(D\), according to the alternation that is present in this method, using Lemma \[3\] (Push Down Method in the Whole Hierarchy), see Appendix [C]. This lemma moves all definitions of a method from the topmost class in the hierarchy that first introduces
5.1. REFACTORIZING RULES

a definition for it in the direction of classes at the bottom of the hierarchy.

= Lemma 3

```plaintext
class D
  ads;
  meth n ⇝ (val x : T • c_d)
  mts_s[Self.n(a)]
end

class A extends D
  ads_a;
  meth n ⇝ (val x : T • c_a)
  mts_a[Self.n(a)]
end

class B extends D
  ads_b;
  meth n ⇝ (val x : T • c_b)
  mts_b[Self.n(a)]
end

class C extends A
  ads_c;
  meth n ⇝ (val x : T • c_c)
  mts_c[Self.n(a)]
end

cds_1 [le.n(a)] cds_2 [le.n(a)] • c
```

This finishes the derivation of the example for rule ⟨Rename Method⟩.

□

In the example, the parameter list of the method being renamed is constituted just of one value parameter. In the rule, however, any number of parameters of different passing mechanisms are accepted. This refactoring is presented by both Opdyke [69, p.61] and Fowler [42, p. 273]. Opdyke considers the case of renaming a method to the same name of an already inherited method, but either this method must not be called on instances of the class that declares the method being renamed and on instances of its subclasses, or both methods are semantically equivalent. In this situation, methods must have compatible signatures. In our rule, we exclude the possibility of renaming a method to the name of an inherited method, as this case is a special case of the one presented here; it is only necessary to impose more restrictions on method call targets. Requiring that both methods are semantically equivalent is the same as requiring that they refine each other.

5.1.6 Parameterise Method

Methods declared in the same class, that perform the same tasks, but that are differentiated by values that they use, can be simplified to a method that handles the variations by means of parameters. The rule ⟨Parameterise Method⟩ (Rule 5.7) parameterise methods that have similar bodies, differentiated just by expressions that appear in these bodies.

On the left-hand side of this rule, in the command cmd in the methods m_1 and m_2 of the class A, there are occurrences of the expressions exp_1 and exp_2, respectively, or of type T. The sequence of class declarations cds_1 contains only subclasses of A; the sequence cds_2 contains all other classes. On the right-hand side, class A declares just one method named m defined by a parameterised command that has a parameter of type T. Every method call le.m_1, for any left-expression le whose type is a subclass of A (or A itself), that appears on the left-hand side, is replaced with a
**Rule 5.7** *(Parameterise Method)*

<table>
<thead>
<tr>
<th>class A extends C</th>
<th>class A extends C</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{ads}_a )</td>
<td>( \text{ads}_a )</td>
</tr>
<tr>
<td>\text{meth } m_1 \triangleq (\bullet \text{cmd}[\text{exp}_1])</td>
<td>\text{meth } m \triangleq (\text{val} \ arg : T \bullet \text{cmd}[\text{arg}])</td>
</tr>
<tr>
<td>\text{meth } m_2 \triangleq (\bullet \text{cmd}[\text{exp}_2])</td>
<td>\text{mts}'_a</td>
</tr>
<tr>
<td>\text{mts}_a</td>
<td>end</td>
</tr>
<tr>
<td>( cds_1 \ cds_2 \bullet c )</td>
<td>( cds_1' cds_2' \bullet c' )</td>
</tr>
</tbody>
</table>

where

\( mts'_a \triangleq mts_a[\text{var } x : T \bullet x := \text{exp}_1; \ \text{le}.m(x) \ \text{end}/\text{le}.m_1, \)

\( \quad \text{var } x : T \bullet x := \text{exp}_2; \ \text{le}.m(x) \ \text{end}/\text{le}.m_2] \)

\( cds'_1 \triangleq cds_1[\text{var } x : T \bullet x := \text{exp}_1; \ \text{self}.m_1(x) \ \text{end}/\text{self}.m_1, \)

\( \quad \text{var } x : T \bullet x := \text{exp}_2; \ \text{self}.m(x) \ \text{end}/\text{self}.m_2, \)

\( \quad \text{var } x : T \bullet x := \text{exp}_1; \ \text{le}.m(x) \ \text{end}/\text{le}.m_1, \)

\( \quad \text{var } x : T \bullet x := \text{exp}_2; \ \text{le}.m(x) \ \text{end}/\text{le}.m_2] \)

\( cds'_2 \triangleq cds_2[\text{var } x : T \bullet x := \text{exp}_1; \ \text{le}.m(x) \ \text{end}/\text{le}.m_1, \)

\( \quad \text{var } x : T \bullet x := \text{exp}_2; \ \text{le}.m(x) \ \text{end}/\text{le}.m_2] \)

\( c' \triangleq c[\text{var } x : T \bullet x := \text{exp}_1; \ \text{le}.m(x) \ \text{end}/\text{le}.m_1, \)

\( \quad \text{var } x : T \bullet x := \text{exp}_2; \ \text{le}.m(x) \ \text{end}/\text{le}.m_2] \)

provided

1. \( cds_1 \) contains only subclasses of \( A \); 2. \( cds_2 \) contains no subclasses of \( A \); 3. \( \text{exp}_1 \)

and \( \text{exp}_2 \) are expressions of type \( T \); 4. \( \text{le} : B \), for a \( B \) such that \( B \leq A \); 5. \( m_1 \)

and \( m_2 \) are not declared in any superclass of \( A \); 6. \( m_1 \) and \( m_2 \) are not redefined

in any subclass of \( A \); 7. \( m \) is not declared in any superclass or subclass of \( A \); 8. \( x \)

is a fresh variable; 9. the types of both \( \text{exp}_1 \) and \( \text{exp}_2 \) are basic.

---

call \( \text{le}.m(x) \) that occurs inside a variable block that declares \( x \). In such block, we assign to \( x \) the

expression \( \text{exp} \). The changes to method calls \( \text{le}.m_2 \) are similar.

In order to apply this rule, we require that the methods \( m_1 \) and \( m_2 \) are not defined in any

superclass of \( A \) or in any of its subclasses. We also require that the method \( m \) is new: not declared

in \( mts_a \) nor in any superclass or subclass of \( A \). Also the types of expressions \( \text{exp}_1 \) and \( \text{exp}_2 \) are basic.
5.1. REFACTORING RULES

**Derivation.** The first step of the derivation is to change the visibility of the declaration of any private attribute to public; we apply Law 85 (change visibility: from private to public), from left to right. Then, we introduce the method \( m \) in \( A \). We apply the Law 90 (method elimination), from right to left. The parameterised command that defines the method \( m \) has a value parameter \( (\text{arg}) \); the command \( cmd[\text{arg}] \) is similar to \( cmd \) with occurrences of \( \text{exp}_1 \) replaced with the parameter \( \text{arg} \).

\[
\text{meth } m \equiv (\text{val } \text{arg} : T \bullet cmd[\text{arg}])
\]

In what follows, we change method \( m_1 \). Changes to method \( m_2 \) are similar.

\[
\text{meth } m_1 \equiv (\bullet cmd[\text{exp}_1])
\]

The first step is to introduce a variable to hold \( \text{exp}_1 \). We declare variable \( x \) and define its initial value to be expression \( \text{exp}_1 \).

\[\text{Law 25 (var elim) (r/l), Law 32 (var := initial value)}\]

\[
\text{var } x : T \bullet x := \text{exp}_1; \text{cmd}[\text{exp}_1] \text{ end}
\]

As the expression assigned to \( x \) occurs in the command \( cmd \), we can replace such occurrences with \( x \).

\[\text{Law 60 (assignment seq comp exp substitution) (1/r)}\]

\[
\text{var } x : T \bullet x := \text{exp}_1; \text{cmd}[x] \text{ end}
\]

From the command \( cmd[x] \), we introduce a parameterised command with the value parameter \( \text{arg} \). This parameterised command is applied to the argument \( x \).

\[\text{Lemma 9 (1/r)}\]

\[
\text{var } x : T \bullet x := \text{exp}_1;
(\text{val } \text{arg} : T \bullet cmd)(x)
\text{ end}
\]

The parameterised command we just introduced is equal to the parameterised command that defines the method \( m \). As the methods \( m \) and \( m_1 \) are defined in the same class, we can introduce a call to \( m \) from \( m_1 \).
= Lemma [H (r/l)]

\[
\begin{align*}
&\textbf{var } x : T \bullet x := \text{exp}_1; \\
&\text{self.m}(x) \\
&\text{end}
\end{align*}
\]

Now we eliminate every call \( l.e.m_1() \), yielding the following sequential composition.

= Law \([95 (method\ call\ elimination) (l/r)\)

\[
\begin{align*}
&\{ \text{le} \neq \text{null} \land \text{le} \neq \text{error} \} \\
&\bullet \textbf{var } x : T \bullet x := \text{exp}_1; \\
&\text{le.m}(x) \\
&\text{end}()
\end{align*}
\]

In ROOL, a parameterised command \( \bullet c() \) can be written just as the command \( c \). So we simplify the parameterised command \( \bullet \textbf{var } x : T \ldots \text{end}() \) to \( \textbf{var } x : T \ldots \text{end} \).

\[
\begin{align*}
&\{ \text{le} \neq \text{null} \land \text{le} \neq \text{error} \} \\
&\textbf{var } x : T \bullet x := \text{exp}_1; \\
&\text{le.m}(x) \\
&\text{end}
\end{align*}
\]

We refine the assumption \( \{ \text{le} \neq \text{null} \land \text{le} \neq \text{error} \} \) to \textbf{skip}. As \textbf{skip} always terminates without changing the state of any variable in a program, we can remove it.

\[\subseteq \text{Law [77 (remove assumption)], Law [16 (; \textbf{–skip unit})] }\]

\[
\begin{align*}
&\textbf{var } x : T \bullet x := \text{exp}_1; \\
&\text{le.m}(x) \\
&\text{end}
\end{align*}
\]

The previous steps regarding the elimination of calls to method \( m_1 \) can be applied to eliminate calls to \( m_2 \). Finally we remove the methods \( m_1 \) and \( m_2 \) from \( A \).

\[
\begin{align*}
&\text{class } A \text{ extends } C \\
&\quad \textit{ads}_a \\
&\quad \textit{meth } m_1 \triangleq (\bullet \text{cmd}[\text{exp}_1]) \\
&\quad \textit{meth } m_2 \triangleq (\bullet \text{cmd}[\text{exp}_2]) \\
&\quad \textit{meth } m \triangleq (\textit{val } \textit{arg} : T \bullet \text{cmd'}) \\
&\quad \textit{mts}_a \\
&\text{end}
\end{align*}
\]
5.1. REFACTORING RULES

\[ \text{Law} \langle \text{method elimination} \rangle \ (1/r) \ (2x) \]

```plaintext
class A extends C
  ads_a
  meth m \triangleq (val arg : T • cmd')
end
```

Finally we change the visibility of any public attribute that appears in \( m \) to private; we apply Law \( \langle \text{change visibility: from private to public} \rangle \), from right to left. This finishes the derivation of the rule \( \langle \text{Parameterize Method} \rangle \).

\[ \square \]

We considered a refactoring similar to that of Fowler [42], in the sense that methods perform the same tasks, but different values are used for these tasks. In this way, these values are passed by means of arguments to the method being parameterised. This is the reason for the use of the value parameter passing mechanism. We also required the types of the expressions, whose values are passed as argument, to be basic.

5.1.7 Encapsulate Field

Public attributes reduce the modularity of object-oriented programs. They break data hiding because client classes can have direct access to them. The rule \( \langle \text{Encapsulate Field} \rangle \) (Rule 5.8) hides a public attribute and provides get and set methods for it.

The class \( A \) on the left-hand side of the rule \( \langle \text{Encapsulate Field} \rangle \) includes a public attribute \( x \). The context for this class is the sequence of classes \( cds \) and the command \( c \), which can contain expressions for direct access to \( x \). In class \( A \) on the right-hand side, the attribute \( x \) is private and get and set methods are declared. The context for class \( A \) is the sequence of classes \( cds' \) and the command \( c' \). Direct accesses to \( x \) that are present in the \( cds \) and \( c \) are replaced with calls to get and set methods on the right-hand side.

To apply this rule, the type of \( x \) must be basic. To apply this rule from left to right, the methods \( \text{getX} \) and \( \text{setX} \) must not be already declared in \( A \) nor in any of its superclasses or subclasses. To apply this rule in the reverse direction, the methods \( \text{getX} \) and \( \text{setX} \) must not be called in any point of the whole program.

The refactoring rule \( \langle \text{Encapsulate Field} \rangle \) (Rule 5.8) leads to changes in the context of the class to which the rule is applied since all direct accesses to a public attribute must now be indirect, by using get and set methods.

Assignments of the form \( le.x := e \), with \( le : N \) for \( N \leq A \), are replaced with calls to the set method \( \text{setX} \), where \( e \) is an argument and the left-expression \( le \) the target of the call. An assignment of the form \( le_1 := le.x \) is replaced with calls \( le.getX(le_1) \). As every access to \( x \) must be carried out by means of method calls, occurrences of \( le.x \) in an expression on the right-hand side of an assignment must be replaced with a local variable used as result argument in a call to \( \text{getX} \).
CHAPTER 5. CONTEXTUAL REFACTORINGS

Rule 5.8 (Encapsulate Field)

<table>
<thead>
<tr>
<th>class A extends C</th>
<th>class A extends C</th>
</tr>
</thead>
<tbody>
<tr>
<td>pub x : T; ads_a;</td>
<td>pri x : T; ads_a;</td>
</tr>
<tr>
<td>mts_a end</td>
<td>meth getX \triangleq (res arg : T \bullet arg := self.x)</td>
</tr>
<tr>
<td></td>
<td>meth setX \triangleq (val arg : T \bullet self.x := arg)</td>
</tr>
<tr>
<td></td>
<td>mts_a end</td>
</tr>
<tr>
<td></td>
<td>cds \bullet c</td>
</tr>
<tr>
<td></td>
<td>cds' \bullet c'</td>
</tr>
</tbody>
</table>

where

For all le : N, such that N \leq A, we have:

\[
\begin{align*}
\text{cds'} & \triangleq \text{cds}[\text{le.setX}(e), \text{le.getX}(le_1), \text{var } y : T \bullet \text{le.getX}(y); le_1 := e[y] \text{ end}, \\
& \quad \text{var } y : T \bullet \text{le.getX}(y); \text{pc}(e[y]) \text{ end/} \\
& \quad \text{le.x} := e, le_1 := \text{le.x}, le_1 := e[\text{le.x}], \text{pc}(e[\text{le.x}])]
\end{align*}
\]

\[c' \triangleq c[... \text{ (same substitutions above)}\]

provided

\[\leftrightarrow\] The type of x is basic;
\[\rightarrow\] getX is not declared in A nor in any superclass or subclass of A;
setX is not declared in A nor in any superclass or subclass of A;

This call must precede the assignment in which le.x appears.

Derivation. Direct access to a public attribute can be performed in various ways. For instance, it can be done by declaring a variable whose type is the class that has a public attribute, assigning an object of such class to the variable, and then selecting the attribute. Another way of directly accessing an attribute of an object is declaring such object as an attribute, initialising it in the new method, and selecting a public attribute of the object. For these reasons, in the derivation of this refactoring rule, we use the following derivation strategy.

1. Introduction of get and set methods for attributes that are going to be made private.
2. Replacement of direct accesses to attributes with calls to get and set methods.
3. Changing the visibility of attributes from public to private.

We begin the derivation by applying Law [90] (method elimination), from right to left, to introduce get and set method in class A. Notice that, in the refactoring rule, we required that these
methods are not declared in the whole hierarchy of class $A$.

```
class $A$ extends $C$
  pub $x : T$; $ads_a$;
  $mts_a$
end
```

$= \text{Law}\ 90 \ (\text{method elimination}) \ (r/l)(2x)$

```
class $A$ extends $C$
  pub $x : T$; $ads_a$;
  meth $getX \equiv (res \ arg : T \bullet arg := self.x)$
  meth $setX \equiv (val \ arg : T \bullet self.x := arg)$
  $mts_a$
end
```

Let us consider, in isolation, each possible way in which $le.x$ may occur in $mts_a$, $cds$, and $c$. We begin with the assignment $le.x := e$. This program is equivalent to the one that introduces the assumption $\{le \neq \text{null} \land le \neq \text{error}\}$ before the assignment to $le.x$. Notice that if $le$ is $\text{null}$ or $\text{error}$ we have that this assignment aborts, so the assumption is innocuous as it behaves as $\text{skip}$ when $le$ is not $\text{null}$ or $\text{error}$.

$= \text{Law}\ 48 \ (\text{innocuous assumption-writing}) \ (1/r)$

```
\{le \neq \text{null} \land le \neq \text{error}\} \ le.x := e
```

To introduce a method call we need a parameterised command. This can be introduced using Lemma $pcom$ value argument, passing the expression $e$ as a value argument.

$= \text{Lemma}\ 9$

```
\{le \neq \text{null} \land le \neq \text{error}\} \ (\text{val} \ arg : T \bullet \ le.x := arg)(e)
```

We have a program that is in the format required by the law for method call elimination. Also, the parameterised command we have is the same that defines the method $setX$ of class $A$. By applying this law, from right to left, we obtain a call to the method $setX$ of $A$.

$= \text{Law}\ 95 \ (\text{method call elimination}) \ (r/l)$

```
le.setX(e)
```
For assignments in the form \( le_1 := le.x \), we begin the derivation by introducing an innocuous assumption. The introduction of this assumption is justified by the fact that if \( le \) is \texttt{null}, the inspection of \( x \) gives \texttt{error} and the assignment aborts; if \( le \) is \texttt{error}, the inspection also gives \texttt{error}, aborting the assignment.

\[
le_1 := le.x
\]

= Law \[47\] \( \text{(innocuous assumption-reading)} \) (r/l)

\[
\{ le \neq \texttt{null} \land le \neq \texttt{error} \} \ le_1 := le.x
\]

The next step introduces a parameterised command applied to \( le_1 \), an argument that is passed by result.

= Lemma \[10\]

\[
\{ le \neq \texttt{null} \land le \neq \texttt{error} \} (\text{res arg : } T \bullet arg := le.x)(le_1)
\]

The parameterised command is the same as the one used in the definition of method \( getX \). Also, it is preceded by an assumption on \( le \) that requires that it is not \texttt{null} or \texttt{error}. We introduce a call to method \( getX \) of \( A \).

= Law \[95\] \( \text{(method call elimination)} \) (r/l)

\[
le.getX(le_1)
\]

Now, let us consider the case \( le_1 := e[le.x] \). The first is to introduce a local variable and assign to this variable the expression \( le.x \) at the end of the new local variable block.

\[
le_1 := e[le.x]
\]

= Law \[25\] \( \text{(var elim)} \) (r/l), Law \[30\] \( \text{(var-:= final value)} \) (r/l)

\[
\text{var } y : T \bullet le_1 := e[le.x]; \ y := le.x \ \text{end}
\]

Then, we move the assignment to \( y \) before the \( le_1 := e[le.x] \), and replace the expression \( le.x \) in \( e \) with the variable \( y \).

= Law \[59\] \( \text{(order independent assignment)} \), Law \[60\] \( \text{(assignment seq comp exp substitution)} \) (l/r)

\[
\text{var } y : T \bullet y := le.x; \ le_1 := e[y] \ \text{end}
\]
Notice that the assignment \( y := le.x \) is exactly the previous case we have dealt with. Following the same derivation steps, we end up with the following.

\[
\text{var } y : T \cdot le.getX(y); \quad le_1 := e[y] \end
\]

The case in which \( le.x \) appears in an expression that is argument of an application of a parameterised command is similar.

The attribute \( x \) is no longer accessed outside class \( A \), so we can change its visibility to private.

```java
class A extends C
    pub x : T; ads\_a;
    meth getX \equiv (res arg : T \cdot arg := self.x)
    meth setX \equiv (val arg : T \cdot self.x := arg)
mts\_a
end
```

= Law \( (\text{change visibility: from private to public}) \ (r/l) \)

```java
class A extends C
    pri x : T; ads\_a;
    meth getX \equiv (res arg : T \cdot arg := self.x)
    meth setX \equiv (val arg : T \cdot self.x := arg)
mts\_a
end
```

With this step we finish the derivation of the rule \( (\text{Encapulate Field}) \).

\( \square \)

In this rule we have dealt with just one attribute. The same approach is taken by Opdyke [69] and Fowler [42, p. 206]. As expected, self encapsulating an arbitrary number of attributes can be done by the application of the rule presented here as many times as the number of attributes to be encapsulated. Differently from Opdyke and Fowler, we require the type of the attribute being encapsulated to be basic.

In this rule we have considered the encapsulation of an attribute that can be read and written in the whole program. Reading an attribute is equivalent to inspecting the value of this attribute. It corresponds to getting the result of the attribute inspection by using a method call with a result argument. On the other hand, writing to an attribute can be done by passing the expression to be assigned to the attribute by means of value argument in a method call. Assigning an expression to a public attribute, and then inspecting the value of the attribute is equivalent to the sequential composition of calls to the set and get methods.
5.2 Further Contextual Refactoring Rules

In this section we present refactoring rules related to commands, without their corresponding derivations. For their derivations, see Appendix [B].

5.2.1 Add and Remove Parameter

Changing a method may require changing the parameters of a method in order to get information that was not necessary before. The refactoring rule \(\text{Add/Remove Parameter}\) (Rule 5.9) allows us to add to or remove a parameter from a method when we apply the rule from left to right or from right to left, respectively.

On the left-hand side of this rule, the method \(m\) of class \(A\) is defined by a parameterised command with parameters \(pds\). On the right-hand side, the method \(m\) presents the parameter \(pd\) as well as the already existing parameters \(pds\). Previous calls to \(m\) of the form \(le.m(a_1)\), where \(a_1\) is an expression of type \(T_1\), are replaced with calls of the form \(le.m(a_1, \text{exp})\), where \(le_1\) is an expression of a type compatible with the type of the parameter whose name is given by \(\alpha(pds)\).

The type of the left-expression \(le\) is given by any subclass of the topmost superclass of \(A\) that first introduces a definition of \(m\). Previous definitions of \(m\) in the hierarchy of \(A\) are replaced with a new definition with the additional parameter \(pd\).

To apply this rule in any direction, all attributes that appear in the definition of \(m\) must be public, and the type of \(le_1\) must be basic. This allows us to apply the law for method call elimination. In order to apply this rule from left to right, the name given by \(\alpha(pd)\) must not be a free variable in the definitions of \(m\) in the hierarchy of \(A\). In order to apply this rule from right to left, \(arg_2\) must not be used in any definition of \(m\).

5.2.2 Separate Query from Modifier

A method that returns values to its caller should not have any other effect other than just returning the required value. However, if a get method has observable side effects, that is, if it changes the state of the object that receives the method call, then the code should be separated so that the query is separated from the modifier. This is the purpose of the refactoring rule \(\text{Separate Query from Modifier}\) (Rule 5.10).

On the left-hand side of this rule, the method \(getSetX\) of class \(A\) returns the value of the attribute \(x\), but also assigns to such attribute the parameter \(arg_2\). In this way, any call to \(getSetX\) not only gets the current value of the attribute \(x\), but also assigns an expression to such attribute. On the right-hand side, the get and set methods for the attribute \(x\) are separated. Every call to \(getSetX\) that appeared on the left-hand side is replaced with a sequence of calls to \(getX\) and \(setX\), in this order.
5.2. FURTHER CONTEXTUAL REFACTORING RULES

**Rule 5.9 (Add/Remove Parameter)**

\[
\begin{array}{|c|}
\hline
\text{class } A \text{ extends } C \\
\text{ads}_a \\
\text{meth } m \equiv (pds \cdot c) \\
mts_a \\
\text{end} \\
cds_1 cds_2 \cdot c \\
\hline
\end{array}
\]

\[
\begin{array}{|c|}
\hline
\text{class } A \text{ extends } C' \\
\text{ads}_a \\
\text{meth } m \equiv (pds; \ pd \cdot c) \\
mts'_a \\
\text{end} \\
cds'_1 cds_2 \cdot c' \\
\hline
\end{array}
\]

where

For all \( le : N \), such that \( N \leq Z \), where \( Z \) is the topmost class of \( A \) that first introduces a definition of \( m \), we have:

\[
mts'_a \equiv mts_a[\textbf{self}.m(a_1, le_1)/\textbf{self}.m(a_1)]
\]

\[
cds'_1 \equiv cds_1[\textbf{meth } m \equiv (pds; \ pd \cdot c), le.m(a_1, le_1)/\textbf{meth } m \equiv (pds \cdot c), le.m(a_1)]
\]

\[
c' \equiv c[le.m(a_1, le_1)/le.m(a_1)]
\]

provided

(\( \leftrightarrow \)) All attributes that appear in all definitions of \( m \) are public;

\( le_1 \) is an expression of the same type as the parameter whose name is given by \( \alpha(pd) \);

the type of \( le_1 \) is basic;

(\( \rightarrow \)) \( \alpha(pd) \) is not a free variable in the method \( m \) of the topmost superclass of \( A \) that first introduces a definition of \( m \) nor in any redefinition of \( m \) in subclasses of such topmost class;

\( cds_1 \) only contains classes that define a method named \( m \) and classes that call method \( m \); \( cds_2 \) contains all other classes;

(\( \leftarrow \)) \( \alpha(pd) \) is not used in the method \( m \) of the topmost superclass of \( A \) that first introduces a definition of \( m \) nor in any redefinition of \( m \) in subclasses of such topmost class.

\( cds'_1 \) only contains classes that define a method named \( m \) and classes that call method \( m \); \( cds_2 \) contains all other classes;
Rule 5.10 (Separate Query from Modifier)

\[
\begin{array}{ll}
\text{class } A \text{ extends } C \\
\text{pri } x : T; \text{ ads}_{a}; \\
\text{meth } \text{getSetX} \equiv (\text{val } \text{arg}_1 : T; \\
\text{res } \text{arg}_2 : T \bullet \text{arg}_2 := \text{self}.x; \\
\text{self}.x := \text{arg}_1) \\
\text{mts}_a \\
\text{end} \\
\text{cds} \bullet c
\end{array}
\]

\[
\begin{array}{ll}
\text{class } A \text{ extends } C \\
\text{pri } x : T; \text{ ads}_{a}; \\
\text{meth } \text{getX} \equiv (\text{val } \text{arg} : T \bullet \\
\text{arg} := \text{self}.x) \\
\text{meth } \text{setX} \equiv (\text{val } \text{arg} : T \bullet \\
\text{self}.x := \text{arg}) \\
\text{mts}'_a \\
\text{end} \\
\text{cds}' \bullet c'
\end{array}
\]

where

\[
\begin{align*}
\text{mts}'_a & \equiv \text{mts}_a[\text{le}.\text{getX}(\text{a}_2); \text{le}.\text{setX}(\text{a}_1)/\text{le}.\text{getSetX}(\text{a}_1, \text{a}_2)] \\
\text{cds}' & \equiv \text{cds}[\text{le}.\text{getX}(\text{a}_2); \text{le}.\text{setX}(\text{a}_1)/\text{le}.\text{getSetX}(\text{a}_1, \text{a}_2)] \\
c' & \equiv c[\text{le}.\text{getX}(\text{a}_2); \text{le}.\text{setX}(\text{a}_1)/\text{le}.\text{getSetX}(\text{a}_1, \text{a}_2)]
\end{align*}
\]

provided

\[
\begin{align*}
(\leftrightarrow) & \text{ le is any left expression of a type } B \text{ such that } B \leq A; \\
& \text{a}_1 \text{ and } \text{a}_2 \text{ have basic types;} \\
(\rightarrow) & \text{getX and setX are not declared in mts}_a \text{ nor in any superclass or subclass of } A; \\
(\leftarrow) & \text{getSetX is not declared in mts}_a \text{ nor in any superclass or subclass of } A
\end{align*}
\]

To apply this rule from left to right, the methods \text{getX} and \text{setX} must not be declared in mts$_a$ nor in any subclass or superclass of A. Also, the types of a$_1$ and a$_2$ must be basic. To apply this rule from right to left, the method \text{getSetX} must not be declared in mts$_a$ nor in any subclass or superclass of A. For both directions, we require the target expressions in the calls to the method \text{getSetX} (\text{getX} and \text{setX}) to be a subtype of A.

5.2.3 Encapsulate Downcast

In order to prevent clients from having to downcast the object that is obtained from a call to a method, we can downcast the object within the method. In this way, clients are provided with the most specific type of the object. This is expressed by the rule (Encapsulate Downcast) (Rule 5.11).

On the left-hand side of this rule, a left expression \text{le}$_1$ is assigned to the parameter \text{arg} of the method \text{m}. The type of \text{le}$_1$ is \text{N}, which is a subclass of \text{M}, the type of the parameter \text{arg}. In the program there may be occurrences of calls to the method \text{m}. As this method has a result
Rule 5.11 (Encapsulate Downcast)

\[
\begin{array}{ll}
\text{class } A \text{ extends } C \\
\quad \text{adss}_A; \\
\quad \text{meth } m \triangleq (\text{res } \text{arg} : M \bullet \\
\quad \quad \text{arg} := \text{le}_1) \\
\quad \text{mts}_a \\
\quad \text{end} \\
\quad \text{cds} \bullet \text{c}
\end{array}
\] =

\[
\begin{array}{ll}
\text{class } A \text{ extends } C \\
\quad \text{adss}_A; \\
\quad \text{meth } m \triangleq (\text{res } \text{arg} : N \bullet \\
\quad \quad \text{arg} := (N) \text{le}_1) \\
\quad \text{mts}_a \\
\quad \text{end} \\
\quad \text{cds}' \bullet \text{c'}
\end{array}
\]

where

\[
cds' \triangleq cds[\text{var } \text{le}_2 : N \bullet \text{le.m(le}_2); \ y := \text{le}_2 \text{ end}] / \\
\quad \text{var } \text{le}_2 : M \bullet \text{le.m(le}_2); \ y := (N) \text{le}_2 \text{ end}]
\]

\[
c' \triangleq c[\text{var } \text{le}_2 : N \bullet \text{le.m(le}_2); \ y := \text{le}_2 \text{ end}] / \\
\quad \text{var } \text{le}_2 : M \bullet \text{le.m(le}_2); \ y := (N) \text{le}_2 \text{ end}]
\]

\[
\text{le} : B, \text{ such that } B \leq A;
\]

provided

\[
(\leftrightarrow) \ m \text{ is not defined in any superclass nor redefined in any subclass of } A. \\
\quad M, N \in cds', \ N \leq M, \text{ and } cds, A \triangleright \text{le}_1 : N. \\
\quad y : T, \text{ such that } T \leq_{cds} N;
\]

\[
(\rightarrow) \ M, N \in cds, \ N \leq M, \text{ and } cds, A \triangleright \text{le}_1 : N.
\]

parameter, callers of such method must give a left-expression as argument. The argument \(\text{le}_2\) is of type \(M\). The method call is composed with an assignment to a variable \(y\) declared to be of type \(N\). This assignment involves a downcast of the variable \(x\) to \(N\).

On the right-hand side, the downcast is placed inside the method \(m\). The type of the parameter \(\text{arg}\) is changed to \(N\). In the program, the assignment to \(y\) does not need a cast of \(\text{le}_2\) to \(N\), because it is now present inside method \(m\). The type of the variable \(\text{le}_2\) is also changed to \(N\).

To apply this rule in any direction, we require that the method \(m\) is not declared in any superclass or subclass of \(A\). Altering the signature of \(m\) in \(A\) would require changing the signature in all superclasses and subclasses of \(A\). The variables \(x\) and \(y\) are declared to be of types \(M\) and \(N\), respectively.

### 5.3 Conclusions

In this chapter we presented a list of contextual refactoring rules, based on the work in [42], along with their derivation. Differently from the refactoring rules presented in Chapter [4], which are compositional in the sense that they do not affect the classes that constitute the context of the
class being refactored nor the main program, the refactoring rules we presented in this chapter change other classes in the program and, possibly, the main command.

Similarly to what we have done in Chapter 4, we can classify some of the refactorings as basic, since they can be used in the derivation of other refactorings, and do not have a derivation that relies on other refactorings. We consider rules \( \langle \text{Change clientship: from subclass to superclass} \rangle \), \( \langle \text{Pull Up/Push Down Field} \rangle \), \( \langle \text{Rename Method} \rangle \), \( \langle \text{Parameterise Method} \rangle \), and \( \langle \text{Encapsulate Field} \rangle \) to be basic refactorings. The refactoring rule \( \langle \text{Extract Superclass} \rangle \) is not basic as its derivation relies on the rule \( \langle \text{Pull Up/Push Down Field} \rangle \); and the derivation of \( \langle \text{Collapse Hierarchy - Subclass} \rangle \) relies on the application of \( \langle \text{changing clientship: from subclass to superclass} \rangle \).
Chapter 6

Refactoring towards Patterns

In this chapter we present design and architectural patterns. Design patterns present solutions to recurrent problems in object-oriented systems development. They are descriptions of communicating objects and classes that are customised to solve a general design problem in a particular context [43]. Architectural patterns define the overall shape and structure of software applications [76]. Our intention is not to present a design (or architectural) pattern as a rule, but to show how, by the application of refactoring rules, we can transform a system in order to obtain a design that conforms to a pattern. The particular refactoring rules to be applied depends on the system being currently designed. We do not limit the structure of the systems of interest and, for this reason, we have no rule for describing design patterns.

We present the pattern Facade [43] and a layered architecture, an architectural pattern, as goals of the derivations of two different systems through the application of refactoring rules. The Facade pattern is a simple structural design pattern. The layered architecture we present has already been used in practical software development [88].

6.1 The Facade Pattern

In this section we show how to obtain a design that is in accordance with a design pattern by the application of transformation rules. The Facade pattern [43] is used to introduce a class that is responsible for providing a unified interface to a set of interfaces in a subsystem [43].

We can reduce the complexity of a system by structuring it into subsystems. The communication between subsystems can be reduced if we use a facade that provides a single interface to the whole subsystem. This facilitates the use of the subsystem; also, changes that do not affect the interface of the facade have no impacts on clients of the facade.

Here, we present an example that involves two clients of two different classes of a system: see Figure 6.1. The classes ClientA and ClientB are clients of the classes Subsys1 and Subsys2, respectively. The method p of class ClientA calls the method m with an object s1 of class Subsys1 as target. The argument \( a_m \) represents expressions whose types are compatible with the formal
class ClientA
  pri s1 : Subsys1;
  meth p \equiv (pds_p \circ c_p[\text{self}.s1.m(a_m)])
end
class Subsys1
  pub x : T; adsSubsys1;
  meth m \equiv (pds_m \circ c_m[x])
  mtsSubsys1
end

cds \circ c

class ClientB
  pri s2 : Subsys2;
  meth q \equiv (pds_q \circ c_q[\text{self}.s2.n(a_n)])
end
class Subsys2
  pub y : T; adsSubsys2;
  meth n \equiv (pds_n \circ c_n[y])
  mtsSubsys2
end

Figure 6.1: The system before refactoring

parameters of the method \( m \) of the class Subsys1.

The class Subsys1 declares the attribute \( x \) of type \( T \), among others. The method \( m \) of this class uses the attribute \( x \). We assume that in method \( m \) there are no calls to the other methods \( mts_{Subsys1} \) of this class. Similarly, ClientB calls the method \( n \) of Subsys2 using \( s_2 \) as a target. Class Subsys2 declares the attribute \( y \) of type \( T \) that is used by the method \( n \). We also assume that \( x \) and \( y \) are not declared by any subclass of ClientA and ClientB, respectively; and the methods \( m \) and \( n \) are not declared by any subclass of ClientA and ClientB, respectively. Also, we assume that there is no class named Facade in the sequence of class declarations \( cds \).

After refactoring, the classes ClientA and ClientB and the subsystem classes must conform to the Facade pattern. In other words, the classes ClientA and ClientB must be clients of a facade class. Calls to methods of the facade are forwarded to classes of the subsystem. All previous direct method calls to objects of the classes Subsys1 and Subsys2 will have as target objects of the facade. These objects just forward the calls to objects of the classes Subsys1 and Subsys2. In this way, the classes of the system will not be known by clients.

**Derivation.** We begin the derivation by eliminating the clientship relation that classes ClientA and ClientB have with classes Subsys1 and Subsys2, respectively. We apply, from left to right, rule (Clientship Elimination) (Rule 4.7). Class ClientA, after the application of this law, declares the attribute \( x \) that appears in the method \( m \) of class Subsys1 and the method \( m \) itself. Similar changes are carried out in class ClientB.
class ClientA
    pri s1 : Subsys1;
    meth p \triangleq (pds_p \bullet c_p[\text{self}.s1.m(a_m)])
end
class Subsys1
    pub x : T; adsSubsys1;
    meth m \triangleq (pds_m \bullet c_m[x])
end

class ClientB
    pri s2 : Subsys2;
    meth q \triangleq (pds_q \bullet c_q[\text{self}.s2.n(a_n)])
end
class Subsys2
    pub y : T; adsSubsys2;
    meth n \triangleq (pds_n \bullet c_n[y])
end

We now introduce a class named Facade with the attributes and methods of the subsystem classes that are present in the client classes. So, the class Facade declares the attributes x and y, and also m and n.

\[
= \text{\textit{(Clientship Elimination) (1/r) (2x)}}
\]

class ClientA
    pri x : T;
    meth p \triangleq (pds_p \bullet c_p[\text{self}.m(a_m)])
    meth m \triangleq (pds_m \bullet c_m[x])
end
class ClientB
    pri y : T;
    meth p \triangleq (pds_p \bullet c_p[\text{self}.n(a_n)])
    meth n \triangleq (pds_n \bullet c_n[y])
end

We change the classes ClientA and ClientB to be clients of the class Facade. This is done by applying twice the rule \textit{(Clientship Elimination)}, from right to left. After this, the class ClientA declares an attribute f of type Facade that is initialised with an object of type Facade. Calls to the method m of the class Facade have as target attribute f. Similar changes apply to class ClientB.

\[
= \text{\textit{(Clientship Elimination) (r/l) (2x)}}
\]

class ClientA
    pri f : Facade;
    meth p \triangleq (pds_p \bullet c_p[\text{self}.m(a_m)])
    new \triangleq \text{self}.f := \text{new} \text{Facade}
end
class ClientB
    pri f : Facade;
    meth q \triangleq (pds_q \bullet c_q[\text{self}.n(a_n)])
    new \triangleq \text{self}.f := \text{new} \text{Facade}
end
The class Facade provides the same methods of the subsystem classes that were used by the client class in the beginning of the derivation. In fact, the attributes and methods of Facade are the same as those of the subsystem classes, except for methods that were not used by clients and attributes that appear in these methods.

We now change the class Facade to be just a delegate class: calls to its methods are forwarded to adequate subsystem classes. We apply rule (Delegation Elimination) (Rule 4.8) twice, from right to left. The method calls have as target the attributes \( s_1 \) and \( s_2 \) of type \( \text{Subsys}1 \) and \( \text{Subsys}2 \), respectively, declared in Facade. These attributes are initialised with objects of this type.

This finishes the derivation. Now the client classes call methods of the class Facade, which forwards such calls to the subsystem classes. Consequently, client classes interact only with the facade class, and not with subsystem classes.

### 6.2 A Layered Architecture

Programs structured according to a layered architecture support enhancements and reuse [76]. Therefore, refactoring of object-oriented programs should be conducted, whenever necessary, to obtain a final program with a layered architecture [6]. Here we aim at a layered architecture originally designed for the integration of object-oriented programming languages with relational databases [88].

The main purpose of the architecture is to avoid, as much as possible, data storage and retrieval to be mixed with code that implements the functional requirements of a system. To achieve this
6.2. A LAYERED ARCHITECTURE

Figure 6.2: The four-layer architecture

purpose, classes are separated into two groups: classes that describe the objects required by the modelling of the systems’ (functional) requirements; and classes for data storage and manipulation. The connection between classes of these groups is defined by interfaces. Classes of the first group are independent of the effective implementation of the data storage and manipulation operations, because these classes do not rely on knowledge of the data structure used for storage, but only on the methods defined by an interface. Classes of the first group contain what we call business code, which implements the functional requirements of a system. Classes of the second group, however, know how the persistence operations are implemented and depend on the data structure used for storage. They contain data code for manipulating data structures.

More generally, this architecture is viewed as being composed by four independent layers (Figure 6.2). Classes of the first group constitute the business layer, and classes of the second group constitute the data layer. The classes that contain code for communication among subsystems compose the communication layer; and classes that implement the user interface compose the interface layer. Here we concentrate on structuring the application into the business and data layers. The classes of the business layer are divided into three groups: basic classes, representing basic entities; collection classes, representing groups of basic objects; and control classes, which define the control flow of functional requirements. The collection classes include methods for adding, searching, and removing items of a collection, and for invoking typical operations of business objects. If the facade pattern [43] is adopted, a single (control) class synthesises the functionality of the application.

From a poorly structured system we intend, by means of data refinement and application of refactoring rules, to reach a well-structured system that fits the layered architecture described here. In the next section we present a new refactoring rule we identified during the introduction of the architectural pattern described from a poorly structured system.

6.2.1 A New Refactoring Rule

Rule (Interface Clientship) (Rule 6.1) introduces clientship between a class $B$ and a class $D$, which models an interface. The class $D$ is adequately extended in order to introduce a concept initially described in class $B$. By applying this rule, we can later provide different implementations of this
### Rule 6.1 \(\text{(Interface Clientship)}\)

<table>
<thead>
<tr>
<th>Class B extends A</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{pri} (x : T; \ ads_b)</td>
</tr>
<tr>
<td>\textbf{meth} (m \triangleq (pds_m \cdot c_m[c'_m]))</td>
</tr>
<tr>
<td>\textbf{mts}_b</td>
</tr>
<tr>
<td>\textbf{end}</td>
</tr>
</tbody>
</table>

\[\equiv_{\text{cds,c}}\]

<table>
<thead>
<tr>
<th>Class B extends A</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{pri} (d : D; \ ads_b)</td>
</tr>
<tr>
<td>\textbf{meth} (m \triangleq (pds_m \cdot \text{self.d.m}))</td>
</tr>
<tr>
<td>\textbf{new} (\triangleq \text{self.d} := \text{new E})</td>
</tr>
<tr>
<td>\textbf{mts}_b</td>
</tr>
<tr>
<td>\textbf{end}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class D</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{meth} (m \triangleq (pds_m \cdot \text{abort}))</td>
</tr>
<tr>
<td>\textbf{end}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class E extends D</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{pri} (x : T; )</td>
</tr>
<tr>
<td>\textbf{meth} (m \triangleq (pds_m \cdot c'_m))</td>
</tr>
<tr>
<td>\textbf{end}</td>
</tr>
</tbody>
</table>

provided

\((-\rightarrow)\)

1. \(D\) and \(E\) are not declared in \(\text{cds}\);
2. \(\text{self}.x\) does not appear in \(\text{mts}_b\);

\((-\leftarrow)\)

1. Classes \(D\) and \(E\) are not referred to in \(\text{cds}\) or \(c\).

On the left-hand side of this rule, class \(B\) declares an attribute \(x\), and a method \(m\) (among other methods in \(\text{mts}_b\)). On the right-hand side, we introduce class \(D\) whose method \(m\) is defined by a parameterised command with body \texttt{abort}. Such a class models a Java interface. A Java interface contains a set of signatures of abstract methods; by defining the method bodies to be \texttt{abort}, we give them the most abstract definition.

On the right-hand side of this rule, class \(E\) extends \(D\), declares an attribute \(x\) and redefines \(m\). Class \(B\) is a client of \(D\), and initialises its attribute \(d\) with an object of class \(E\), avoiding in this way program abortion. Command \(c_m\) is defined in terms of the call \texttt{self.d.m}.

For the application of rule \(\langle\text{Interface Clientship}\rangle\), from left to right, we require that classes \(D\) and \(E\) are not declared in \(\text{cds}\). Also, attribute \(x\) should not be accessed in methods of \(B\) other than \(m\). In order to apply this rule from right to left, there must be no references to classes \(D\) and \(E\), except in class \(B\).

We can derive this refactoring rule in the following way. We introduce classes \(D\) and \(E\) using Law 82 (\textit{class elimination}) from right to left. However, attribute \(x\) of class \(E\) must be public, initially. The next step is the data refinement of class \(B\). Instead of using the attribute \(x\) of class \(B\), we use \(x\) of class \(E\). This is realised by the application of Law 108 (\textit{private attribute-coupling invariant}). For this reason, we introduce class \(E\) with a public attribute \(x\). Accesses to \(x\) are now realised by calling method \(m\) of \(E\).
This refactoring cannot be fully derived from other refactorings. It cannot be simply derived by the application of rule $\langle$Extract Class$\rangle$ (Rule 4.6), as $B$ is a client of class $D$ whose methods are defined with command abort, and this is not a class extraction.

### 6.2.2 The Architectural Pattern Derivation

Our refactoring strategy consists of three stages. Each stage involves the introduction of new classes, and data and algorithmic refinement of an already existing class. Data refinement typically involves the introduction of new attributes to obtain information hiding or to restructure a class in order to improve reuse. From the first stage (Stage 1) to the last one (Stage 3), the program changes from a poorly structured one to a well-structured program according to the layered architecture described above. The developer should identify in which stage of development its program is, and apply refactoring from this stage to the last one. The main reason for dividing the development in stages is the simplification of data refinement.

**Stage 1**

In the first stage, we deal with a class that is monolithic. Data and business code are mixed. The purpose of this stage is to identify basic entities in such a monolithic description, and model each entity as a separate class, with its relevant attributes and methods.

The general form of a monolithic class is given below. The attribute $aTable$ is used to model a database. The type of $aTable$ is given by a partial injective function (symbol $\mapsto \rightarrow$) from a type $T_1$ to a type $T_2$. The method $update$ is used to update a record of the table. It takes the record identifier $n$ and its new value $m$ as arguments; it also has a result parameter $rp$: a string that reports whether the update was successful or not. First, the method $update$ checks if $n$ belongs to the domain of $aTable$, which is a business rule. If it does, then $aTable$ is updated at position $n$ with the expression $\text{exp}$ in which there may be occurrences of the expression $m$, a data operation. The symbol $\oplus$ in the body of method $update$ stands for function overriding. The class $Application$ also presents methods for inserting new elements in $aTable$, deleting already existing elements, and a method for inspecting the value associated with a given element in the domain of the table.

```plaintext
class Application
  pri aTable : $T_1 \rightarrow T_2$: ...

  meth update $\triangleq$ (val n, m : $T_1$, $T_2$: res rp : string $\bullet$
    if (n $\in$ dom aTable) $\rightarrow$ self.aTable := self.aTable $\oplus \{ n \mapsto \text{exp}[m]\};$
      rp := "Updated"
    $\mid$ (n $\notin$ dom aTable) $\rightarrow$ rp := "Not Updated"
  fi

  $\cdots$
end
```
At the end of this stage, we want to have the concept (class) which characterises the elements of aTable separated. The class Application is transformed as shown in Figure 6.3.

By using Law 82 (class elimination), from right to left, we introduce the class BasicEntity that captures the concept introduced by the domain and range of the attribute aTable. This class provides get and set methods for the attribute at2, because it is changed along the lifetime of objects of the class BasicEntity. The value of attribute at1 is usually established at object creation and not modified along the lifetime of an object of BasicEntity. This reflects the fact that attribute at1 is used to identify which attribute at2 is associated with it, in the class Application. The class BasicEntity represents basic objects necessary to implement the functional requirements of the system, it separates the concept of pair initially present in the class Application.

```
class BasicEntity
  pri at1, at2 : T1, T2;
  meth setAt2 ≡ (val m : T2 • self.at2 := m)
  meth getAt2 ≡ (res m : T2 • m := self.at2)
  new ≡ (val n, m : T1, T2 • self.at1 := n; self.at2 := m)
end
```

The next step is to prepare the class Application for data refinement. This preparation consists of applications of Law 78 (simple specification) to assignments to the attribute aTable. The application of this law changes assignments into corresponding specification statements.

```
class Application
  pri aTable : T1 ⇐ T2; ...
  meth update ≡ (val n, m : T1, T2; res rp : string •
    if (n ∈ dom aTable) → self.aTable : [self.aTable = self.aTable ⊕ {n → exp[m]}];
    rp := "Updated";
    [] (n ∉ dom aTable) → rp := "Not Updated"
  fi)
  ...
end
```

Afterwards, a new (private) attribute is introduced in the original class Application. A coupling invariant relates the new attribute with the old one. From the point of view of data refinement, the new variables are concrete variables. We use Law 108 (private attribute-coupling invariant) to add an attribute data of type seq BasicEntity (sequence of BasicEntity) to Application. The coupling invariant CIStage1 is used to relate the attribute aTable and data, which is a sequence of objects of class BasicEntity.

```
CIStage1 ≡ aTable = \{i : 0..#data - 1 • data(i).at1 ↦ data(i).at2\} ∧
(∀ i, j : 0..#data - 1 • i ≠ j ⇒ data(i).at1 ≠ data(j).at1)
```
This coupling invariant guarantees that \( aTable \) is formed by mappings relating the values in each object present in \( data \). Moreover, the values for the attribute \( at_1 \) of the objects in \( data \) must be distinct.

The application of Law \(^{[108]}\langle \text{private attribute-coupling invariant} \rangle \) to the class \( Application \) changes the methods of this class according to the laws of data refinement \(^{[64]}\). Specification statements and guards, as expected, must assume the coupling invariant. The class \( Application \) now is as follows.

```
class Application
  pri data : seq BasicEntity;
  pri aTable : T1 \mapsto T2; ...
  meth update \( n,m : T_1, T_2; \ res \ rp : \text{string} \bullet \)
     if (\( n \in \text{dom} \ aTable \land CI \)) \rightarrow self.aTable : [CI, self.aTable = self.aTable \oplus \{n \mapsto \text{exp}[m]\} \land CI];
     \( rp := "Updated" \)
     \( \emptyset (p = \text{null}) \rightarrow rp := "Not_Updated" \)
     fi
  end

  meth search \( j : \text{int}; \ res \ obj, pos : \text{Pair}, \text{int} \bullet \ldots \)
  ...
end
```

Figure 6.3: The class \( Application \) in the end of Stage 1

Now we refine the class \( Application \) in order to remove references to the abstract variable \( aTable \). First, by applying Law \(^{[90]}\langle \text{method elimination} \rangle \), from right to left, we introduce method \( search \) in class \( Application \). This method returns an object of type \( BasicEntity \) whose attribute \( at_1 \) has the same value as \( n \). We proceed with algorithmic refinement of specification statements and guards. Such refinement is carried out for all methods of \( Application \). The method \( update \), after refinement, is as follows.
class Application
    pri collect : BusinessCollection;
    meth update ≜ (val n, m : T₁, T₂; res rp : string •
        collect.update(n, m, rp)) end
    new ≜ collect := new BusinessCollection()
    ...
end

Figure 6.4: The class Application in the end of Stage 2

meth update ≜ (val n, m : T₁, T₂; res rp : string •
    var p, i : BasicEntity, int • self.search(n, p, i);
    if (p is BasicEntity) → p.setAt2(exp[m]); self.data(i) := p; rp := "Updated"
    [] (p = null) → rp := "Not Updated"
end)

The method update uses two local variables: p and i (see Figure 6.3). First it calls the new method search to get, in p, the object identified by n, and in i, its index in data. If p is null, then there is no element identified by n, and this is reported through rp. If p is not null, then a call to a method of BasicEntity is used to set its value to that of an expression involving the parameter m of update, the sequence data is updated, and success is reported.

In the development, the variables p and i are introduced along with a specification statement that is refined to introduce a call to the method search, which is also called by the other methods of class Application, those used for insertion and deletion of objects in data.

Calls to methods of BasicEntity have to replace the direct access to the (abstract) attribute aTable which, after that, can be removed from the class Application. This is done by refining the methods of Application using laws similar to those presented by Morgan [64]. By using Law S3 (attribute elimination), we remove the attribute aTable from Application.

Stage 2

In this stage, we have a program in which different concepts are described in different classes. In the end, our purpose is to have the class Application as a facade to the system, where the bodies of its methods basically delegate responsibilities to the business collections through method calls.

We obtain the classes Application (Figure 6.4) and BusinessCollection (Figure 6.5) in two steps. The first step is the introduction of class BusinessCollection. Then, we make class Application just a delegating class by using (Delegation Elimination) (Rule 4.8), from right to left.
6.2. A LAYERED ARCHITECTURE

```plaintext
class BusinessCollection
    pri data : seq BasicEntity;
    meth update ≡ (val n, m : T1, T2; res rp : string •
        var p, i : BasicEntity, int • self.search(n, p, i);
        if (p is BasicEntity) → p.setAt2(m); self.data(i) := p; rp := "Updated"
        [] (p = null) → rp := "Not Updated"
    fi
    end )
    meth search ≡ (val j : T1; res obj, pos : BasicEntity, T1 • ... )
    ...
end
```

Figure 6.5: The class BusinessCollection in the end of Stage 2

Stage 3

At this point, the collection class and the persistence mechanism are still interwoven. This hinders reuse and extensibility, because if the persistence mechanism is changed, part of the system must be redesigned. The business code that can be reused, when adapting the persistence mechanism, should be separated from data code. This is the purpose of this stage.

We proceed with the application of refactoring rule \( \langle \text{Interface Clientship} \rangle \) (Rule 6.1). This results in classes BusinessCollection (Figure 6.6), and RepositoryClass and RepositoryClassRef, which are presented in Figure 6.7. Class BusinessCollection now is client of RepositoryClass and initialises attribute rep with an object of class RepositoryClassRef. This attribute is also target of call to methods of RepositoryClassRef. Class RepositoryClass defines an interface between the BusinessCollection and the class that deals with the persistence mechanism. A class like RepositoryClass is similar to a Java interface. Class RepositoryClassRef implements the access to the data structure originally defined in class BusinessCollection.

The use of an interface provides independence between the collection and the repository classes. We can change the repository class, for instance, from a class that uses a list to one that uses a tree, with minimal impact on the collection class. Only the initialiser of this class needs to change to create an object of the new implementation of the repository. We now have a program structured according to the architecture described in Section 6.2.

The strategy described here can also be used to obtain systems structured according to a three-layer architecture [19]. The main difference between our architecture and the three-layer one is the presence of the communication layer. We have developed an example of a bank application [31], which was informally presented by Viana and Borba [88], following the strategy we have just presented. However, the derivation presented in [31] is based on the use of small transformation steps.
6.3 Conclusions

In this chapter we exemplified how, from a specific sequence of class declarations, we can achieve a design in accordance with a design pattern. We also presented how to achieve a well-structured program from a poorly-structured one. The final program is structured according to a layered architecture, which provides independence between business and data layers.

In this chapter we also presented a new refactoring rule that was needed in the transformation of a poorly structured system into a system designed according to a layered architecture. This refactoring was identified from a revision of the development presented in [31]. Indeed, this is an indication that, in practice, new refactorings will always arise.

Surprisingly, it was not possible to obtain such derivation just from refactoring rules presented in Chapters 4 and 5. In particular, refactoring \( \langle \text{Interface Clientship} \rangle \) could not be derived from refactoring \( \langle \text{Extract Class} \rangle \), which would seem to be compulsory in such derivation. This is because we cannot change a class to be a client of another whose methods are defined with \texttt{abort} (representing methods of an interface).
6.3. CONCLUSIONS

In his refactoring catalog, Fowler [42] suggests refactorings that allow, for instance, separating business logic from the user interface code. Such refactorings are called “big refactorings”. The refactoring *Separate Domain from Presentation* [42, p. 370] could be applied to obtain the set of classes in the end of Stage 2, when considering the case in which data in the user interface should be part just of the business logic. This is expressed by the attribute *data* of class *Application* in the end of Stage 1. Fowler also considers situations in which data is used only for user interface purposes, and data is used both by the user interface and the business logic. On the other hand, to achieve classes of Stage 3, it would be necessary to use “small refactorings” such as *Extract Class*, *Move Method*, *Push Down Method*, and *Extract Interface*.

Design patterns are not presented as rules. The reason for this is that only the right-hand side could effectively be characterised; the left-hand side should correspond to any system that needs to be refactored to achieve a structure according to a design pattern. In fact, one goal of refactoring a system is to have such a system in accordance with a design pattern [43]. Describing a design pattern as a rule would restrict its application just to systems that match the left-hand side of the rule. It would be necessary to write rules for all situations that result in a design pattern that can be achieved from an existing design.

A formal description of design patterns has already been provided by Flores et al. [41] and Eden [38]. They use different notations for the description of object-oriented design patterns. In particular, Flores et al. present elements that constitute a general object-oriented design and their formal model, and explain how to formally link a design to a pattern, which has a formal model and properties that must be satisfied. However, as already said, we transform a particular design with the aim of obtaining a new design according to a design or architectural pattern.

Lano et al. [53] formally justify design patterns by relating two sets of classes, the “before” and “after” systems. The “after” system consists of a collection of classes organised according to a pattern. The proof that the “after” system is an extension of the “before” one is given via a suitable interpretation that is proved for selected axioms. Differently, we adopt a transformational approach that is based on rules, and not on the verification that a system extends the original system.
In order to preserve behaviour, changes to a software must be carried out in a disciplined way. One way of imposing a discipline to software transformation is by using formal methods either in a guess-and-verify approach, which requires a transformation to be proved to preserve the semantics of the original system, or in a constructive approach, in which case a system is transformed into another one by the application of laws. In this thesis, we adopted a constructive approach to software transformation based on algebraic rules. These rules correspond mainly to some of the refactoring practices documented by Fowler and Opdyke.

We presented a set of refactoring rules along with their derivation in terms of basic laws of object-oriented programming, and laws of data refinement. These laws were expressed in the language rool, which includes recursive classes, visibility control, dynamic binding, and recursive methods. Also, it includes specification constructs from the refinement calculi. It has a copy semantics rather than a reference semantics. The semantics of ROOL is given by means of predicate transformers. The laws were proved correct against this semantics and simulation laws. The refactoring rules presented deal, mainly, with the structure of classes and their relationships. The laws of classes (see Appendix E) are essential for the derivation of refactoring rules. Indeed, these laws play the role of the basic refactorings that Opdyke presents in his thesis. In fact, these laws can be viewed as the most basic refactorings that can be used to define other refactorings. Moreover, these laws comprise a comprehensive set, which is useful for reducing programs to a normal form described in terms of a restricted subset of ROOL. On the other hand, we have not used all the laws of commands (see Appendix D); we have used mainly laws that deal with introduction of class invariants as assumptions, and introduction of parameterised commands.

Besides the formalisation of existing refactoring practice, we have also presented some new refactoring rules that are used for the derivation of other refactoring rules and patterns. We have presented the formalisation of one design pattern and of one architectural pattern. However, differently from the presentation of the refactoring rules, both kinds of patterns are not presented as rules. The reason is that we can characterise only the final result (what would be the right-hand side of a rule), whereas the left-hand side would correspond to any system that needs to be
refactored to achieve a system that is in accordance with a pattern. Writing a rule would restrict its application to systems that match the left-hand side. Instead of a single rule application, we achieve a system that conforms to a pattern by means of refactoring rule applications. Surprisingly, we identified another refactoring rule.

The derivation of refactoring rules may also be based on other refactoring rules. For instance, one of the steps of the derivation of ⟨Extract Superclass⟩ is the application of the refactoring ⟨Pull Up/Push Down Field⟩. Consequently, we consider the ⟨Pull Up/Push Down Field⟩ to be a basic refactoring, while the refactoring ⟨Extract Superclass⟩ is a derived one, in the sense that its derivation uses not only laws of classes, but also refactoring rules.

The proofs of the refactoring rules by the application of laws of object-oriented programming confirm the effectiveness of object-oriented refinement calculi for the development of programs. To our knowledge, this is the first time an object-oriented refinement calculus is applied to prove program transformations like refactorings. Moreover, the laws of ROOL provide guidance on the construction of programs.

The formalisation of refactorings revealed new laws of ROOL. Some of these laws were previously known for imperative programming, but were not initially proposed for ROOL. Examples of these laws are those that deal with introduction of parameterised commands from a non-parameterised command. Other laws make sense only in the context of object-oriented programming like, for instance, the one for introducing innocuous assumptions about objects whose attributes are already inspected in a program (law Law 47 (innocuous assumption-reading)). We have also presented new laws for dealing with the expression `new` and the `extends` clause. These laws allowed us to derive refactorings that change an inheritance hierarchy by removing a class from it, and also extracting a superclass from two classes. We have also presented a (new) simulation law that is applicable to a class hierarchy instead of a single class (law Law 109 (superclass attribute-coupling invariant)). This law is a result of a joint work [17, 17].

The presentation of the refactoring rules in ROOL, a subset of sequential Java, does not imply lack of expressiveness of these rules. We dealt with rules that change the structure of classes in the sense that we can, for instance, extract methods, move methods between classes, parameterise methods, change the name of methods, and move attributes between classes in a hierarchy. For these, ROOL is sufficiently expressive to make our results meaningful in practice.

A limitation of our work is that we restricted the set of refactoring rules to those that deal mainly with the structure of classes, and with attributes and methods. We have not presented rules that deal with data sharing because ROOL has a copy semantics. For this reason, it is not possible to describe refactorings like Duplicated Observed Data [42, p. 189]. Refactoring rules whose derivations involve data refinement may also be invalid in the presence of pointers, as they involve specification statements in which only the values of the variables in the frame can be changed. In fact, all refactoring rules must be reviewed when using a language with pointers.

Another limitation is that, from an arbitrary command, we cannot introduce a parameterised
command with an object as argument, if such object is a method call target. The reason is the application of laws Law \( \text{Law } 62 \langle \text{var block-val} \rangle \) and Law \( \text{Law } 63 \langle \text{var block-res} \rangle \), depending on the kind of parameter passing mechanism to be used. These laws are the first step for the introduction of parameterised command. Both laws requires the variables to be passed as arguments not to be method call targets. This has limited the types of parameters of parameterised commands we introduce in derivations to be basic because there is no sense in passing an objects as an argument on which no method can be called. Consequently, refactoring rules like, for instance, \( \langle \text{Extract/Inline Method} \rangle \) involves only arguments of basic type. For instance, for introducing a parameterised command with a value argument, we apply law \( \langle \text{var block-val} \rangle \text{var block-val} \) leading to the declaration of a local variable and an assignment of the variable to be passed as argument to the local variable. Also, variable to be passed as argument is substituted with the (new) local variable. In a parameterised command, if the variable we pass as a value argument is a method call target, any change to this argument, due to the method call, is not reflected in the variable is pass as argument. This is a consequence of the copy semantics of ROOL. In a reference semantics, the restriction on the basic types of arguments should be discarded.

Also, we have not formalised refactorings that involve class methods and class variables. These features are not available in ROOL, but they are in Java. As a consequence, it is not possible to describe, for instance, refactorings that deal with the replacement of type codes with subclasses, since type codes are described using class variables.

We presented refactoring rules in which we deal with just one attribute (rule \( \langle \text{Move Attribute} \rangle \), Rule 4.3, for instance) and one method (for instance, rule \( \langle \text{Move Method} \rangle \), Rule 4.2). O This simplified derivation of refactoring rules because we do not have tool support yet. Certainly, dealing with more than one attribute and one method is is more appropriate in a context in which a tool gives support to refactorings, as it could deal more easily with an arbitrary number of attributes and methods.

The proof of the simulation laws presented in Section 3.5.1 is beyond the scope of this work. Simulation has already been addressed in [23], with respect to its soundness. This is still an ongoing research subject.

7.1 Related Work

Most of the refactoring rules that we considered are based on those presented by Fowler [42]. However, the approach adopted by Fowler is informal, based on program compilation and tests. In our approach, refactoring rules are presented with a meta-program on the left-hand side and another one on the right-hand side, along with side-conditions for the application of the rule. Besides that, we present the proof of correctness of the refactorings by the program derivation. The rules we derived slightly differ from those of Fowler. The differences involve, for instance, the number of attributes and classes involved in a refactoring.
Fowler’s work is highly dependent on compilation. This is evidenced, for instance, in the refactoring `RENAME METHOD`, in which interfaces are not mentioned. Even though `ROOL` does not provide interfaces as in Java, the description of the refactoring rule for method renaming (Rule 5.6) allowed perceiving the need for mentioning interfaces in the Fowler’s refactoring. In refactoring `EXTRACT SUPERCLASS` [42, p. 336], Fowler only addresses changes of clients of subclasses that could be changed to use the superclass. However, there is no condition concerning type tests involving variables whose types are changed to the superclass. Again, the verification of this situation is left to compilers.

Some refactorings proposed by Opdyke are “low-level refactorings” [69, Chapter 5] and are, in fact, programming laws of `ROOL`. This is the case of the refactoring that deal with the introduction of attributes, for instance. Other refactorings by Opdyke are described here as refactoring rules. For instance, the refactoring `abstract_access_to_member_variable` corresponds to the refactoring rule ⟨`Encapsulate Field`⟩ (Rule 5.8). Opdyke uses “low-level refactorings” refactorings to create an abstract superclass, for instance. We take a similar approach: the programming laws of `ROOL` serve as the basis for the derivation of the refactoring rules we presented. We have not derived any refactoring rule using the semantics of `ROOL` directly.

Opdyke proposes a set of preconditions that need to be satisfied in order for the program transformation to be behaviour preserving. In our approach, each refactoring rule has side-conditions for the application of a refactoring. If the conditions are not satisfied, the refactoring rule cannot be applied. However, Opdyke does not describe formally the effect of the application of a refactoring. This is done by Roberts [74] who extended the work of Opdyke with postconditions. In both works, preconditions and postconditions are written as predicates. None of the works present refactorings as algebraic rules. We describe refactorings between program fragments along with the conditions that must be satisfied for the application of a refactoring rule to be valid.

One of the properties that Opdyke identified as being easily violated in if explicit checks were not made before a program was refactored is `Semantically Equivalence and Operations`, which is defined as follows: "(...) let the external interface to the program be via de function `main`. If the function `main` is calle twice (once before and once after a refactoring) with the same set of inputs, the resulting set of output values must be the same" [69]. Refactoring must satisfy this condition. For instance, refactoring `change_member_function_name` [69, p. 61] requires that if we rename a method to the same name of a method that already exists in a superclass of the class that declares the method we want to rename, then these methods must be semantically equivalent. Of course, they must have the same signatures. To show that methods are semantically equivalent, it is necessary to show that they pass the same tests. However, there is no formal basis to ensure semantic equivalence between methods, there is no way to derive one method from the other, for instance, with a basis on laws of programming.

We impose explicit conditions on type tests and casts in some refactoring rules (for instance, Rule 5.3). However, conditions on type tests and casts are not explicit in Opdyke’s work. These
7.1. RELATED WORK

conditions are implicit in the property *Semantically Equivalent References and Operations*, guaranteeing that the “type of a variable can be changed by a refactoring, as long as each operation referenced on the variable is defined equivalently for its new type” [69, p.41].

Roberts [74] argues that a suite of tests can be used as a specification and the correctness of a refactoring can be proved against this specification. However, as is well-known, a suite of tests is able only to uncover errors, not to prove their absence. In order to deal with the correctness of refactorings, in our approach the derivation of a refactoring rule is carried out by the use of laws that are proved against the semantics of ROOL.

Mens, Demeyer, and Janssens [61] present a technique for the formalisation of refactorings: graph rewriting. As they use a lightweight approach, they look at notions of behaviour preservation that can be detected statically. This technique is language independent, but they recognise that some refactorings require techniques to deal with complex graphs. Refactorings such as *move method* and *push down method* are difficult to manipulate by means of graph rewriting. Although we use a specific language for describing and proving refactorings, the features of ROOL are those commonly encountered in object-oriented programming languages used in practice. Also, we do believe that presenting the rules as equality and refinement of meta-programs is a more appealing approach for practical use and automation.

Snelting and Tip [77] presented a method for finding design problems in a class hierarchy by the analysis of the *usage* of the hierarchy by a set of applications. The method is based on concept analysis. For instance, it is possible to find attributes that are redundant or that can be moved to a subclass, or it is possible to detect situations in which it is appropriate to split a class. Their work can be viewed as complementary to ours with respect to the detection and suggestion of refactorings.

The Eclipse IDE [37] does not take casts into account in its refactorings. This is a design decision for the refactoring capabilities of Eclipse [84]. In refactorings in which there are accesses to attributes via cast expressions, a warning is given to the user about these accesses being not possible after refactoring. This occurs, for instance, in the refactoring for moving a method from a class to its superclass when the method of the subclass accesses an attribute declared in this class via an expression with a cast. However, if the method is manually moved, no error is given during compilation and the program behaves as before moving the method up to the superclass. The design decision restrictions the program that can be refactored using Eclipse not to use casts.

The Eclipse IDE [37] allowed us to move to a superclass methods with the same name but with different bodies. This has not preserved the program behaviour because just one of the bodies is moved to the superclasses. In our approach, moving methods that have the same name (and parameters) from classes to the common superclass of the classes in which these methods are declared involves the application of laws Law91 (move original method to superclass) and Law89 (move redefined method to superclass). The last one introduces a conditional in the method of the superclass. The dynamic type of the object that is target of the method call deter-
mines the code which code must be executed. But, this is not the case of the refactoring in Eclipse, leading to a program behavior different from the one before refactoring. In the case of moving a method from a superclass to subclasses, excluding one of the subclasses whose instances are target of calls to such method, no warning is given to the user about the introduction of errors in the program with this change. Only at compilation time, an error is reported.

7.2 Future Work

In this work, we used a simulation law that is applicable to a class hierarchy instead of a single class (law Law\textsuperscript{109}(superclass attribute-coupling invariant)). However, the proof of this law is not available yet and is beyond the scope of the present work. Changing data representation in a single class can be viewed as the traditional module data refinement. Differently from the law Law\textsuperscript{108}(private attribute-coupling invariant), whose soundness has already been proved\textsuperscript{25}, dealing with data refinement in a class hierarchy is a new issue whose soundness must be investigated, ensuring that simulation of a class hierarchy entails refinement of such a hierarchy. This is one future direction of work.

Data refinement between attributes of different classes usually involves attributes declared by classes, not inherited ones. For instance, for the elimination of a client relationship between a class $A$ and a class $B$, we eliminate every call to methods of $B$ that appears inside $A$. We restricted methods of $B$ just to access attributes declared in $B$ itself. We have not dealt with possibly inherited attributes that could be accessed inside methods of $B$ that are called by $A$. It is necessary to investigate data refinement laws that deal with inherited attributes in situations as the one just described.

The formalisation of refactorings involves the application of several laws for commands, classes, and simulation. The activities of elaboration and presentation of these laws, the presentation of their proofs and their application are very tedious and error-prone. This highlights the need of using a theorem prover to check proofs both of laws and of refactorings. In this way, we can improve the reliability of the refactorings. Previous experience in the automation of a normal form strategy for ROOL, presented in\textsuperscript{57}, may serve as guide for the right direction to be followed for automatic proofs of refactorings.

Dealing with pointers in ROOL is also a topic of future research. This new feature will allow us to develop new refactoring rules like Duplicated Observed Data\textsuperscript{42} p. 189. It will be possible to derive patterns like Observer\textsuperscript{13} p. 293. Laws of commands of ROOL should be reviewed to guarantee their validity in a reference semantics. Also, laws that involve specification statements, like those of data refinement, should be reviewed, since frames in specifications restrict the variables that can be modified. It is necessary to investigate how to deal with frames of specification statement in a reference semantics. On the other hand, the laws for object-oriented features do not rely on copy semantics, except one for change of data representation.
7.2. FUTURE WORK

The development of a tool to support program transformation based on the refactoring rules we presented here is another direction for future research. It is necessary, for instance, to investigate techniques that can be used to determine if the conditions of a refactoring rule are satisfied, allowing its application.
Appendix A

Derivation of Compositional Refactoring Rules

In this appendix, we present derivations of refactorings rules presented in Chapter 4.

A.1 Delegation Elimination

Derivation. From \langle inline delegate \rangle (Figure A.1) and \langle (undo)inline delegate \rangle (Figure A.2).

A.1.1 inline delegate

Derivation

class A extends C

\begin{align*}
& \text{pri } b : B; \text{ ads}_{a} \\
\text{meth } m \triangleq (pds \cdot \text{self}.b.\alpha(pds)) \\
\text{mts}_{a} \\
& \text{new } \triangleq \text{self}.b := \text{new } B
\end{align*}

end

= Law 85 \text{(change visibility: from private to public) (1/r)}, Law 95 \text{(method call elimination) (1/r)}

class A extends C

\begin{align*}
& \text{pri } b : B; \text{ ads}_{a} \\
\text{meth } m \triangleq (pds \cdot \{\text{self}.b \neq \text{null} \land \text{self}.b \neq \text{error}\} (pds \cdot c) | \text{self}.b / \text{self}| (\alpha(pds))) \\
\text{mts}_{a} \\
& \text{new } \triangleq \text{self}.b := \text{new } B
\end{align*}

definition
class $A$ extends $C$
      pri $b : B$; $ads_a$
      meth $m \triangleq (pds \cdot b \cdot n(\alpha(pds)))$
      $mts_a$
end

class $B$
      pri $x : T$
      meth $n \triangleq (pds \cdot c)$
      $mts_b$
end

provided

**super** does not appear in $n$;

$b \neq \text{null} \land b \neq \text{error}$ is an invariant of $A$;

**self.a** does not appear in $n$, for any public or protected attribute $a$ that is declared
by $D$ or by any of its superclasses;

**self.p** does not appear in $n$, for any method $p$ declared in $mts_b$ or in any super-
classes of $B$;

**self.b** does not appear in $mts_a$;

$x$ is not declared in $ads_a$ nor in any superclass or subclass of $A$;

$\sqsubseteq_{cds,c}$

Figure A.1: Law \textit{(inline delegate)}

\begin{align*}
\text{class } A \text{ extends } C \\
\text{pri } b : B; \text{ } ads_a \\
\text{meth } m \triangleq (pds \cdot \{ \text{self.b} \neq \text{null} \land \text{self.b} \neq \text{error} \} \ c[\text{self.b}/\text{self}]) \\
\text{mts}_a \\
\text{new } \triangleq \text{self.b} := \text{new } B \\
\text{end}
\end{align*}

$\sqsubseteq \text{Law} [108] \text{(private attribute-coupling invariant)}$

$CI \triangleq \text{self.x} = \text{self.b.x}$

\begin{align*}
\text{class } A \text{ extends } C \\
\text{pri } b : B; \text{ } ads_a \\
\text{pri } x : T; \\
\text{meth } m \triangleq (pds \cdot \{ \text{self.b} \neq \text{null} \land \text{self.b} \neq \text{error} \} \ CI(c[b/\text{self}]))) \\
CI(mts_a) \\
\text{new } \triangleq \text{self.b} := \text{new } B \\
\text{end}
\end{align*}
A.1. DELEGATION ELIMINATION

Law [77] (remove assumption), Law [16] (\(\text{–skip unit}\))

\[
\begin{align*}
\text{class } A & \text{ extends } C \\
\text{pri } x : T; & \\
\text{pri } b : B; \text{ ads}_a & \\
\text{meth } m \triangleq (\text{pds} \bullet c[\text{self}.b/\text{self}]) & \\
\text{mts}_a & \\
\text{new } \triangleq \text{self}.b := \text{new } B & \\
\end{align*}
\]

Law [83] (attribute elimination) (r/l)

\[
\begin{align*}
\text{class } A & \text{ extends } C \\
\text{pri } x : T; \text{ ads}_a & \\
\text{meth } m \triangleq (\text{pds} \bullet CI(c[\text{self}.b/\text{self}])) & \\
CI(\text{mts}_a) & \\
\end{align*}
\]

Finally, we apply Law [85] (change visibility: from private to public), from right to left, to change the visibility of attribute \(x\) of class \(B\) to private.

A.1.2 (undo) inline delegate

\[
\begin{align*}
\text{class } A & \text{ extends } C & \text{class } B \\
\text{pri } x : T; \text{ ads}_a & & \text{pri } x : T \\
\text{meth } m \triangleq (\text{pds} \bullet c) & & \text{meth } n \triangleq (\text{pds} \bullet c) \\
\text{mts}_a & & \text{mts}_b \\
\end{align*}
\]

[Definition: \((\text{pds} \bullet c) = (\text{pds} \bullet (\text{pds} \bullet c)(\alpha(\text{pds})))]

\[
\begin{align*}
\text{class } A & \text{ extends } C \\
\text{pri } x : T; \text{ ads}_a & \\
\text{meth } m \triangleq (\text{pds} \bullet (\text{pds} \bullet c)(\alpha(\text{pds}))) & \\
\text{mts}_a & \\
\end{align*}
\]

Law [85] (change visibility: from private to public) (l/r), Law [108] (change visibility: from private to public)
provided

\textbf{super} does not appear in \(n\);
\(b \neq \text{null} \land b \neq \text{error}\) is an invariant of \(A\);
\(b\) is not declared in \(ads_a\) nor in any superclass or subclass of \(A\);
\(\text{self}.x\) does not appear in \(mts_a\);

\begin{figure}[h]
\centering
\begin{align*}
\text{class } A \text{ extends } C \\
\text{pri } x : T; \text{ ads}_a \\
\text{meth } m \triangleq (pds \bullet c) \\
mts_a \\
\end{align*}
\end{figure}

\begin{figure}[h]
\centering
\begin{align*}
\text{class } B \\
\text{pri } x : T \\
\text{meth } n \triangleq (pds \bullet c) \\
mts_b \\
\end{align*}
\end{figure}

\begin{figure}[h]
\centering
\begin{align*}
\text{class } A \text{ extends } C \\
\text{pri } b : B; \text{ ads}_a \\
\text{meth } m \triangleq (pds \bullet \text{self}.b.n(\alpha(pds))) \\
mts_a \\
\text{new } \triangleq \text{self}.b := \text{new } B \\
mts_b \\
\end{align*}
\end{figure}

\begin{figure}[h]
\centering
\begin{align*}
\text{class } B \\
\text{pri } x : T \\
\text{meth } n \triangleq (pds \bullet c) \\
mts_b \\
\end{align*}
\end{figure}

\[ [\text{Assignment, specifications, and guards diminution}] \]

\begin{figure}[h]
\centering
\begin{align*}
\text{class } A \text{ extends } C \\
\text{pri } b : B; \text{ ads}_a \\
\text{pri } x : T; \text{ ads}_a \\
\text{meth } m \triangleq (pds \bullet (pds \bullet CI(c)))(\alpha(pds))) \\
\text{new } \triangleq \text{self}.b := \text{new } B \\
mts_a \\
\end{align*}
\end{figure}

\begin{figure}[h]
\centering
\begin{align*}
\text{class } A \text{ extends } C \\
\text{pri } b : B; \\
\text{pri } x : T; \text{ ads}_a \\
\text{meth } m \triangleq (\text{pds} \bullet (\text{pds} \bullet c)][\text{self}.b/\text{self}](\alpha(pds))) \\
\text{new } \triangleq \text{self}.b := \text{new } B \\
mts_a \\
\end{align*}
\end{figure}
A.2 Inline Class

For the derivation of the refactoring rule ⟨Inline Class⟩ we copy attributes of \( B \) to \( A \) all clients of that class and eliminate calls to methods of \( B \) that appear in \( A \).
We begin the derivation with the left-hand side of the rule.

\[
\text{class } A \text{ extends } C \\
\text{pri } b : B; \text{ ads}_{a} \\
\text{meth } \text{getX} \equiv (\text{res } \text{arg} : T \cdot \text{self}.b \text{.getX}(\text{arg})) \\
\text{meth } \text{setX} \equiv (\text{val } \text{arg} : T \cdot \text{self}.b \text{.setX}(\text{arg})) \\
\text{meth } m \equiv (\text{pds} \cdot \text{cm}[\text{self}.b \text{.m}(\alpha(\text{pds}))]) \\
\text{new } \equiv b := \text{new } B(); \\
\text{mts}_{a}
\]

\[
\text{class } B \\
\text{pri } x : T; \\
\text{meth } \text{getX} \equiv (\text{res } \text{arg}' : T \cdot \text{arg}' := \text{self}.x) \\
\text{meth } \text{setX} \equiv (\text{val } \text{arg}' : T \cdot \text{self}.x := \text{arg}') \\
\text{meth } m \equiv (\text{pds} \cdot \\
\text{ cm}[\text{var } \text{aux} : T \cdot \text{self}.\text{getX}(\text{aux}); \\
\text{ le } := \text{exp}_{1}[\text{aux}]\text{end,} \\
\text{ self}.\text{setX}(\text{exp}_{2}, \alpha(\text{pds}))])
\]

\[
\text{mts}_{a}
\]

We eliminate the clientship relation between \(A\) and \(B\) The attribute \(x\) of \(B\) is copied to \(A\) and all methods of \(B\) that are called from \(A\) are also copied to \(A\). Calls to methods of \(B\) are replaced with calls to the methods we copied from \(B\) to \(A\). The attribute \(b\) initially present in \(A\) is removed from this class.

\[= \langle \text{Clientship Elimination} \rangle \ (1/r)\]

\[
\text{class } A \text{ extends } C \\
\text{pri } x : T; \text{ ads}_{a} \\
\text{meth } \text{getX} \equiv (\text{res } \text{arg} : T \cdot \text{self}.b \text{.getX}(\text{arg})) \\
\text{meth } \text{setX} \equiv (\text{val } \text{arg} : T \cdot \text{self}.b \text{.setX}(\text{arg})) \\
\text{meth } \text{bgetX} \equiv (\text{res } \text{arg}' : T \cdot \text{arg}' := \text{self}.x) \\
\text{meth } \text{bsetX} \equiv (\text{val } \text{arg}' : T \cdot \text{self}.x := \text{arg}') \\
\text{meth } m \equiv (\text{pds} \cdot \text{cm}[\text{self}.p(\alpha(\text{pds}))]) \\
\text{meth } p \equiv (\text{pds} \cdot \\
\text{ cm}[\text{var } \text{aux} : T \cdot \text{self}.b \text{.getX}(\text{aux}); \\
\text{ le } := \text{exp}_{1}[\text{aux}]\text{end,} \\
\text{ self}.b \text{.setX}(\text{exp}_{2}, \alpha(\text{pds}))])
\]

\[
\text{mts}_{a}
\]

We eliminate calls inside \(A\) that have \texttt{self} as target. We do this by applying the laws method call and parameterised command elimination.
A.2. INLINE CLASS

\[ A.2. \text{INLINE CLASS} \]

= Lemma \[ \text{Lemma 1} \langle \text{pcom elimination-val} \rangle \langle \text{l/r} \rangle (2x), \text{Law 64} \langle \text{var block-val} \rangle \langle \text{r/l} \rangle (2x), \text{Law 65} \langle \text{pcom elimination-res} \rangle \langle \text{l/r} \rangle (2x), \text{Law 63} \langle \text{var block-res} \rangle \langle \text{r/l} \rangle (2x), \]

\[ \text{class } A \text{ extends } C \]

\[ \text{pri } x : T; \text{ adsa;} \]
\[ \text{meth } \text{getX } \equiv (\text{res } \text{arg } : T \bullet \text{arg } := \text{self}.x) \]
\[ \text{meth } \text{setX } \equiv (\text{val } \text{arg } : T \bullet \text{self}.x := \text{arg}) \]
\[ \text{meth } \text{bGetX } \equiv (\text{res } \text{arg}' : T \bullet \text{arg}' := \text{self}.x) \]
\[ \text{meth } \text{bsetX } \equiv (\text{val } \text{arg}' : T \bullet \text{self}.x := \text{arg}') \]
\[ \text{meth } m \equiv (\text{pds } \bullet \text{cm}[\text{self}.p(\alpha(\text{pds}))]) \]
\[ \text{meth } p \equiv (\text{pds } \bullet \text{cp}[\text{var aux } : T \bullet \text{aux } := \text{self}.x; \text{le } := \text{exp1}[\text{aux}] \text{end, self}.x := \text{exp2}, \alpha(\text{pds})]) \]

\[ \text{mts}_a \quad \text{end} \]

In the command \( \text{cp} \) we can eliminate the variable \( \text{aux} \). We replace the occurrence of \( \text{aux} \) in the expression \( \text{exp1} \) with the expression \( \text{self}.x \) that is assigned to \( \text{aux} \). After this, we advance the assignment to \( \text{aux} \) with respect to the assignment to \( \text{le} \). We can do it because \( \text{aux} \) does not appear in the assignment to \( \text{le} \).

\[ = \text{Law 60} \langle \text{assignment seq comp exp substitution} \rangle \langle \text{r/l} \rangle, \text{Law 59} \langle \text{order independent assignment} \rangle \]

\[ \text{cp}[\text{var aux } : T \bullet \text{le } := \text{exp1}[\text{self}.x]; \text{aux } := \text{self}.x \text{ end, self}.x := \text{exp2}, \alpha(\text{pds})] \]

At this point, we can eliminate the assignment to \( \text{aux} \) because it is at the end of the local variable block. We do this by applying law \( \text{Law 30} \langle \text{var } := \text{final value} \rangle \). The only command present in the block that declares \( \text{aux} \) is the assignment to \( \text{le} \), in which there are no occurrences of \( \text{aux} \). We can remove this variable by applying law \( \text{Law 25} \langle \text{var elim} \rangle \).

\[ = \text{Law 30} \langle \text{var } := \text{final value} \rangle \langle \text{l/r} \rangle, \text{Law 25} \langle \text{var elim} \rangle \langle \text{l/r} \rangle \]

\[ \text{cp}[\text{le } := \text{exp1}[\text{self}.x], \text{self}.x := \text{exp2}, \alpha(\text{pds})] \]

As the methods \( \text{bGetX} \) and \( \text{bsetX} \) were introduced in a previous step, they are not called inside \( A \), we can remove them.
APPENDIX A. DERIVATION OF COMPOSITIONAL REFACTORING RULES

= Law $^{90}$ (method elimination) (l/r)/(2x)

class $A$ extends $C$
    pri $x : T$; $ads_a$;
    meth $m_1 \equiv (pds_1 \bullet c_1[le_1 := exp_1[x], self.x := exp_2])$
    new $\equiv (val$ arg $: T \bullet self.x := arg)$
mts_a
end

This finishes the derivation of (Inline Class).

A.3 Self Encapsulate Field

We begin the derivation with the class $A$ that appears on the left-hand side of Rule $^{110}$

class $A$
    pri $x : T$; $ads_a$
    meth $m_1 \equiv (pds_1 \bullet c_1[le_1 := exp_1[x], self.x := exp_2])$
    new $\equiv (val$ arg $: T \bullet self.x := arg)$
mts_a
end

First, we introduce setting and getting methods for the variable $x$ that we want to be self
encapsulated.

= Law $^{90}$ (method elimination) (r/l) (2x)

class $A$
    pri $x : T$; $ads_a$
    meth $m_1 \equiv (pds_1 \bullet c_1[le_1 := exp_1[x], self.x := exp_2])$
    meth $getX \equiv (res$ arg $: T \bullet arg := self.x)$
    meth $setX \equiv (val$ arg $: T \bullet self.x := arg)$
    new $\equiv (val$ arg $: T \bullet self.x := arg)$
mts_a
end

We concentrate on commands of the form $le := exp_1[x]$ in which the variable $x$ occurs in the
expression $exp_1$ and is directly read. The first step to eliminate such direct access is to declare a
variable to capture the value to of the attribute $x$. According to a condition to the application of
this rule, the variable $aux$ is new.
A.3. SELF ENCAPSULATE FIELD

= Law 25 (var- elim) (r/l)

\[ \text{var aux : } T \bullet \{ \text{self} \neq \text{null} \land \text{self} \neq \text{error} \} \, le_1 := \exp[x] \text{end} \]

We assign the variable \( x \) to \( aux \) at the end of the scope of the last variable.

= Law 30 (var- := final value)

\[ \text{var aux : } T \bullet \{ \text{self} \neq \text{null} \land \text{self} \neq \text{error} \} \, le_1 := \exp_1[x] ; \, aux := x \text{ end} \]

The variable \( aux \) is new, allowing us to move the assignment \( aux := x \) to be placed before the assignment to \( le \).

= Law 59 (order independent assignment)

\[ \text{var aux : } T \bullet \{ \text{self} \neq \text{null} \land \text{self} \neq \text{error} \} \, aux := x ; \, le_1 := \exp_1[x] \text{ end} \]

The variable \( x \) that is assigned to \( aux \) also occurs in the expression \( \exp_1 \). We substitute this occurrence of \( x \) with the variable \( aux \).

= Law 60 (assignment seq comp exp substitution)

\[ \text{var aux : } T \bullet \, aux := x ; \, le_1 := \exp[aux] \text{end} \]

As we want to indirectly access the variable \( x \), we need to introduce a call to \( \text{getX} \). We declare a variable \( p_{aux} \) to which is assigned the variable \( x \) and replace the \( aux \) with \( p_{aux} \) in the command \( aux := x \).

= Lemma 10 (1/r)

\[ \text{var aux : } T \bullet \{ \text{self} \neq \text{null} \land \text{self} \neq \text{error} \} \, (\text{res arg : } T \bullet \, \text{arg} := b \cdot x)(aux) \quad \triangledown \]

\[ \, le_1 := \exp[aux] \text{end} \]

The parameterised command obtained in the previous step is equal to the one that defines the method \( \text{getX} \). So, we introduce a call to such method, obtaining the following program.
For commands of the form \( \text{self}.x := \text{exp}_2 \) we proceed as follows. We declare the variable \( p_{\text{arg}} \) to which we assign the expression \( \text{exp}_2 \). That variable substitutes the expression \( \text{exp}_2 \) in the assignment to \( \text{self}.x \).

\[
\{ \text{self} \neq \text{null} \land \text{self} \neq \text{error} \} \quad \text{self}.x := \text{exp}_2
\]

This parameterised command, not taking into account the application to the argument \( \text{exp}_2 \), is the same as that of method \( \text{setX} \). So, we introduce a call to this method.

\[
\text{self}.\text{setX}(\text{exp}_2)
\]

The same steps for introducing a call to the method \( \text{setX} \) above are followed for changing the initialiser of class \( A \) and introducing the same method call. The final class \( A \) is as follows.

\[
\text{class } A
\]
\[
\begin{align*}
\text{pri } x &: T; \text{ads}_a \\
\text{meth } m_1 &\triangleq (pds_1 \bullet c[\text{var } aux : T \bullet \text{self}.\text{getX}(aux); \\
&le_1 := \text{exp}_1[aux]\text{end, self}.\text{setX}(\text{exp}_2)]) \\
\text{meth } \text{getX} &\triangleq (\text{res } arg : T \bullet arg := \text{self}.x) \\
\text{meth } \text{setX} &\triangleq (\text{val } arg : T \bullet \text{self}.x := arg) \\
\text{new} &\triangleq (\text{val } arg : T \bullet \text{self}.\text{setX}(arg)) \quad \text{mts}_a
\end{align*}
\]

### A.4 Decompose Conditionals

For the derivation of this refactoring, it is necessary to parameters with a specific passing mechanism, not the abstract list of parameters \( pds \). Also, we consider variable \( a \) and \( b \), \( c \) and \( d \) to appear in commands \( c_{m_1} \) and \( c'_{m_1} \). We suppose that variables \( a \) and \( c \) hold information for the branches of
the alternation; variables $b$ and $c$ are used to get values that results from the branches. We could think about these variables as parameters value parameters ($a$ and $c$) and result parameters ($b$ and $d$) of method $m_1$. For simplicity, we assume these variables have type $T$. We begin the derivation with the class $A$ of the left-hand side of the rule.

\begin{verbatim}
class A
  ads
  meth $m_1 \triangleq (pds_1 \bullet$
    if $exp \rightarrow c_{m_1}[a, b]$
    [ ] $\neg exp \rightarrow c'_{m_1}[c, d]$
  fi)
  mts
end
\end{verbatim}

First, we introduce the methods $m_2$, $m_3$, and $m_4$. The body of method $m_2$ is a parameterised command that has a result argument that returns to the caller the boolean value of the expression $exp$. We define the methods $m_3$ and $m_4$ with parameterised commands that have parameters passed by value-result. The commands that appear in these methods are those in the branches of the alternation.

\begin{verbatim}
= Law 90 (method elimination) (r/l)(3x)
class A
  ads
  meth $m_1 \triangleq (pds_1 \bullet$
    if $exp \rightarrow c_{m_1}[a, b]$
    [ ] $\neg exp \rightarrow c'_{m_1}[c, d]$
  fi)
  meth $m_2 \triangleq (\text{res arg : bool} \bullet \text{arg := exp})$
  meth $m_3 \triangleq (\text{val arg}_1 : T; \text{res arg}_2 : T \bullet c_{m_1}[\text{arg}_1, \text{arg}_2])$
  meth $m_4 \triangleq (\text{val arg}_3 : T; \text{res arg}_4 : T \bullet c'_{m_1}[\text{arg}_3, \text{arg}_4])$
  mts
end
\end{verbatim}

We declare $a$ the boolean variable $b$ in method $m_1$. 


At the end of the scope of the variable \( b \), we assign the expression \( \text{exp} \) to \( b \).

Since \( b \) is new—it does not occur in the alternation—, we can move the assignment to it to be placed in the beginning of the variable block.

The expression \( \text{exp} \) occurs in the guards of the alternation. We replace it with the variable \( b \) to which is assigned the boolean value of \( \text{exp} \).

= Law 59 (order independent assignment)

= Law 60 (assignment seq comp exp substitution)
The command \( b := \text{exp} \) is similar to that present in the method \( m_2 \). We introduce a call to such method by introducing a variable block in which we replace the variable \( b \) with the new variable \( p\_\text{arg} \). At the end of the block of variable \( p\_\text{arg} \), we assign it to \( b \).

\[
\text{Law 63 } \langle \text{var block-res} \rangle \ (1/r)
\]

\begin{verbatim}
var p\_arg : bool •
p\_arg := \text{exp};
b := p\_arg
end
\end{verbatim}

We introduce a parameterised command that corresponds to the variable block we declared in the previous step.

\[
\text{Law 65 } \langle \text{pcom elimination-res} \rangle \ (1/r)
\]

\begin{verbatim}
(res \ arg : bool • arg := \text{exp})(b)
\end{verbatim}

We obtained exactly the parameterised command in the body of method \( m_2 \), applied to the argument \( b \).

\[
\text{Lemma \[2\] (r/l)}
\]

\[
\text{self.m}_2(b)
\]

The class \( A \) now is as follows.

\begin{verbatim}
class A
  ads
  meth \ m_1 \cong (\text{pds}_1 •
    var \ b : bool • self.m_2(b);
    \text{if } b \rightarrow c_{m_1}[a, b] \triangleleft
    \text{[} \neg b \rightarrow c'_{m_1}[c, d]\text{] fi}
  end
  meth \ m_2 \cong (\text{res \ arg : bool • arg := \text{exp}})
  meth \ m_3 \cong (\text{vres \ arg}_3 : T_3 • c[\text{arg}_3])
  meth \ m_4 \cong (\text{vres \ arg}_4 : T_4 • c'[\text{arg}_4])
  mts
end
\end{verbatim}

We proceed by introducing calls to the methods \( m_3 \) and \( m_4 \) from the guarded commands \( c_{m_1} \).
and \( c'_{m_1} \). The steps we follow are similar to those we have used to introduce the call to the method \( m_2 \). First we introduce variable blocks with assignments according to the parameter passing mechanism we intend to use; here the mechanisms are value and result. Then we introduce parameterised commands that correspond to the variable blocks just introduced.

The same steps are followed for introducing a call to the method \( m_4 \) from the command \( c'_{m_1} \). The final class \( A \) is the following.

```plaintext
class A
  ads
  meth \( m_1 \) = (pds_1 •
      var \( b \) : bool • self.m_2(b);
      if \( b \rightarrow \) self.m_3(a_i)
        \( \neg b \rightarrow \) self.m_4(a_j)
      fi
  end
  meth \( m_2 \) = (res \( arg \) : bool • \( arg := exp \))
  meth \( m_3 \) = (vres \( arg_3 \) : \( T_3 \) • \( c'[arg_3] \))
  meth \( m_4 \) = (vres \( arg_4 \) : \( T_4 \) • \( c'[arg_4] \))
  mts_a
end
```

This finishes the derivation of the refactoring rule \( \langle \text{Decompose Conditional} \rangle \), which is presented only by Fowler \[42\].

This refactoring improves legibility, making a method clearer by clarifying reasons for branches of the alternation. Names of methods being extracted have a crucial role in improving legibility. We can see this refactoring as the application of \( \langle \text{Extrac/Inline Method} \rangle \), from left to right, for guards and for each branch of the alternation.

### A.5 Introduce Explaining Variable

We begin the derivation of Rule \([4.12]\) with the command \( c \) in which there is an occurrence of the expression \( e \).

\[ c[e] \]

First we introduce a variable that is used to hold the values of the expression we want to make clear its purpose.

\[ = \text{Law} [25] (\var elim) (r/l) \]

\[ \var x : T \ • \ c[e] \ \text{end} \]
At the end of the variable block, we assign to \( x \) the expression \( e \).

\[
\text{Law 30 (var- := final value)}
\]

\[
\text{var } x : T \cdot c[e]; \ x := e \ \text{end}
\]

As the variable \( x \) is new, we can move the assignment of \( e \) to \( x \) just before the command \( c[e] \).

\[
\text{Law 59 (order independent assignment)}
\]

\[
\text{var } x : T \cdot x := e; \ c[e] \ \text{end}
\]

We replace the occurrences of \( e \) with \( x \) in the command \( c[e] \) based on the fact that \( x \) is new, thus not free in \( c \).

\[
\text{Law 60 (assignment seq comp exp substitution)}
\]

\[
\text{var } x : T \cdot x := e; \ c[x] \ \text{end}
\]

### A.6 Consolidate Conditional Expression

The cases covered in refactoring (Consolidate Conditional Expression) are those of two basic laws of ROOL that deal with alternations.

**Cases**

- Or: Law 8 (if- \( \lor \) distrib1)
- And: Law 11 (if- \( \land \) distrib)

### A.7 Consolidate Duplicate Conditional Fragments

This refactoring is, in fact, the basic law of ROOL that deal with the distribution of a command over an alternation.

- Law 20 (if- \( \land \) left dist)

### A.8 Substitute Algorithm

This rule is a direct consequence of law Law 67 (command refinement-class refinement).
Appendix B

Derivation of Contextual Refactoring Rules

In this appendix, we present derivations of refactoring rules presented in Chapter 5.

B.1 Add/Remove Parameter

The derivation of rule \( \langle \text{Add/Remove Parameter} \rangle \) follows the strategy below.

1. Moving of all definitions of the method \( m \) to the topmost class that first introduces a definition for it;
2. Elimination of calls to the method \( m \);
3. Addition of the new parameter \( pd \) to the parameterised command that resulted from step 2;
4. Elimination of the method with the original number of parameters;
5. Introduction of method \( m \) with the desired number of parameters;
6. Introduction of calls to the new method;
7. Moving down of the definition of method \( m \) to the subclasses of the topmost class that first introduces a definition of such method.

Writing the new parameter as \( pd \) gives a programmer the opportunity to add a parameter with a value or a result parameter passing mechanism. The refactoring Add Parameter presented by Fowler [42] is a particular case of the one presented here because he considers just information that a method needs from its callers. In our refactoring, the programmer may also define information that a method may return to its callers.
An example for \((\text{Add/Remove Parameter})\)

We consider the following sequence of class declarations. The method \(m\) is defined by a parameterised command with the value parameter \(x\). Our aim is to add the value parameter \(y\).

\[
\begin{align*}
\text{class } D & \\
\& \quad \text{mhs} = (\text{val } x : T \cdot c_d[\text{exp}]) \\
\& \quad \text{mts} = [\text{self.m}(a)] \\
\text{end}
\end{align*}
\[
\begin{align*}
\text{class } B \text{ extends } D & \\
\& \quad \text{mhs} = (\text{val } x : T \cdot c_b[\text{exp}]) \\
\& \quad \text{mts} = [\text{self.m}(a)] \\
\text{end}
\end{align*}
\[
\begin{align*}
\text{class } A \text{ extends } D & \\
\& \quad \text{mhs} = (\text{val } x : T \cdot c_a[\text{exp}]) \\
\& \quad \text{mts} = [\text{self.m}(a)] \\
\text{end}
\end{align*}
\]

\[
\begin{align*}
\text{class } C \text{ extends } A & \\
\& \quad \text{mts} = [\text{self.m}(a)] \\
\text{end}
\end{align*}
\]

The first step is to move all definitions of \(m\) to the topmost class of \(A\) that first introduces a definition of \(m\). In our example, we have to move all definitions to \(D\).

\[
\begin{align*}
\text{class } D & \\
\& \quad \text{mhs} = (\text{val } x : T \cdot c_d[\text{exp}]) \\
\& \quad \text{mts} = [\text{self.m}(a)] \\
\text{end}
\end{align*}
\]

\[
\begin{align*}
\text{class } B \text{ extends } D & \\
\& \quad \text{mhs} = (\text{val } x : T \cdot c_b[\text{exp}]) \\
\& \quad \text{mts} = [\text{self.m}(a)] \\
\text{end}
\end{align*}
\]

\[
\begin{align*}
\text{class } A \text{ extends } D & \\
\& \quad \text{mhs} = (\text{val } x : T \cdot c_a[\text{exp}]) \\
\& \quad \text{mts} = [\text{self.m}(a)] \\
\text{end}
\end{align*}
\]

\[
\begin{align*}
\text{class } C \text{ extends } A & \\
\& \quad \text{mts} = [\text{self.m}(a)] \\
\text{end}
\end{align*}
\]

\[
\begin{align*}
= \text{Lemma 2}
\end{align*}
\]

\[
\begin{align*}
\text{class } D & \\
\& \quad \text{mhs} = (\text{val } x : T \cdot \text{if } \text{self is } B \rightarrow c'_b[\text{exp}] \\
\& \quad \quad \text{fi}) \\
\& \quad \text{mts} = [\{\text{self isn't null } \land \text{self isn't error}\} (\text{val } x : T \cdot c_d[\text{exp}])](a)] \\
\text{end}
\end{align*}
\]
B.1. ADD/REMOVE PARAMETER

class B extends D
  
  ads
  
  mts
  
  {self ≠ null ∧ self ≠ error}
  
  (val x : T • cb)(a)
  
  end

class A extends D
  
  ads
  
  mts
  
  {self ≠ null ∧ self ≠ error}
  
  (val x : T • ca[exp])(a)
  
  end

class C extends A
  
  ads
  
  mts
  
  {self ≠ null ∧ self ≠ error}
  
  (val x : T • cc[exp])(a)
  
  end

We eliminate every method call \text{le.m}(a) in \text{cds}, and \text{c}. Let us consider the following method call where the type of the left-expression \text{le} is given by any class in the hierarchy of \text{D}.

\text{le.m}(a)

= Law95 \text{(method call elimination)} \ (1/r)

\{le ≠ null ∧ le ≠ error\}

(val x : T • if le is B → cb[exp]

[] ¬ (le is B) → if le is A → cd[exp]

[] ¬ (le is A) → if (A)le is C → cc[exp]

[] ¬ ((A)le is C) → cc[exp]

fi

fi

fi

\fi(a)

The Lemma9 introduces a parameterised command with a value argument from a non-parameterised command. It can be found in Appendix C

= Lemma9

\{le ≠ null ∧ le ≠ error\}

(val x : T •

(val y : T •

if le is B → cb[y]

[] ¬ (le is B) → if le is A → cd[y]

[] ¬ (le is A) → if (A)le is C → cc[y]

[] ¬ ((A)le is C) → cc[y]

fi

fi

fi

\fi((exp))(a)
= Law \[66\] (pcom merge)

\[
\{(le \neq \text{null} \land le \neq \text{error}\}
\]

\[
\begin{align*}
&\text{(val } x : T; \text{ val } y : T) \bullet \\
&\text{if } le \text{ is } B \rightarrow c'_b[y] \\
&\text{[] } \neg (le \text{ is } B) \rightarrow \text{if } le \text{ is } A \rightarrow c_d[y] \\
&\quad \text{[] } \neg (le \text{ is } A) \rightarrow \text{if } (A)le \text{ is } C \rightarrow c'_c[y] \\
&\quad \text{[] } \neg ((A)le \text{ is } C) \rightarrow c'_d[y] \\
&\text{fi}
\end{align*}
\]

\[
\text{fi}(a, \text{exp})
\]

In the class \(D\) we eliminate the old method \(m\) and introduce a new method \(m\) defined by the same parameterised command we obtained in the previous step.

= Law \[90\] (method elimination) (l/r), Law \[90\] (method elimination) (r/l)

\[
\text{class } D
\]

\[
\quad \text{ads}_c
\]

\[
\quad \begin{align*}
&\text{(val } x : T; \text{ val } y : T) \bullet \\
&\text{if } le \text{ is } B \rightarrow c'_b[y] \\
&\text{[] } \neg (le \text{ is } B) \rightarrow \text{if } le \text{ is } A \rightarrow c_d[y] \\
&\quad \text{[] } \neg (le \text{ is } A) \rightarrow \text{if } (A)le \text{ is } C \rightarrow c'_c[y] \\
&\quad \text{[] } \neg ((A)le \text{ is } C) \rightarrow c'_d[y] \\
&\text{fi}
\end{align*}
\]

\[
\text{fi}(a, \text{exp})
\]

\[
\text{mts}_d[\text{\{self } \neq \text{null } \land \text{self } \neq \text{error}\}] (\text{val } x : T; \text{ val } y : T \bullet c_d[y])(a, \text{exp})
\]

end

Now we introduce calls to the new method \(m\). This step is carried out in every point of the program in which there was an original call to \(m\). In other words, we introduce method calls to each composition of an assumption about an object of type \(D\), or any of its subclasses, and the parameterised command the defines the method \(m\) in \(D\).

\[
\{(le \neq \text{null} \land le \neq \text{error}\}
\]

\[
\begin{align*}
&\text{(val } x : T; \text{ val } y : T) \bullet \\
&\text{if } le \text{ is } B \rightarrow c'_b[y] \\
&\text{[] } \neg (le \text{ is } B) \rightarrow \text{if } le \text{ is } A \rightarrow c_d[y] \\
&\quad \text{[] } \neg (le \text{ is } A) \rightarrow \text{if } (A)le \text{ is } C \rightarrow c'_c[y] \\
&\quad \text{[] } \neg ((A)le \text{ is } C) \rightarrow c'_d[y] \\
&\text{fi}
\end{align*}
\]

\[
\text{fi}(a, \text{exp})
\]
The last step is to move down the definition of \( m \) from \( D \) to its subclasses. We use Lemma 3 which moves a definition of a method down in the hierarchy of the class that first introduces a definition for it.

\[
\text{class } D\ads_c \begin{align*}
\text{meth } m & \triangleq (\text{val } x : T; \text{val } y : T \bullet c_d[y]) \\
mts_c & [\text{self}.m(a, exp)]
\end{align*}
\]  
\[
\text{class } B \text{ extends } D \ads_b \\
\text{meth } m & \triangleq (\text{val } x : T; \text{val } y : T \bullet c_b[y]) \\
mts_b & [\text{self}.m(a, exp)]
\]
\[
\text{class } A \text{ extends } D \ads_a \\
\text{meth } m & \triangleq (\text{val } x : T; \text{val } y : T \bullet c_a[y]) \\
mts_a & [\text{self}.m(a, exp)]
\]
\[
\text{cds} & [\text{le}.m(a, exp)] \bullet c[\text{le}.m(a, exp)]
\]

This finishes the example for rule \( \langle \text{Add/Remove Parameter} \rangle \).

\[ \Box \]

## B.2 Separate Query From Modifier

We begin the derivation by changing the visibility of the private attribute \( x \).

\[
\text{class } A \begin{align*}
\text{pri } x : T; & \ads_a; \\
\text{meth } & \text{getSetX } \triangleq (\text{val } \text{arg}_1 : T; \text{res } \text{arg}_2 : T \bullet \text{arg}_2 := \text{self}.x; \text{self}.x : \text{arg}_1) \\
mts_a
\end{align*}
\]

\[ = \text{Law}\ [95] \text{ (change visibility: from private to public)} \]

\[
\text{class } A \begin{align*}
\text{pub } x : T; & \ads_a; \\
\text{meth } & \text{getSetX } \triangleq (\text{val } \text{arg}_1 : T; \text{res } \text{arg}_2 : T \bullet \text{arg}_2 := \text{self}.x; \text{self}.x : \text{arg}_1) \\
mts_a
\end{align*}
\]

For all commands \( \text{le}.\text{getSetX}(a_1, a_2) \) that appear in \( \text{cds} \) and \( c \), we proceed as follows. First, we
eliminate the call to the method `getSetX` of class `A`, and separate the parameters, resulting in the following sequential composition of commands.

\[\text{le}.\text{getSetX}(a_1, a_2)\]

\[= \text{Law 95 (method call elimination)(l/r)}\]

\[\{\text{le} \neq \text{null} \land \text{le} \neq \text{error}\} \ (\text{val } \text{arg}_1 : T; \ \text{res } \text{arg}_2 : T \bullet \ \text{arg}_2 := \text{le}.x; \ \text{le}.x := \text{arg}_1)(a_1, a_2)\]

\[= \text{Law 66 (pcom merge)(l/r)}\]

\[\{\text{le} \neq \text{null} \land \text{le} \neq \text{error}\} \ (\text{val } \text{arg}_1 : T \bullet (\text{res } \text{arg}_2 : T \bullet \ \text{arg}_2 := \text{le}.x; \ \text{le}.x := \text{arg}_1)(a_2))(a_1)\]

We eliminate the parameterised commands we obtained in the last step of the derivation, resulting in the following program.

\[= \text{Law 65 (pcom elimination-res)(l/r)}, \ \text{Law 64 (pcom elimination-val)(l/r)}\]

\[\{\text{le} \neq \text{null} \land \text{le} \neq \text{error}\}\]

\[\text{var } p_{-\text{arg}_1} : T \bullet\]

\[p_{-\text{arg}_1} := a_1;\]

\[\text{var } p_{-\text{arg}_2} : T \bullet\]

\[p_{-\text{arg}_2} := \text{le}.x;\]

\[\text{le}.x := p_{-\text{arg}_1};\]

\[a_2 := p_{-\text{arg}_2}\]

end end

We move the assignment \(p_{-\text{arg}_1} := a_1\) inside the block that declares the variable \(p_{-\text{arg}_2}\), as this variable does not occur in the assignment. Our intention is to obtain two separated variable blocks, the first assigning a value to \(\text{le}.x\), and the second just reading \(\text{le}.x\) and assigning the value read to the local variable \(p_{-\text{arg}_2}\).
= Law 29 (var-; right dist)(l/r)

{\text{le} \neq \text{null} \land \text{le} \neq \text{error}}

\begin{align*}
\text{var } & p_{\text{arg1}} : T \quad \bullet \\
\text{var } & p_{\text{arg2}} : T \quad \bullet \\
& p_{\text{arg1}} := a_1; \quad < \\
& p_{\text{arg2}} := \text{le}.x; \\
& \text{le}.x := p_{\text{arg1}}; \\
& a_2 := p_{\text{arg2}} \quad < \\
\text{end } & \\
\text{end }
\end{align*}

The order in which the assignments to the variables \(p_{\text{arg1}}\) and \(p_{\text{arg2}}\) occur is irrelevant, as \(p_{\text{arg1}}\) and \(p_{\text{arg2}}\) are not free in the assigning expressions.

= Law 59 (order independent assignment) (l/r)

{\text{le} \neq \text{null} \land \text{le} \neq \text{error}}

\begin{align*}
\text{var } & p_{\text{arg1}} : T \quad \bullet \\
\text{var } & p_{\text{arg2}} : T \quad \bullet \\
& p_{\text{arg2}} := \text{le}.x; \\
& p_{\text{arg1}} := a_1; \\
& \text{le}.x := p_{\text{arg1}}; \\
& a_2 := p_{\text{arg2}} \quad < \\
\text{end } & \\
\text{end }
\end{align*}

We also move the assignment to the variable \(a_2\) to just after the assignment \(p_{\text{arg2}} := \text{le}.x\).

= Law 59 (order independent assignment) (r/l)(2x)

{\text{le} \neq \text{null} \land \text{le} \neq \text{error}}

\begin{align*}
\text{var } & p_{\text{arg1}} : T \quad \bullet \\
\text{var } & p_{\text{arg2}} : T \quad \bullet \\
& p_{\text{arg2}} := \text{le}.x; \\
& a_2 := p_{\text{arg2}}; \\
& p_{\text{arg1}} := a_1; \\
& \text{le}.x := p_{\text{arg1}} \\
\text{end } & \\
\text{end }
\end{align*}

We move the last two assignments outside the block that declares \(p_{\text{arg2}}\), obtaining a sequential composition in which the first component is the variable block, and the second component is the sequence of assignments we are moving. We can do this because the variable \(p_{\text{arg2}}\) does not occur
in the assignments.

Law \[28\] (\texttt{var}--; left dist)(r/l) \((2x)\)

\[
\{le \neq \texttt{null} \land le \neq \texttt{error}\}
\]

\begin{verbatim}
var parg1 : T •
  var p_arg2 : T •
     p_arg2 := le.x;
     a2 := p_arg2;
   end;
     p_arg1 := a1;
     le.x := p_arg1
 end
\end{verbatim}

We move the block that declares variable \(p\_arg2\) outside the block of \(p\_arg1\), so that now the variable blocks constitute a sequential composition in which the first component is the block that declares the variable \(p\_arg2\).

Law \[29\] (\texttt{var}--; right dist)(r/l)

\[
\{le \neq \texttt{null} \land le \neq \texttt{error}\}
\]

\begin{verbatim}
var p_arg2 : T •
  p_arg2 := le.x;
  a2 := p_arg2;
end;
var p_arg1 : T •
  p_arg1 := a1;
  le.x := p_arg1
end
\end{verbatim}

We need the assumption \(\{le \neq \texttt{null} \land le \neq \texttt{error}\}\) before both variable blocks, as we need them for introducing method calls later.
As there are no calls to the method getSetX of class A, we can remove this method from such class.

\begin{verbatim}
class A
  pub x : T; ads a;
  meth getSetX ≜ (val arg1 : T; res arg2 : T •
                     arg2 := self.x; self.x : arg1)
   mtsa
end
\end{verbatim}

We introduce get and set methods for the attribute \(x\) of class \(A\).

\begin{verbatim}
class A
  pub x : T; ads a;
  mtsa
end
\end{verbatim}

We introduce get and set methods for the attribute \(x\) of class \(A\).
blocks we have obtained from the original call to method \textit{getSetX}.

\begin{verbatim}
{\texttt{le} \neq \texttt{null} \land \texttt{le} \neq \texttt{error}}
\textbf{var} \texttt{p\_arg}_2 : T \bullet \\
\quad \texttt{p\_arg}_2 := \texttt{le}.x; \\
\quad a_2 := \texttt{p\_arg}_2;
\textbf{end};
{\texttt{le} \neq \texttt{null} \land \texttt{le} \neq \texttt{error}}
\textbf{var} \texttt{p\_arg}_1 : T \bullet \\
\quad \texttt{p\_arg}_1 := a_1; \\
\quad \texttt{le}.x := \texttt{p\_arg}_1
\textbf{end}
\end{verbatim}

First we rename the variables \texttt{p\_arg}_1 and \texttt{p\_arg}_2 to \texttt{p\_arg}.

\begin{verbatim}
\textbf{Law} [26] \langle \textit{var rename} \rangle \ (2x)
{\texttt{le} \neq \texttt{null} \land \texttt{le} \neq \texttt{error}}
\textbf{var} \texttt{p\_arg} : T \bullet \\
\quad \texttt{p\_arg} := \texttt{le}.x; \\
\quad a_2 := \texttt{p\_arg};
\textbf{end};
{\texttt{le} \neq \texttt{null} \land \texttt{le} \neq \texttt{error}}
\textbf{var} \texttt{p\_arg} : T \bullet \\
\quad \texttt{p\_arg} := a_1; \\
\quad \texttt{le}.x := \texttt{p\_arg}
\textbf{end}
\end{verbatim}

We introduce the parameterised commands that correspond to the variable blocks, applied to proper arguments.

\begin{verbatim}
\textbf{Law} [65] \langle \textit{pcom elimination-res} \rangle \ (r/l), \textbf{Law} [64] \langle \textit{pcom elimination-val} \rangle \ (r/l)
{\texttt{le} \neq \texttt{null} \land \texttt{le} \neq \texttt{error}} \ (\textit{res} \textbf{arg} : T \bullet \textbf{arg} := \texttt{le}.x)(a_2);
{\texttt{le} \neq \texttt{null} \land \texttt{le} \neq \texttt{error}} \ (\textit{val} \textbf{arg} : T \bullet \texttt{le}.x := \textbf{arg})(a_1)
\end{verbatim}

The parameterised commands are the same that define the methods \textit{setX} and \textit{getX}, respectively. Both are preceded by assumptions about the object denoted by \texttt{le}. Therefore, we can introduce calls to the methods \textit{setX} and \textit{getX} of class \textit{A}.

\begin{verbatim}
\textbf{Law} [95] \langle \textit{method call elimination} \rangle \ (r/l) \ (2x)
\texttt{le}.\textit{setX}(a_2); \texttt{le}.\textit{getX}(a_1)
\end{verbatim}
Finally, we change the visibility of attribute \( x \) to private as it is not directly accessed from outside \( A \).

```
class A
  pub x : T; ads_a;
  meth getX = (res arg : T • arg := self.x)
  meth setX = (val arg : T • self.x := arg)
  mts_a
end
```

\[= \text{Law} \langle \text{change visibility: from private to public} \rangle \ (r/1)\]

```
class A
  pri x : T; ads_a;
  meth getX = (res arg : T • arg := self.x)
  meth setX = (val arg : T • self.x := arg)
  mts_a
end
```

This finishes the derivation of the rule \( \langle \text{Separate Query from Modifier} \rangle \).

\[\square\]

### B.3 Encapsulate Downcast

**Derivation.** For deriving this refactoring rule, we follow the steps below.

1. Elimination of calls to the method \( m \) in \( cds \) and \( c \);

2. Elimination of downcasts in the clients;

3. Elimination of the original method \( m \) from class \( A \);

4. Introduction of a new method \( m \) in \( A \) defined by a parameterised command that includes a downcast;

5. Introduction of calls to the newly introduced method \( m \).

For all commands of the form \( \text{var } le_2 : M \bullet le.m(le_2); \ y := (N) le_2 \ \text{end} \), we proceed as follows. First we eliminate calls to \( m \).

```
var le_2 : M \bullet le.m(le_2); \ y := (N)le_2 \ \text{end}
```
APPENDIX B. DERIVATION OF CONTEXTUAL REFACTORIZING RULES

= Law 95 \(\text{method call elimination}\) (l/r)

\[
\begin{array}{ll}
\text{var } le_2 : M \bullet & \{ le \neq \text{null} \land le \neq \text{error} \} (\text{res } arg : M \bullet arg := le_1[le/self](le_2) ; y := (N) le_2) \\
\end{array}
\]

The parameterised command obtained from method elimination is replaced with the command that appears within it, with appropriate substitutions of parameters for arguments.

= Lemma 10 \(\text{r/l}\)

\[
\begin{array}{ll}
\text{var } le_2 : M \bullet & \{ le \neq \text{null} \land le \neq \text{error} \} (le_2 := (N) le_1) \text{[le/self]} ; y := (N) le_2 \text{ end} \\
\end{array}
\]

We can introduce a trivial cast with respect to the expression \(le_1\). The cast has the same type as the declared type of \(le_1\).

= Law 97 \(\text{introduce trivial cast in expressions}\)

\[
\begin{array}{ll}
\text{var } le_2 : M \bullet & \{ le \neq \text{null} \land le \neq \text{error} \} (le_2 := (N) le_1 \text{[le/self]} ; y := (N) le_2) \text{ end} \\
\end{array}
\]

As there are no occurrences of \text{self} in \(le_2\), we can simplify the expression \((le_2 := (N) le_1 \text{[le/self]}\) to \(le_2 := (N) le_1 \text{[le/self]}\).

= Property of substitution

\[
\begin{array}{ll}
\text{var } le_2 : M \bullet & \{ le \neq \text{null} \land le \neq \text{error} \} le_2 := (N) le_1 \text{[le/self]} ; y := (N) le_2 \text{ end} \\
\end{array}
\]

We change the type of \(le_2\) from \(M\) to \(N\) by applying the law Law 99 \(\text{change variable type}\). We can do this because the type of the expression that is assigned to \(le_2\) is a subtype of the declared type of \(le_2\).

= Law 99 \(\text{change variable type}\)

\[
\begin{array}{ll}
\text{var } le_2 : N \bullet & \{ le \neq \text{null} \land le \neq \text{error} \} le_2 := (N) le_1 \text{[le/self]} ; y := (N) le_2 \text{ end} \\
\end{array}
\]

We write the specification statement that corresponds to the assignment to \(le_2\).

= Law 78 \(\text{simple specification}\) (1/r)

\[
\begin{array}{ll}
\text{var } le_2 : N \bullet & \{ le \neq \text{null} \land le \neq \text{error} \} le_2 : [le_2 = (N) le_1 \text{[le/self]}] ; y := (N) le_2 \text{ end} \\
\end{array}
\]
As the type of the expression that is assigned to $le_2$ is $N$, we can strengthen the postcondition of the specification statement to take into account the fact the type of the object currently assigned to $le_2$ is $N$.

\[
\var le_2 : N \bullet \\
\{ le \neq \text{null} \land le \neq \text{error} \} \; le_2 : [le_2 = (N) \; le_1[le/\text{self}] \land le_2 \text{ is } N] ; \; y := (N) \; le_2 \\
\end
\]

From the postcondition of the specification statement, we can introduce an assumption about the type of $le_2$.

\[= \text{Law } 76 \langle \text{merge coercions} \rangle (r/l)\]

\[
\var le_2 : N \bullet \\
\{ le \neq \text{null} \land le \neq \text{error} \} \; le_2 : [le_2 = (N) \; le_1[le/\text{self}] ; \; le_2 \text{ is } N] ; \; y := (N) \; le_2 \\
\end
\]

\[
\var le_2 : M \bullet \\
\{ le \neq \text{null} \land le \neq \text{error} \} \; le_2 : [le_2 = (N) \; le_2] ; \; le_2 \text{ is } N \; \{ le_2 \text{ is } N \} ; \; y := (N) \; le_2 \\
\end
\]

\[= \text{Law } 76 \langle \text{merge coercions} \rangle (1/r)\]

\[
\var le_2 : N \bullet \\
\{ le \neq \text{null} \land le \neq \text{error} \} \; le_2 : [le_2 = (N) \; le_1[le/\text{self}] \land le_2 \text{ is } N] ; \; le_2 \text{ is } N \; \{ le_2 \text{ is } N \} ; \; y := (N) \; le_2 \\
\end
\]

The specification statement $le_2 : [le_2 = (N) \; le_1[le/\text{self}] \land le_2 \text{ is } N]$ is refined by the assignment $le_2 := (N) \; le_1[le/\text{self}]$.

\[= \text{Law } 69 \langle \text{assignment} \rangle\]

\[
\var le_2 : N \bullet \\
\{ le \neq \text{null} \land le \neq \text{error} \} \; le_2 := (N) \; le_1[le/\text{self}] ; \; le_2 \text{ is } N \; \{ le_2 \text{ is } N \} ; \; y := (N) \; le_2 \\
\end
\]

As the type of $le_2$ is the same as that of $y$, we can eliminate the cast to $N$ that is applied to $le_2$ in the assignment to $y$. 
= (eliminate cast of expressions) eliminate cast of expressions

\[ \text{var } le_2 : N \bullet \{ le \neq \text{null} \land le \neq \text{error} \} \]
\[ le_2 := (N)le_1[le/\text{self}]; \{ le_2 \text{ is } N \} \]
\[ y := le_2 \]
\[ \text{end} \]

We merge the assumptions about the type of \( le_2 \) and then we remove the resulting assumption.

= Law [75] (merge assumptions), Predicate calculus

\[ \text{var } le_2 : N \bullet \{ le \neq \text{null} \land le \neq \text{error} \} \]
\[ le_2 := (N)le_1[le/\text{self}]; \{ le_2 \text{ is } N \} \]
\[ y := le_2 \]
\[ \text{end} \]

\[ \subseteq \text{Law [77] (remove assumption), Law [16] (}; \text{ -skip unit)} \]

\[ \text{var } le_2 : N \bullet \{ le \neq \text{null} \land le \neq \text{error} \} \]
\[ le_2 := (N)le_1[le/\text{self}]; \]
\[ y := le_2 \text{end} \]

= Property of substitution

\[ \text{var } le_2 : N \bullet \{ le \neq \text{null} \land le \neq \text{error} \} \]
\[ (le_2 := (N)le_1)[le/\text{self}]; \]
\[ y := le_2 \text{end} \]

As there are no calls to the method \( m \) of class \( A \), we can remove it from such class.

\[ \text{class } A \]
\[ \text{ads}_a; \]
\[ \text{meth } m \triangleright (\text{res } \text{arg} : M \bullet \text{arg} := \text{exp}) \]
\[ mts_a \]
\[ \text{end} \]

= Law [90] (method elimination) (1/r)

\[ \text{class } A \]
\[ \text{ads}_a; \]
\[ mts_a \]
\[ \text{end} \]

We introduce a method \( m \) in class \( A \) defined by a parameterised command whose main command presents a cast to \( N \).
We can introduce a call to the method $m$ of class $A$ in place of all commands like those below, as they are similar to the main command of the parameterised command that defines the method $m$.

First, we introduce the necessary parameterised command.

$$\{ \text{le} \neq \text{null} \land \text{le} \neq \text{error} \} \text{ le}_2 := (N)\text{le}_1; \text{ y := le}_2$$

$$\overset{=}{\text{Law } 90\ (method\ elimination)} (r/l)$$

```plaintext
class A
  ad$ads_a$;
  meth $m$ $\triangleq (\text{res arg : N } \bullet \text{ arg := (N)le}_1)$
  mts$_a$
end
```

Then, we introduce a call to the method $m$.

$$\overset{=}{\text{Law } 95\ (method\ call\ elimination)} (r/l)$$

$$\text{le}.m(\text{le}_2); \text{ y := le}_2$$

This finishes the derivation of the rule $\langle Encapsulate\ Downcast \rangle$. 

$\square$
Appendix C

Lemmas for Program Derivation

Lemma 1  (method call elimination-self).

Consider that the following class declaration

```plaintext
class C extends D  
  ads  
  meth m = pc  
  mts  
end
```

is included in cds. Then

\[ \text{cds}, C \triangleright pc(e) = \text{self}.m(e) \]

Proof

1. Change private attributes that appear in \(pc\) into public.

   Application of Law \([85]\) (change visibility: from private to public), from left to right.

2. Let us call \(p\) the method called on \texttt{super} in \(pc\). Change private attributes that appear in \(p\) into public.

   Application of Law \([85]\) (change visibility: from private to public), from left to right.

3. Eliminate any occurrence of \texttt{super} in \(pc\). This implies in the elimination of \texttt{super} in the whole hierarchy. First, in classes that do not have the definition of the method called on \texttt{super} in \(pc\), introduce a redefinition of such method, which is \(p\) in item 2.
4. Eliminate super in the whole hierarchy.

Application of Law 94 (eliminate super), from left to right, from the first subclass of the class closest to C that introduces a definition or a redefinition of p, in the direction of C, which introduces the call to m on self.

5. Introduction of assumption on self before the parameterised command application pc(e)

Application of Lemma 6

6. Introduce call to method m, having self as target, in class C.

Application of Law 95 (method call elimination), from right to left.

7. Introduce super in the whole hierarchy.

Application of Law 94 (eliminate super), from right to left, from C to the first subclass of the class closest to C that introduces a definition or a redefinition of p.

8. Eliminate occurrences of super in m. This implies in the elimination of super in the whole hierarchy.

Application of Law 88 (introduce method redefinition), from right to left, from the class that introduces the call to m on self in the direction of the first subclass of the class closest to C that introduces a definition or a redefinition of p.


Application of Law 85 (change visibility: from private to public), from right to left.

10. Change public attributes that appear in m into private.

Application of Law 85 (change visibility: from private to public), from right to left.
Lemma 2 (Pull Up Method in the Whole Hierarchy) In class inheritance hierarchy, the definitions of a method in the whole hierarchy are moved to the topmost class that first introduces such method, provided private attributes do not appear in the definition of the method being moved up in such topmost class in the inheritance hierarchy nor in any redefinition of the method being moved up present in the subclasses of such topmost class.

Proof
We call $Z$ the topmost class in the inheritance hierarchy of the class whose method $m$ we want to move up, and that first introduces a definition for it. The steps of this proof are as follows.

1. Change the visibility of private attributes that appear in every definition of $m$ to public;
   
   Application of Law 85 (change visibility: from private to public), from left to right.

2. Introduction of trivial method redefinitions, when there is not an explicit redefinition, in all subclasses of $Z$. We do this from $Z$ in the direction of the leaf subclasses—those in the bottom of the class hierarchy—of $Z$;
   
   Application of Law 88 (introduce method redefinition), from left to right.

3. Elimination of super in all definitions of $m$, starting from the immediate subclasses of $Z$ in the direction of its leaf subclasses;
   
   Application of Law 94 (eliminate super), from left to right.

4. Introduction of (trivial) casts in all definitions of the method $m$;
   
   Application of Law 97 (introduce trivial cast in expressions), from left to right.

5. Exhaustively move up the method $m$ from the bottom of the hierarchy in the direction of $Z$. After each move of $m$ from a class to its superclass, introduce new (trivial) casts in expressions. This should be carried out until we reach the immediate subclasses of $Z$, in which we also introduce trivial casts;

   Application of laws:
   
   Law 89 (move redefined method to superclass), from left to right.
   Law 97 (introduce trivial cast in expressions), from left to right.

$\square$
Lemma 3 (Push Down Method in the Whole Hierarchy) In a class inheritance hierarchy, the definition of a method in the hierarchy is moved from the topmost class in the hierarchy that introduces such method to all its subclasses, provided private attributes do not appear in the definition of the method being moved down in such topmost class.

Proof
We call $Z$ the topmost class in the class inheritance hierarchy of that first introduces a definition $m$, the method we want to move down. The steps of this proof are as follows.

1. Exhaustively moving down the method $m$ from $Z$ in the direction of the bottom of the hierarchy.

   Application of law:

   Law $89$ (move redefined method to superclass), from right to left.

2. Elimination of (trivial) casts in all definitions of the method $m$;

   Application of Law $97$ (introduce trivial cast in expressions), from right to left.

3. Introduction of super in all definitions of $m$, starting from leaf subclasses—those in the bottom of the class hierarchy of the hierarchy in direction of $Z$

   Application of Law $94$ (eliminate super), from right to left.

4. Elimination of method redefinitions, when there is not an explicit redefinition, in all subclasses of $Z$. We do this from leaf subclasses in the direction of $Z$.

   Application of Law $88$ (introduce method redefinition), from right to left.

5. Change the visibility of public attributes that appear in every definition of $m$ to private;

   Application of Law $85$ (change visibility: from private to public), from right to left.

□
Example. Here we present an example of the application of Lemmas 2 and 3. The derivation from Step 1 to Step 5 corresponds to Lemma 2. The derivation in the reverse direction corresponds to Lemma 3. Let us consider the following class declarations.

```plaintext
class D
  ads
  meth m (val x : T • c_d)
  mts_c [self.m(a)]
end
class B extends D
  ads_b
  meth m (val x : T • c_b)
  mts_b [self.m(a)]
end
class A extends D
  ads_a
  meth m (val x : T • c_a)
  mts_a [self.m(a)]
end
class C extends A
  ads_c
  meth m (val x : T • c_c)
  mts_c [self.m(a)]
end
cds [le.m(a)] • c[le.m(a)]
```

In this example, we want to move up all definitions of m to D. The argument a is an expression of type T.

Step 1 First we change the visibility of private attributes that appear the every definition of m from private to public. We apply Law 85 (change visibility: from private to public), from left to right.

Step 2 The class C does not redefine the method m: it does not have an explicit definition of m. We must introduce such definition by applying the Law 88 (introduce method redefinition).

```plaintext
class C extends A
  ads_c
  meth m (val x : T • c_c)
  mts_c
end
```

Step 3 By applying the Law 94 (eliminate super), we eliminate the occurrence of super in the method m of class C, resulting in the command c_c, which is, in fact, the same as c_a.

```plaintext
class C extends A
  ads_c
  meth m (val x : T • c_c)
  mts_c
end
```

Step 4 We need to introduce trivial casts in all definitions of m in the subclasses of D so that we can apply the law for moving redefined methods to the superclass in a later step. To introduce the trivial casts, we apply Law 97 (introduce trivial cast in expressions).
Step 5 Now we move all definitions of $m$ in the direction of $D$, as this is the topmost class in the inheritance hierarchy of $A$ that first introduces a definition of $m$. We exhaustively apply the move redefined method to superclass, moving method $m$ from $C$ to $A$. This result in the following method $m$ in class $A$, in which there are occurrences of self without casts.

$$\begin{align*}
    \text{if } \text{self is } C &\rightarrow c'_c \\
    [\] \neg (\text{self is } C) &\rightarrow c'_a \\
    \text{fi}
\end{align*}$$

where $c'_a$ indicates that this command differs from the original $c_a$ by the introduction of trivial casts in expressions present in $c_a$. Similarly, the command $c'_c$ denotes the command $c_c$ that was present in class $C$ with trivial casts in expressions. In order to move this method from $A$ to $D$, it is necessary to introduce casts, as required by the law for moving redefined methods to a superclass. Again, we apply Law 89 (introduce trivial cast in expressions), yielding the following alternation.

$$\begin{align*}
    \text{if } (A)\text{self is } C &\rightarrow c'_c \\
    [\] \neg ((A)\text{self is } C) &\rightarrow c'_a \\
    \text{fi}
\end{align*}$$

The method $m$ of class $A$ can be moved to $D$. As in the previous steps, after we introduce trivial casts, we can move such method from $B$ to $D$. Finally, we move the definitions of $m$ in $B$ and $A$ to $D$. This yields the following classes.

```
class D

\begin{align*}
    \text{meth } m \triangleq (\text{val } x : T \bullet \\
    &\text{if self is } B \rightarrow c'_b \\
    &\neg (\text{self is } B) \rightarrow \text{if } \neg (\text{self is } A) \text{is } A \rightarrow c_d \\
    &\neg (\text{self is } A) \rightarrow \text{if } (A)\text{self is } C \rightarrow c'_c \\
    &\neg ((A)\text{self is } C) \rightarrow c'_a \\
    \text{fi}
\end{align*}

\text{f)}

\text{mts}_d[\text{self}.m(a)]

\text{end}
```
This exemplifies how we move all definitions of a method to the topmost class that first introduces a definition of that method. Following the proposed steps in the reverse order, moves the definition of a method in the direction of the subclasses of the class where the method is defined.

\[ \text{Lemma 4 (Eliminate super in Hierarchy)} \]

In an inheritance hierarchy, the occurrences of super are eliminated in the definition of a method and in superclasses, which do not redefine such method, of the class in which we want to eliminate the occurrences of super.

**Proof**

We call Z the superclass in the inheritance hierarchy of the class whose method m we want to remove occurrences of super from. The class Z first introduces a definition for m or redefines it. The steps of this proof are as follows.

1. Change the visibility of private attributes that appear in the topmost definition or redefinition of m to public;

   Application of Law 85 \((\text{change visibility: from private to public})\), from left to right.

2. Introduction of trivial method redefinitions, when there is not an explicit redefinition, in all subclasses of Z. We do this from Z in the direction of the leaf subclasses—those in the bottom of the class hierarchy—of Z;

   Application of Law 88 \((\text{introduce method redefinition})\), from left to right.

3. Elimination of super in all definitions of m, starting from the immediate subclasses of Z in the direction of its leaf subclasses;

   Application of Law 94 \((\text{eliminate super})\), from left to right.
Lemma 5 \((\text{Eliminate super and Trivial Methods in Hierarchy})\) In an inheritance hierarchy super is introduced in every trivial definition of a method \(m\) that appear in subclasses of a class that is the first to introduce a definition for \(m\) or is the last to redefine \(m\).

**Proof**

We call \(Z\) the first superclass in the inheritance hierarchy of the class whose method \(m\) we want to remove occurrences of super from, and that first introduces a definition for it or redefines it. The steps of this proof are as follows.

1. Introduction of super in all definitions of \(m\), starting from the leaf subclasses of \(Z\) in the direction of \(Z\);

   Application of Law \(\#3\) (eliminate super), from right to left.

2. Elimination of trivial method redefinitions in all subclasses of \(Z\). We do this from \(Z\) in the direction of the root subclasses—those in the top of the class hierarchy—of \(Z\);

   Application of Law \(\#8\) (introduce method redefinition), from right to left.

3. Change the visibility of private attributes that appear in the topmost definition or redefinition of \(m\) to private;

   Application of Law \(\#5\) (change visibility: from private to public), from right to left.

\(\square\)

Lemma 6 \((\text{self non-null non-error})\) An assumption about self, which is not null nor error is introduced in any position in a method of a class.

**Proof**

The steps of this proof are as follows.

- Introduction of class invariant \(\{\text{self} \neq \text{null} \land \text{self} \neq \text{error}\}\)

  Application of Law \(\#107\) (introduce class invariant), from left to right.

- Moving the invariant to the adequate code position

  Applications of Law \(\#53\) (assumption before or after command), from right to left.

\(\square\)
Lemma 7 (Type changes in program)

\[ cds_1 \bullet c_1 = cds_2 \bullet c_2 \]

where

\[ cds_2 \triangleq cds_1 \begin{array}{l}
\text{visib at : } M, \var v : M, \par p : M/visib at : N, \var v : N, \par p : N
\end{array} \]

where \( \text{visib} \in \{ \text{pri, prot, pub} \} \) and \( \par \in \{ \text{val, res} \} \)

\[ c_2 \triangleq c_1 \begin{array}{l}
\var x : B, \par p : M/\var x : C, \par p : N
\end{array} \]

provided

\(-\) (1) every non-assignable occurrence of attributes, local variables, value and result parameters of type \( N \) in expressions are cast with \( N \) or any subtype of \( N \); (2) every actual parameter associated with formal parameters of type \( N \) in \( cds_1 \) and \( c_1 \) are of type \( M \) or any supertype of \( M \).

\(\leftarrow\) (1) every non-assignable occurrence of attributes, local variables, value and result parameters of type \( M \) in expressions are cast with \( N \) or any subtype of \( N \); (2) every expression assigned to an attribute of type \( M \) is of type \( N \) or any subtype of \( N \); (3) every use of an attribute of type \( M \) as a result argument is for a corresponding formal parameter of type \( N \) or any subtype of \( N \); (4) every actual parameter associated with a value parameter of type \( M \) is of type \( N \) of any subtype of \( N \); (5) every expression assigned to a value argument of type \( M \) is of type \( N \) or any subtype of \( N \); (6) every use of a value argument of type \( M \) as result argument is for a corresponding formal parameter of type \( N \) or any subtype of \( N \); (7) every use of a result parameter of type \( M \) as a result argument is for a corresponding formal parameter of type \( N \) or any subtype of \( N \); (8) every expression assigned to a result parameter of type \( M \) is of type \( N \) or any subtype of \( N \); (9) every expression assigned to variables of type \( M \) is of type \( N \) or any subtype of \( N \); (10) and every use of variables of type \( M \) is for a corresponding formal parameter of type \( N \) or any subtype of \( N \).

Proof (From left to right) By the application of laws

- Law [86] ⟨change attribute type⟩, from left to right;
- Law [99] ⟨change variable type⟩, from left to right;
- Law [93] ⟨change result parameter type⟩, from left to right;
- Law [92] ⟨change value parameter type⟩, from left to right.
(From right to left) By the application of laws

- Law [86] (*change attribute type*), from right to left;
- Law [99] (*change variable type*), from right to left;
- Law [93] (*change result parameter type*), from right to right;
- Law [92] (*change value parameter type*), from right to right.

Lemma 8 (*Program cast elimination*) Type casts for a class \( N \) that appears in program \( \text{cds} \bullet c \) are removed along with any assumption with type tests provided.

**Proof** By the application of the following laws

- Law [96] (*eliminate cast of method call*), from left to right;
- Law [98] (*eliminate cast of expressions*), from left to right;

For each assumption that results from application of the previous laws, apply laws

- Law [77] (*remove assumption*), from left to right;
- Law [16] (*; −skip unit*)

in this order.

Lemma 9 (*pcom value-argument*)

\[
c = (\text{val } vl : T \bullet c[vl/x]) (x)
\]

provided \( vl \) is fresh—not free in \( c \)—; in \( c \), \( x \) is not on the left-hand side of assignments, it is not a result argument, \( x \) does not occur in the frame of specification statements in \( c \), \( x \) is not a method call target, and \( x \neq \text{error} \)

**Proof**

\[
c
= \text{var } l : T \bullet l := x; \ c[l/x] \text{ end} \quad \text{[by Law [62] (*var block-val*)]}
= (\text{val } vl : T \bullet c[vl/x]) (x) \quad \text{[by a syntactic transformation]}
\]
Lemma 10 (pcom result-argument)

\[ c = (\text{res } vl : T \bullet c[vl/x])(x) \]

provided \( vl \) is fresh—not free in \( c \)—; in \( c \), \( x \) is not on the right-hand side of assignments and it is not a value argument nor a method call target, \( x \) is not used in attribute selection nor in update expression.

Proof

\[
\begin{align*}
c & = \text{var } l : T \bullet c[l/x]; \ x := l \text{ end} & \quad \text{[by Law 63 (var block-res)]} \\
& = (\text{res } vl : T \bullet c[vl/x])(x) & \quad \text{[by a syntactic transformation]}
\end{align*}
\]
Appendix D

Laws of Commands

D.1 Assignment

Law 1 (\(\langle \mathrm{:=} \rangle \mathrm{skip}\))
If \(le \neq \mathrm{error}\), then \((le := le) = \mathrm{skip}\)

Law 2 (\(\langle \mathrm{:=} \mathrm{identity}\rangle\))
If \(e \neq \mathrm{error}\), then \((le, le_1 := e, le_1) = (le := e)\)

Law 3 (\(\langle \mathrm{:=} \mathrm{symmetry}\rangle\))
If \(i\) ranges over 1..\(n\) and \(\pi\) is any permutation of 1..\(n\), then
\((le_i := e_i) = (le_{\pi(i)} := e_{\pi(i)})\), provided \(e_i \neq \mathrm{error}\)

D.2 Conditional

Law 4 (\(\langle \mathrm{if} \ \mathrm{symmetry}\rangle\))
If \(i\) ranges over 1..\(n\) and \(\pi\) is any permutation of 1..\(n\), then
\(\text{if } \bot \cdot \psi_i \rightarrow c_i \text{ fi } = \text{if } \bot \cdot \psi_{\pi(i)} \rightarrow c_{\pi(i)} \text{ fi}\)

Law 5 (\(\langle \mathrm{if} \ \mathrm{true \ guard}\rangle\))
\(\text{if } \text{true} \rightarrow c \text{ fi } = c\)
Law 6 \( \text{if false unity} \)
if false \( \rightarrow \) c \( [] \psi_i \rightarrow c_i \) fi = if \( \psi_i \rightarrow c_i \) fi

Law 7 \( \text{if abort unity} \)
if () fi = abort

Law 8 \( \text{if } \lor \text{ distrib1} \)
if \( \psi_1 \rightarrow c \) \( [] \psi_2 \rightarrow c \) \( (\psi_i \rightarrow c_i) \) fi = if \( \psi_1 \lor \psi_2 \rightarrow c \) \( [] \) \( (i \bullet \psi_i \rightarrow c_i) \) fi

Law 9 \( \text{if elim} \)
If \( i \) ranges over 1..n
\[
\begin{align*}
\text{if } \psi & \rightarrow \left( \begin{array}{c}
\text{if } \psi \rightarrow c \\
[] \ (i \bullet \psi_i \rightarrow c_i) \\
\text{fi}
\end{array} \right) \\
\text{fi} & = \left( \begin{array}{c}
\text{if } \psi \rightarrow c \\
\text{fi}
\end{array} \right) \\
\right. \\
\end{align*}
\]

Law 10 \( \text{if } \lor \text{ distrib2} \)
\[
\begin{align*}
\text{if } \psi_1 & \rightarrow c_1 \\
\begin{array}{c}
\left[ \right] \neg \psi_1 \rightarrow \left( \begin{array}{c}
\text{if } \psi_2 \rightarrow c_1 \\
\left[ \left( \left[ i \bullet \psi_i \rightarrow c_i \right) \\
\text{fi}
\end{array} \right) \\
\text{fi}
\end{array} \right) \\
\text{fi} & = \text{if } \psi_1 \lor \psi_2 \rightarrow c_1 \\
\begin{array}{c}
\left[ \right] \neg (\psi_1 \lor \psi_2) \rightarrow c_2 \\
\text{fi}
\end{array} \\
\right. \\
\end{align*}
\]

Law 11 \( \text{if } \land \text{ distrib} \)
\[
\begin{align*}
\text{if } \psi_1 & \rightarrow \left( \begin{array}{c}
\text{if } \psi_2 \rightarrow c_1 \\
\left[ \right] \neg \psi_2 \rightarrow c_2 \\
\text{fi}
\end{array} \right) \\
\text{fi} & = \text{if } \psi_1 \land \psi_2 \rightarrow c_1 \\
\begin{array}{c}
\left[ \right] \neg \psi_1 \rightarrow c_2 \\
\text{fi}
\end{array} \\
\right. \\
\end{align*}
\]
D.3. RECURSION

Law 12 (if gc intro)
If $i$ ranges over $1..n$ and $\neg (\psi \land \psi_i)$, for all $i$, then
if $[[i \cdot \psi_i \rightarrow c_i] \fi] \subseteq \text{if } ([[] i \cdot \psi_i \rightarrow c_i)] \psi \rightarrow c \fi$

Law 13 (if gc elim)
If $i$ ranges over $1..n$ and $\psi \Rightarrow \forall i \cdot \psi_i$, then
if $\psi \rightarrow \text{c} [[[][] i \cdot \psi_i \rightarrow c_i] \fi] \subseteq \text{if } ([[] i \cdot \psi_i \rightarrow c_i) \fi$

Law 14 (if true guard ref)
if true $\rightarrow \text{c} [[] (i \cdot \psi_i \rightarrow c_i) \fi] \subseteq \text{c}$

D.3 Recursion

Law 15 (rec fixed point)
rec $X \cdot F(X) = F(\text{rec} \cdot F(X))$

D.4 Sequential Composition

Law 16 (; -skip unit)
(skip; $c) = c = (c; \text{skip})$

Law 17 (; -abort left zero)
abort; $c = \text{abort}$

Law 18 (; -miracle left zero)
miracle; $c = \text{miracle}$
Law 19 (⟨; assoc⟩)
\((c_1; c_2); c_3 = c_1; (c_2; c_3)\)

Law 20 (⟨; −if left dist⟩)
If \(i\) ranges over 1..\(n\)
\(\text{if } [] i \cdot \psi_i \rightarrow c_i \text{ fi: } c = \text{ if } [] i \cdot \psi_i \rightarrow (c_i; c) \text{ fi}\)

Law 21 (⟨; −if selection⟩)
If \(i\) and \(j\) range over 1..\(n\) and \(\neg (\psi_j \land \psi_i)\), then for \(j \neq i\) we have
\([\psi_j]\text{ if } [] i \cdot \psi_i \rightarrow c_i \text{ fi } = [\psi_j]; c_j\)

Law 22 (⟨; −:= combination⟩)
\((le := e_1; le := e_2) = (le := e_2[e_1/le])\)

Law 23 (⟨:= −<> right dist⟩)
If \(e\) is total, then
\(le := e; \text{ if } [] i \cdot \psi_i \rightarrow c_i \text{ fi } = \text{ if } [] i \cdot \psi_i[e/le] \rightarrow (le := e; c_i) \text{ fi}\)

D.5 Local Variable Block

Law 24 (var symmetry)
\(\text{var } x_1 : T_1 \cdot (\text{var } x_2 : T_2 \cdot c \text{ end}) \text{ end } = \text{var } x_2 : T_2 \cdot (\text{var } x_1 : T_1 \cdot c \text{ end}) \text{ end}\)

Law 25 (var elim)
If \(x\) is not free in \(c\), then \(\text{var } x : T \cdot c \text{ end } = c\)

Law 26 (var rename)
If \(x_2\) is not free in \(c\), then
\(\text{var } x_1 : T \cdot c \text{ end } = \text{var } x_2 : T \cdot c[x_2/x_1] \text{ end}\)
Law 27 \( \langle \text{var} - \text{if dist} \rangle \)
If \( i \) ranges over 1..\( n \) and \( x \) is not free in \( \psi_i \), then
\[
\text{if } \lbrack \lbrack i \bullet \psi_i \rightarrow (\text{var} \; x \cdot T \cdot c_i \; \text{end}) \rbrack \rbrack = \text{var} \; x : T \bullet \text{if } \lbrack \lbrack i \bullet \psi_i \rightarrow c_i \rbrack \rbrack \text{ end}
\]
\( \square \)

Law 28 \( \langle \text{var} - ; \text{left dist} \rangle \)
If \( x \) is not free in \( c_2 \), then \( \text{var} \; x : T \bullet c_1 \; \text{end} ; c_2 = \text{var} \; x : T \bullet c_1 ; c_2 \; \text{end} \)
\( \square \)

Law 29 \( \langle \text{var} - ; \text{right dist} \rangle \)
If \( x \) is not free in \( c_1 \), then \( c_1 ; \text{var} \; x : T \bullet c_2 \; \text{end} = \text{var} \; x : T \bullet c_1 ; c_2 \; \text{end} \)
\( \square \)

Law 30 \( \langle \text{var} - := \text{final value} \rangle \)
If \( x \) is not free in \( c \), then \( \text{var} \; x : T \bullet c ; x := e \; \text{end} = \text{var} \; x : T \bullet c \; \text{end} \)
\( \square \)

Law 31 \( \langle \text{var} - ; \text{dist} \rangle \)
\( \text{var} \; x : T \bullet c_1 \; \text{end} ; \text{var} \; x : T \bullet c_2 \; \text{end} \sqsubseteq \text{var} \; x : Tc_1 ; c_2 \; \text{end} \)
\( \square \)

Law 32 \( \langle \text{var} - := \text{initial value} \rangle \)
Provided \( e \neq \text{error} \), then \( \text{var} \; x : T \bullet c \; \text{end} \sqsubseteq \text{var} \; x : T \bullet x := e ; c \; \text{end} \)
\( \square \)

D.6 Angelic Variable Block

Law 33 \( \langle \text{avar symmetry} \rangle \)
\( \text{avar} \; x_1 : T_1 \bullet (\text{avar} \; x_2 : T_2 \bullet c \; \text{end}) \; \text{end} = \text{avar} \; x_2 : T_2 \bullet (\text{avar} \; x_1 : T_1 \bullet c \; \text{end}) \; \text{end} \)
\( \square \)

Law 34 \( \langle \text{avar elim} \rangle \)
If \( x \) is not free in \( c \), then \( \text{avar} \; x : T \bullet c \; \text{end} = c \)
\( \square \)
Law 35 ⟨avar rename⟩
If \( x_2 \) is not free in \( c \), then
\[
\text{avar } x_1 : T \cdot c \text{ end} = \text{avar } x_2 : T \cdot c[x_2/x_1] \text{ end}
\]

Law 36 ⟨avar - if dist⟩
If \( i \) ranges over \( 1..n \) and \( x \) is not free in \( \psi_i \), then
\[
\text{if } [[]i \cdot \psi_i \rightarrow (\text{avar } x : T \cdot c_i \text{ end}) \text{ fi } = \text{avar } x : T \cdot \text{if } [[]i \cdot \psi_i \rightarrow c_i \text{ fi end}}
\]

Law 37 ⟨avar-; left dist⟩
If \( x \) is not free in \( c_2 \), then
\[
\text{avar } x : T \cdot c_1 \text{ end}; c_2 = \text{avar } x : T \cdot c_1; c_2 \text{ end}
\]

Law 38 ⟨avar-; right dist⟩
If \( x \) is not free in \( c_1 \), then
\[
c_1; \text{ avar } x : T \cdot c_2 \text{ end} = \text{avar } x : T \cdot c_1; c_2 \text{ end}
\]

Law 39 ⟨avar- := final value⟩
If \( x \) is not free in \( c \), then
\[
\text{avar } x : T \cdot c; x \cdot := e \text{ end} = \text{avar } x : T \cdot c \text{ end}
\]

Law 40 ⟨avar - var relationship⟩
If \( x \) is not free in \( e \), then
\[
\text{avar } x : T \cdot \{x = e\}; c \text{ end} = \text{var } x : T \cdot [x = e]; c \text{ end}
\]

Law 41 ⟨avar-; dist⟩
\[
\text{avar } x : T \cdot c_1; c_2 \text{ end} \sqsubseteq \text{avar } x : T \cdot c_1 \text{ end}; \text{avar } x : T \cdot c_2 \text{ end}
\]

Law 42 ⟨avar- := initial value⟩
Provided \( e \neq \text{error} \), then \[
\text{avar } x : T \cdot x \cdot := e; c \text{ end} \sqsubseteq \text{avar } x : T \cdot c \text{ end}
\]
D.7. ADDITIONAL LAWS

Law 43 ⟨var - avar refinement⟩
\[ \text{var} x : T \cdot c \text{ end} \sqsubseteq \text{avar} x : T \cdot c \text{ end} \]

\[ \square \]

D.7 Additional Laws

In this section we present some additional command laws that have arisen along the derivation of refactoring rules.

D.7.1 Alternation

Law 44 ⟨if identical guarded commands⟩
If \( \bigvee i : 1..n \cdot \psi_i = \text{true} \), then
\[ \text{if} \[i : 1..n \cdot \psi_i \rightarrow c \text{ fi} = c \]

\[ \square \]

D.7.2 Guards

Law 45 ⟨iteration guards⟩
\[
\text{do} \[i \cdot g_i \rightarrow c_i \text{ od} \sqsubseteq \text{do} \[i \cdot h_i \rightarrow c_i \text{ od}
\]

provided

i. \( h_i \Rightarrow g_i \)

ii. \( gg \Rightarrow hh \)

\[ \square \]

Law 46 ⟨weakening guards⟩
\[
\text{if} \[i \cdot g_i \land g \rightarrow c_i \text{ fi} \sqsubseteq \text{if} \[i \cdot g_i \rightarrow c_i \text{ fi}
\]

provided

i. \( (\forall i \cdot g_i \land g) \Rightarrow (\forall i \cdot g_i) \)

ii. \( (\forall i \cdot g_i \land g) \land g_i \Rightarrow g_i \land g \)

\[ \square \]
D.7.3 Assumptions

Law 47 (innocuous assumption-reading)
If \( x \) is an attribute of the object denoted by \( le \), then
\[
le_1 := \text{le}.x = \{ \text{le} \neq \text{null} \land \text{le} \neq \text{error} \}; \; le_1 := \text{le}.x
\]

Law 48 (innocuous assumption-writing)
If \( x \) is an attribute of the object denoted by \( le \), then
\[
\text{le}.x := \text{exp} = \{ \text{le} \neq \text{null} \land \text{le} \neq \text{error} \}; \; \text{le}.x := \text{exp}
\]

Law 49 (assumption guard)
\[
\text{if } [] i \cdot \psi_i \rightarrow c_i \text{ fi } = \text{if } [] i \cdot \psi_i \rightarrow \{ \psi_i \} \; c_i \text{ fi}
\]

Law 50 (var block absorb assumption)
If \( x \) is not free in \( \psi \), then
\[
\{ \psi \} \operatorname{var} x : T \cdot c \; \text{end} = \operatorname{var} x : T \cdot \{ \psi \} \; ; \; c \; \text{end}
\]

Law 51 (alternation absorb assumption)
\[
\{ \phi \} \text{if } \psi_i \rightarrow c_i \text{ fi } = \text{if } \phi \land \psi_i \rightarrow c_i \text{ fi}
\]

Law 52 (pcom absorb assumption)
If the parameters present in \( pd \) and \( pds \) do not occur free in \( \phi \), then
\[
\{ \phi \}(pd; \; pds \cdot c)(a_n) = (pd; \; pds \cdot \{ \phi \}c)(a_n)
\]

Law 53 (assumption before or after command)
Provided the free variables of \( \phi \) are not free in expressions on the left-hand side of assignments in \( c \), they are not used as result arguments \( c \), they are not in the frame of specification statements in \( c \), and they are not targets of method calls, then
\[
\{ \phi \} \; c = c; \; \{ \phi \}
\]
Law 54 \textit{(assumption advance command)}

If free variables of $\phi$ are not on the left-hand side of assignments in $c$, they are not result arguments in method calls in $c$, they are not in the frame of specification statements in $c$, and they are not targets of method calls, then

$$\{\phi\} c = \{\phi\} c; \{\phi\}$$

\hfill $\square$

Law 55 \textit{(assumption distribution)}

Provided the free variables of $\phi$ are not free in expressions on the left-hand side of assignments in $c_i$, they are not used as result arguments $c_i$, they are not in the frame of specification statements in $c_i$, and they are not targets of method calls, then

$$\{\phi\} c[i : 1..n \cdot c_i] = c[i : 1..n \cdot \{\phi\} c_i]$$

\hfill $\square$

Law 56 \textit{(new assumption)}

If $c$ is the body of $\text{new}$ of class $N$, $l$ is the list of global variables of $c$, and $l : [\text{true, true}] \sqsubseteq c$, then

$$x := \text{new} \; N() \sqsubseteq x := \text{new} \; N(); \{x \neq \text{null} \land x \neq \text{error}\}$$

\hfill $\square$

Law 57 \textit{(assumption intro)}

$$w : [\psi_1, \psi_2] \sqsubseteq w : [\psi_1, \psi_2]; \{\psi_2\}$$

\hfill $\square$

D.7.4 Assignment

Law 58 \textit{(repeated assignment)}

$$(le := e) = (le := e; le := e)$$

\hfill $\square$

Law 59 \textit{(order independent assignment)}

If $x$ and $y$ are not free in $e_2$ and $e_1$, respectively, then
\[(x := e_1; y := e_2) = (y := e_2; x := e_1)\]
Law 64 \langle pcom elimination-val \rangle

\[(\text{val} \ vl : T \bullet c)(x) = \text{var} \ l : T \bullet l := x; \ c[l/vl] \ \text{end}\]

provided the variables of \(l\) are fresh: not free in \(c, x, \) and \(vl\). Variables in \(vl\) do not appear on the left-hand side of assignments, are not used as result arguments, do not occur in the frame of specification statement, nor are method call targets, and they are not error.

\[\square\]

Law 65 \langle pcom elimination-res \rangle

\[(\text{res} \ vl : T \bullet c)(x) = \text{var} \ l : T \bullet c[l/vl]; \ x := l \ \text{end}\]

provided the variables of \(l\) are fresh: not free in \(c, x, \) and \(vl\). Variables in \(vl\) are not on the right-hand side of assignments, they are not used as value arguments nor are method call targets, and they are not used in attribute selection nor in update expression.

\[\square\]

Law 66 \langle pcom merge \rangle

\[(\text{par} \ x : T; \ pds \bullet c)(a_1, a_2) = (\text{par} \ x : T; (pds \bullet c)(a_2))(a_1)\]

where \(\text{par} \in \{\text{val, res}\}\), and provided the variables of \(x\) are not free in \(a_2\).

\[\square\]

Law 67 \langle command refinement-class refinement \rangle

\[
\begin{array}{c}
\text{class} \ N \\
ad_{n}; \ \\
meth \ m \triangleq (pds \bullet c_1) \\
mts_{n} \\
\text{end}
\end{array}
\quad
\begin{array}{c}
\text{class} \ N \\
ad_{n}; \ \\
meth \ m \triangleq (pds \bullet c_2) \\
mts_{n} \\
\text{end}
\end{array}
\]

provided

\(c_1 \subseteq c_2\)

\[\square\]
D.8 Laws from Morgan’s work [64]

In this section, we present laws proposed by Morgan [64] that we have used in our work.

Law 68 \textit{(absorb coercion)}

\[
  w : [\psi_1, \psi_2]; [\phi] = w : [\psi, \psi_2] \land \phi
\]

\[
  \square
\]

Law 69 \textit{(assignment)}

If \( \psi_1 \Rightarrow \psi_2[E/w] \), then \( w, x : [\psi_1, \psi_2] \sqsubseteq w := E \)

\[
  \square
\]

Law 70 \textit{(augment assignment)}

The assignment \( w := E \) can be replaced by the fragment \( w, c := E, F \) provided that \( CI \Rightarrow CI[E, F/w, c] \)

\[
  \square
\]

Law 71 \textit{(augment guards)}

The guard \( G \) may be replaced by \( G' \) provided that \( [CI \Rightarrow (G \Leftrightarrow G')] \)

\[
  \square
\]

Law 72 \textit{(augment specification)}

The specification \( w : [\psi_1, \psi_2] \) becomes \( w, c : [CI \land \psi_1, CI \land \psi_2] \)

\[
  \square
\]

Law 73 \textit{(diminish assignment)}

If \( E \) contains no variables \( a \), then the assignment \( w, a := E, F \) can be replaced by the assignment \( w := E \).

\[
  \square
\]

Law 74 \textit{(diminish specification)}

The specification \( w, a : [\psi_1, \psi_2] \) becomes

\[
  w : [(\exists a : A \bullet \psi_1), (\forall a_0 : A \bullet \psi_{10} \Rightarrow (\exists a : A \bullet \psi_2))] \]

\[
  \square
\]
Law 75 \( \langle \text{merge assumptions} \rangle \)
\[
\{ \psi_2 \} \{ \psi_2 \} = \{ \psi_1 \land \psi_2 \}
\]

Law 76 \( \langle \text{merge coercions} \rangle \)
\[
[\psi_1][\psi_2] = [\psi_1 \land \psi_2]
\]

Law 77 \( \langle \text{remove assumption} \rangle \)
\[
\{ \psi \} \sqsubseteq \text{skip}
\]

Law 78 \( \langle \text{simple specification} \rangle \)
Provided \( E \) contains no \( w \),
\[
w := E = w : [w = E]
\]

Law 79 \( \langle \text{strengthen postcondition} \rangle \)
If \( \psi'_2 \Rightarrow \psi_2 \), then
\[
w : [\psi_1, \psi_2] \sqsubseteq w : [\psi_1, \psi'_2]
\]

Law 80 \( \langle \text{weaken precondition} \rangle \)
If \( \psi_1 \Rightarrow \psi'_1 \), then
\[
w : [\psi_1, \psi_2] \sqsubseteq w : [\psi'_1, \psi_2]
\]

D.9 Data Refinement

Law 81 \( \langle \text{Data refinement—variable blocks with initialisation} \rangle \)
\[
\text{var} \ vl; \ \text{avl} \bullet \text{avl} : [\text{true, init}]; \ c_1 \ \text{end}
\]
\[
\sqsubseteq
\]
\[
\text{var} \ vl; \ \text{cvl} \bullet \text{cvl} : [\text{true, } (\exists \text{avl} \bullet CI \land init)]; \ c_2 \ \text{end}
\]
provided

c_1 \preceq c_2

The variables of \text{cvl} are not free in \text{init} and \text{c}_1, and are not in \text{avl};

The variables of \text{avl} are not free in \text{c}_2;

□
Appendix E

Laws of Classes

E.1 Normal Form Laws

In this section, we present laws that are useful for the description and justification of a strategy for reducing programs to a normal form expressed in terms of restricted subset of ROOL \[17\], \[16\]. The major application of these laws is to formally derive elaborate behaviour preserving program transformations.

E.1.1 Class Declaration

Law 82 (class elimination)

\[c_{\text{ds}} \cdot c_{1} \cdot c = c_{\text{ds}} \cdot c\]

provided

\((\rightarrow)\) The class declared in \(c_{d1}\) is not referred to in \(c_{\text{ds}}\) or \(c\);

\((\leftarrow)\) (1) The name of the class declared in \(c_{d1}\) is distinct from those of all classes declared in \(c_{\text{ds}}\); (2) the superclass appearing in \(c_{d1}\) is either object or declared in \(c_{\text{ds}}\); (3) and the attribute and method names declared by \(c_{d1}\) are not declared by its superclasses in \(c_{\text{ds}}\), except in the case of method redefinitions.
E.1.2 Attribute Declaration

Law 83 (attribute elimination)

\[
\text{class } B \text{ extends } A
\begin{align*}
\text{pri } a &: T; \quad \text{ads} \\
\text{ops} \\
\text{end}
\end{align*}
\]

\[
\text{class } B \text{ extends } A
\begin{align*}
\text{ads} \\
\text{ops} \\
\text{end}
\end{align*}
\]

provided

\(\rightarrow\) \(B.a\) does not appear in \(\text{ops}\);

\(\leftarrow\) \(a\) does not appear in \(\text{ads}\) and is not declared as an attribute by a superclass or subclass of \(B\) in \(\text{cds}\).

\[\square\]

Law 84 (change visibility: from protected to public)

\[
\text{class } C \text{ extends } D
\begin{align*}
\text{prot } a &: T; \quad \text{ads} \\
\text{ops} \\
\text{end}
\end{align*}
\]

\[
\text{class } C \text{ extends } D
\begin{align*}
\text{pub } a &: T; \quad \text{ads} \\
\text{ops} \\
\text{end}
\end{align*}
\]

provided

\(\leftarrow\) \(B.a\), for any \(B \leq C\), appears only in \(\text{ops}\) and in the subclasses of \(C\) in \(\text{cds}\).

\[\square\]

Law 85 (change visibility: from private to public)

\[
\text{class } C \text{ extends } D
\begin{align*}
\text{pri } a &: T; \quad \text{ads} \\
\text{ops} \\
\text{end}
\end{align*}
\]

\[
\text{class } C \text{ extends } D
\begin{align*}
\text{pub } a &: T; \quad \text{ads} \\
\text{ops} \\
\text{end}
\end{align*}
\]

provided

\(\leftarrow\) \(B.a\), for any \(B \leq C\), does not appear in \(\text{cds}\), \(c\).

\[\square\]
The law that allows us to change attribute visibility from private to protected, and vice-versa, can be derived from the above two laws. For instance, you can make a private attribute protected, by making it public, with an application of Law \[85\] and then protected, by applying Law \[84\].

In subsequent laws we use the concept of non-assignable identifiers. We define the concept of assignable and non-assignable occurrences of identifiers \[17, 16\]. Assignable occurrences of identifiers are result arguments and targets of assignments. For instance, in \texttt{self.a := e} and \texttt{le.m(self.a)}, the occurrences of \texttt{a} are assignable, if the single parameter of \texttt{m} is passed by result. On the other hand, in an assignment \texttt{self.a.x := e}, there is an assignable occurrence of \texttt{x} but not of \texttt{a}. Therefore, \texttt{a} is required to be cast in the proviso above. The same comment applies to a result argument \texttt{self.a.x}.

Occurrences of identifiers as result arguments and targets of assignments are not cast anywhere; like in Java, this is not allowed in ROOL \[17, 16\].

**Law 86** (change attribute type)

\[
\begin{array}{l}
\text{class } C \text{ extends } D \\
\text{ pub } a : T; \text{ ads} \\
\text{ ops} \\
\text{ end}
\end{array}
=_{cds,c}
\begin{array}{l}
\text{class } C \text{ extends } D \\
\text{ pub } a : T'; \text{ ads} \\
\text{ ops} \\
\text{ end}
\end{array}
\]

provided

\((\leftrightarrow)\) \(T \leq T'\) and every non-assignable occurrence of \texttt{a} in expressions of \texttt{ops}, \texttt{cds} and \texttt{c} is cast with \texttt{T} or any subtype of \texttt{T} declared in \texttt{cds}.

\((\leftarrow)\) (1) every expression assigned to \texttt{a}, in \texttt{ops}, \texttt{cds} and \texttt{c}, is of type \texttt{T} or any subtype of \texttt{T}; (2) every use of \texttt{a} as result argument is for a corresponding formal parameter of type \texttt{T} or any subtype of \texttt{T}.

**Law 87** (move attribute to superclass)

\[
\begin{array}{l}
\text{class } B \text{ extends } A \\
\text{ ads} \\
\text{ ops} \\
\text{ end class } C \text{ extends } B \\
\text{ pub } a : T; \text{ ads}' \\
\text{ ops'} \\
\text{ end}
\end{array}
=_{cds,c}
\begin{array}{l}
\text{class } B \text{ extends } A \\
\text{ pub } a : T; \text{ ads} \\
\text{ ops} \\
\text{ end class } C \text{ extends } B \\
\text{ ads'} \\
\text{ ops'} \\
\text{ end}
\end{array}
\]

provided

\((\rightarrow)\) The attribute name \texttt{a} is not declared by the subclasses of \texttt{B} in \texttt{cds};
APPENDIX E. LAWS OF CLASSES

\(\leftarrow\) \(D.a\), for any \(D \leq B\) and \(D \not\leq C\), does not appear in \(cds, c, ops\), or \(ops'\).

\[\square\]

E.1.3 Method Declaration

**Law 88** *(introduce method redefinition)*

\[
\begin{align*}
\text{class } B & \text{ extends } A \\
ads & \\
meth & m \doteq pc \\
ops & \\
\text{end}
\end{align*}
\]

\[
\begin{align*}
\text{class } C & \text{ extends } B \\
ads' & \\
meth & m \doteq pc \\
ops' & \\
\text{end}
\end{align*}
\]

provided

\(\rightarrow\) \(m\) is not declared in \(ops'\).

\[\square\]

**Law 89** *(move redefined method to superclass)*

\[
\begin{align*}
\text{class } B & \text{ extends } A \\
ads & \\
meth & m \doteq (pds \bullet b) \\
ops & \\
\text{end}
\end{align*}
\]

\[
\begin{align*}
\text{class } C & \text{ extends } B \\
ads' & \\
meth & m \doteq (pds \bullet b') \\
ops' & \\
\text{end}
\end{align*}
\]

\(=_{cds,c}\)

provided

\(\rightarrow\) (1) \text{super} and private attributes do not appear in \(b'\); (2) \text{super}.\(m\) does not appear in \(ops'\);
\((\to)\) \(b'\) does not contain uncast occurrences of \texttt{self} nor expressions in the form \(((C)\texttt{self}).a\) for any private attribute \(a\) in \texttt{ads}';
\n\((\leftarrow)\) \(m\) is not declared in \texttt{ops}'.

Law 90 \textit{(method elimination)}

\[
\begin{array}{|c|c|}
\hline
\text{class } C \text{ extends } D \\
\text{\hspace{1cm}} \text{ads} \\
\text{\hspace{1cm}} \text{meth } m \overset{\text{pc}}{\equiv} \text{end; ops} \\
\hline
\end{array}
=_{\text{cds},c}
\begin{array}{|c|}
\hline
\text{class } C \text{ extends } D \\
\text{\hspace{1cm}} \text{ads} \\
\text{\hspace{1cm}} \text{ops} \\
\hline
\end{array}
\begin{array}{|c|}
\hline
\text{end} \\
\hline
\end{array}
\]  

provided  
\[(\to)\) \(B.m\) does not appear in \texttt{cds}, \texttt{c} nor in \texttt{ops}, for any \(B\) such that \(B \leq C\).
\[(\leftarrow)\) \(m\) is not declared in \texttt{ops} nor in any superclass or subclass of \(C\) in \texttt{cds}.

Law 91 \textit{(move original method to superclass)}

\[
\begin{array}{|c|c|}
\hline
\text{class } B \text{ extends } A \\
\text{\hspace{1cm}} \text{ads} \\
\text{\hspace{1cm}} \text{ops} \\
\hline
\end{array}
=_{\text{cds},c}
\begin{array}{|c|c|}
\hline
\text{class } B \text{ extends } A \\
\text{\hspace{1cm}} \text{ads} \\
\text{\hspace{1cm}} \text{meth } m \overset{\text{pc}}{\equiv} \text{end} \\
\text{\hspace{1cm}} \text{ops} \\
\hline
\end{array}
\begin{array}{|c|}
\hline
\text{end} \\
\hline
\end{array}
\begin{array}{|c|c|}
\hline
\text{class } C \text{ extends } B \\
\text{\hspace{1cm}} \text{ads}' \\
\text{\hspace{1cm}} \text{meth } m \overset{\text{pc}}{\equiv} \text{end} \\
\text{\hspace{1cm}} \text{ops}' \\
\hline
\end{array}
\]  

provided  
\[(\leftrightarrow)\) (1) \texttt{super} and private attributes do not appear in \texttt{pc}; (2) \(m\) is not declared in any superclass of \(B\) in \texttt{cds};
\[(\to)\) (1) \(m\) is not declared in \texttt{ops}, and can only be declared in a class \(D\), for any \(D \leq B\) and \(D \nleq C\), if it has the same parameters as \texttt{pc}; (2) \texttt{pc} does not contain uncast occurrences of \texttt{self} nor expressions in the form \(((C)\texttt{self}).a\) for any private attribute \(a\) in \texttt{ads}';
\[(\leftarrow)\) (1) \(m\) is not declared in \texttt{ops}'; (2) \(D.m\), for any \(D \leq B\), does not appear in \texttt{cds}, \texttt{c}, \texttt{ops} or \texttt{ops}'.

\qed
E.1.4 Parameter Type

Law 92 (change value parameter type)

\[
\text{class } C \text{ extends } D \\
ads \\
\text{meth } m \equiv \\
\text{val } x : T ; pds \bullet b \\
\text{ops} \\
\text{end}
\]

\[
\text{class } C \text{ extends } D \\
ads \\
\text{meth } m \equiv \quad =_{cds,c} \\
\text{val } x : T' ; pds \bullet b \\
\text{ops} \\
\text{end}
\]

provided

\((\leftrightarrow)\) \(T \leq T'\) and every non-assignable occurrence of \(x\) in expressions of \(b\) are cast with \(T\) or any subtype of \(T\);

\((\rightarrow)\) (1) every actual parameter associated with \(x\) in \(ops\), \(cds\), and \(c\) is of type \(T\) or any subtype of it; (2) every expression assigned to \(x\) in \(b\), is of type \(T\) or any subtype of \(T\); (3) every use of \(x\) as result argument in \(b\) is for a corresponding formal parameter of type \(T\) or any subtype of \(T\).

\(\square\)

Law 93 (change result parameter type)

\[
\text{class } C \text{ extends } D \\
ads \\
\text{meth } m \equiv \\
\text{res } x : T ; pds \bullet b \\
\text{ops} \\
\text{end}
\]

\[
\text{class } C \text{ extends } D \\
ads \\
\text{meth } m \equiv \quad =_{cds,c} \\
\text{res } x : T' ; pds \bullet b \\
\text{ops} \\
\text{end}
\]

provided

\((\leftrightarrow)\) \(T \leq T'\) and every non-assignable occurrence of \(x\) in expressions of \(b\) are cast with \(T\) or any subtype of \(T\);

\((\rightarrow)\) every actual parameter associated with formal parameter \(x\) in \(ops\), \(cds\), and \(c\) is of type \(T'\) or any supertype of it;

\((\leftarrow)\) (1) every expression assigned to \(x\) in \(b\) is of type \(T\) or any subtype of \(T\); (2) every use of \(x\) as result argument in \(b\) is for a corresponding formal parameter of type \(T\) or any subtype of \(T\).

\(\square\)
E.1.5 Method Calls

**Law 94** *(eliminate super)*

Consider that $CDS$ is a set of two class declarations as follows.

```plaintext
class B extends A
  ads
  meth m = pc
  ops
end

class C extends B
  ads'
  ops'
end
```

Then we have that

$$cds CDS, C \triangleright super.m = pc$$

provided

$$(\rightarrow) \ super \ and \ the \ private \ attributes \ in \ ads \ do \ not \ appear \ in \ pc.$$  

\[ \square \]

**Law 95** *(method call elimination)*

Consider that the following class declaration

```plaintext
class C extends D
  ads
  meth m = pc
  ops
end
```

is included in $cds$ and $cds, A \triangleright le : C$. Then

$$cds, A \triangleright le.m(e) = \{ le \neq \textbf{null} \land le \neq \textbf{error} \}; \ pc[le/self](e)$$

provided

$$(\leftrightarrow) \ (1) \ m \ is \ not \ redefined \ in \ cds \ and \ pc \ does \ not \ contain \ references \ to \ super; \ (2) \ all \ attributes \ which \ appear \ in \ the \ body \ pc \ of \ m \ are \ public.$$  

\[ \square \]
E.1.6 Casts

Law 96 \textit{(eliminate cast of method call)}
If $cds, A \triangleright e : B$, $C \leq B$ and $m$ is declared in $B$ or in any of its superclasses in $cds$, then

$$cds, A \triangleright ((C)e).m(e') = \{e \text{ is } C\}; e.m(e')$$

Law 97 \textit{(introduce trivial cast in expressions)}
If $cds, A \triangleright e : C$, then $cds, A \triangleright e = (C)e$.

For simplicity, this is formalised as a law of expressions, not commands. Nevertheless, it should be considered as an abbreviation for several laws of assignments, conditionals, and method calls that deal with each possible pattern of expressions. For example, it abbreviates the following laws, all with the same antecedent as Law 97:

$$cds, A \triangleright le := e.x = le := ((C)e).x$$
$$cds, A \triangleright e'.m(e) = e'.m((C)e)$$

This is equally valid for left-expressions, which are a particular form of expression.

Law 98 \textit{(eliminate cast of expressions)}
If $cds, A \triangleright le : B$, $e : B'$, $C \leq B'$ and $B' \leq B$, then

$$cds, A \triangleright le := (C)e = \{e \text{ is } C\}; le := e$$

E.1.7 Commands and expressions

Law 99 \textit{(change variable type)}

$$cds, A \triangleright \text{var } x : T \bullet c \text{ end } = \text{var } x : T' \bullet c \text{ end}$$

provided

$(\rightarrow)$ $T \leq T'$ and every non-assignable occurrence of $x$ in expressions of $c$ is cast with $T$ or any subtype of $T$;

$(\leftarrow)$ (1) every expression assigned to $x$ in $c$ is of type $T$ or any subtype of $T$; (2) every use of $x$ as result argument in $c$ is for a corresponding formal parameter of type $T$ or any subtype of $T$.
Law 100 (change angelic variable type)

\[
\text{cds, } A \triangleright a\text{var } x : T \bullet c \text{ end } = \text{avar } x : T' \bullet c \text{ end}
\]

provided

\(\leftrightarrow\) \(T \leq T'\) and every non-assignable occurrence of \(x\) in expressions of \(c\) is cast with \(T\) or any subtype of \(T\);

\(\leftarrow\) (1) every expression assigned to \(x\) in \(c\) is of type \(T\) or any subtype of \(T\); (2) every use of \(x\) as result argument in \(c\) is for a corresponding formal parameter of type \(T\) or any subtype of \(T\).

Law 101 (is test true)

If \(N \leq_{\text{cds}} M\), then \(\text{cds, } N \triangleright \text{self is } M = \text{true}\)

Law 102 (law is test false)

If \(N \not\leq_{\text{cds}} M\) and \(M \not\leq_{\text{cds}} N\), then \(\text{cds, } N \triangleright \text{self is } M = \text{false}\)

E.2 Further object-oriented programming laws

In this section, we present additional object-oriented programming laws that deal with the expression new and with changing a superclass of a class.

E.2.1 Laws for new

Law 103 (new superclass)

\[
\begin{align*}
\text{class } A \text{ extends } C \\
\qquad \text{ads}_a \\
\qquad \text{mts}_a \\
\text{end} \\
\text{class } B \text{ extends } A \\
\text{end}
\end{align*}
\]

\[
\begin{align*}
\text{class } A \text{ extends } C \\
\qquad \text{ads}_a \\
\qquad \text{mts}_a \\
\text{end} \\
\text{class } B \text{ extends } A \\
\text{end}
\end{align*}
\]

\[
\begin{align*}
\text{cds} \bullet c
\end{align*}
\]

\[
\begin{align*}
\text{cds}' \bullet c'
\end{align*}
\]
where

\[ cds' \equiv cds[\text{new } A/\text{new } B] \]
\[ c' \equiv c[\text{new } A/\text{new } B] \]

provided

(\rightarrow) (1) \( B \) is not used in type casts or type tests in \( cds \) or \( c \) for expressions of type \( A \);
(2) ‘\text{new } B’ is assigned only to attributes or variables of type \( A \) or any supertype of \( A \).
(3) ‘\text{new } B’ is used as a value argument only in calls to methods with a corresponding formal parameter of type \( A \) or any supertype of \( A \);
(4) ‘\text{new } B’ only is assigned to a result argument of type \( A \) or any supertype of \( A \).

\[ \Box \]

Law 104 (new subclass)

\begin{align*}
\text{class } A \text{ extends } C \\
ads_a \\
mts_a \\
end
\\
\text{class } B \text{ extends } A \\
ads_b \\
mts_b \\
end
\end{align*}

\begin{align*}
\text{cds } \bullet \text{ } c
\end{align*}

\begin{align*}
\text{class } A \text{ extends } C \\
ads_a \\
mts_a \\
end
\\
\text{class } B \text{ extends } A \\
ads_b \\
mts_b \\
end
\end{align*}

\begin{align*}
\text{cds}' \bullet \text{ } c'
\end{align*}

where

\[ cds' \equiv cds[\text{new } B/\text{new } A] \]
\[ c' \equiv c[\text{new } B/\text{new } A] \]

provided

(\rightarrow) (1) Methods of \( mts_a \) are not redefined by methods of \( mts_b \);

(\rightarrow) (1) \( B \) is not used in type casts or type tests in \( cds \) or \( c \) for expressions of type \( A \);
(2) ‘new B’ is assigned only to attributes or variables of type A or any supertype of A.

(3) ‘new B’ is passed as a value argument in calls to methods with a corresponding formal parameter of type A or any supertype of A.

(4) ‘new B’ only is assigned to a result argument of type A or any supertype of A.

E.2.2 Laws for changing a superclass

Law 105  
\textit{(change superclass: from object to any class)}

\begin{align*}
\text{class } C \text{ extends object} \\
ads_c \\
mts_c \\
\text{end} &=_{cds,c} \\
\text{class } C \text{ extends } B \\
ads_c \\
mts_c \\
\text{end}
\end{align*}

provided

\text{(→)}
1. All attributes in \(ads_c\) and in subclasses of \(C\) are distinct from those declared in \(B\) and in superclasses of \(B\);
2. Methods in \(mts_c\) and in subclasses of \(C\) that have the same name must have the same parameter declaration of methods declared or inherited by \(B\);

\text{(←)}
1. \(C\) or any of its subclasses in \(cds\) is not used in type casts or tests involving any expression of type \(B\) or of any supertype of \(B\);
2. There are no assignments of the form \(le := exp\), for any \(le\) whose declared type is \(B\) or any superclass of \(B\) and any \(exp\) whose type is \(C\) or any subclass of \(C\);
3. Expressions of type \(C\) or of any subclass of \(C\) are not used as value arguments in calls with a corresponding formal parameter whose type is \(B\) or any superclass of \(B\);
4. Expressions whose declared type is \(B\) or any of its superclasses are not result arguments in calls with a corresponding formal parameter whose declared type is \(C\) or any subclass of \(C\);
5. \texttt{self.a} does not appear in \(C\), nor in any subclass of \(C\), for any public or protected attribute \(a\) of \(B\) or of any of its superclasses;
6. \texttt{le.a}, for any \(le : C\), does not appear in \(cds\) or \(c\) for public attribute \(a\) of \(B\) or of any of its superclasses;
APPENDIX E. LAWS OF CLASSES

(7) There is no \( D.m \), for any \( m \) and \( D \) such that \( m \) is declared in \( B \) or in any of its superclasses, but not in \( mts_c \), and \( D \leq C \);
(8) \texttt{super} does not appear in any method in \( mts_c \).

\[ \square \]

Law 106 \textit{(change superclass: from an empty class to immediate superclass)}

\begin{align*}
\text{class } B & \text{ extends } A \\
\text{end} & \\
\text{class } C & \text{ extends } B \\
ads_c & \\
mts_c & \\
\text{end} & =_{cds,c} \\
\text{class } B & \text{ extends } A \\
\text{end} & \\
\text{class } C & \text{ extends } A \\
ads_c & \\
mts_c & \\
\text{end}
\end{align*}

provided

\[ \to \]

(1) \( C \) or any of its subclasses in \( cds \) is not used in type casts or tests involving expressions of type \( B \);
(2) There are no assignments of the form \( le := exp \), for any \( le \) whose declared type is \( B \) or any of its superclasses and the type of \( exp \) is \( C \) or any subclass of \( C \);
(3) Expressions of type \( C \) or of any subclass of \( C \) are not used as value arguments in calls with a corresponding formal value parameter whose type is \( B \);
(4) Expressions whose declared type is \( B \) are not result arguments in calls with a corresponding formal result parameter whose declared type is \( C \) or any subclass of \( C \);

\[ \to \]

(1) Casts to class \( B \) are not applied to attributes, variables or parameters of type \( A \) to which are assigned expressions of type \( C \)

\[ \square \]

E.2.3 Class invariant

Law 107 \textit{(introduce class invariant)}
E.3. Simulation

Law 108 (private attribute-coupling invariant)

Law 109 (superclass attribute-coupling invariant)

provided

\( \text{CI} \) refers only to public and protected attributes in \( \text{adsA} \);

\( \text{cds} \) contains no subclasses of \( A \)

\( \text{cds} \) only contains subclasses of \( A \)
Appendix F

Proofs of Laws of Commands

Lemma 11

\[ [\Gamma, N \triangleright \text{skip} : \text{com}] \eta \psi \]
\[ = [\Gamma, N \triangleright : \text{true, true} : \text{com}] \eta \psi \]
\[ = \text{true} \land \text{true} \Rightarrow \psi \]
\[ = \psi \]

[by definition of \text{skip}]  
[by the semantics of specification statement]  
[by predicate calculus]

Lemma 12

\[ [\Gamma, N \triangleright \text{abort} : \text{com}] \eta \psi \]
\[ = [\Gamma, N \triangleright x : \text{false, true} : \text{com}] \eta \psi \]
\[ = \text{false} \land (\forall x : T \bullet \text{true} \Rightarrow \psi) \]
\[ = \text{false} \]

[by definition of \text{abort}]  
[by the semantics of specification statement]  
[by predicate calculus]

Lemma 13 If \( x \) is not free in \( c \) and \( \psi \), then \( x \) is not free in

\[ [\Gamma, N, N \triangleright c : \text{com}] \eta \psi \]

Proof Straightforward induction.

Lemma 14 For a typing environment \( \Gamma \), a class \( N \), a proper environment \( \eta \), a command \( c \), a predicate \( \psi \), and \( x \) a variable of type \( T \) not free in \( c \) and \( \psi \), if \( y \) is not free in \( c \), then

\[ [\Gamma; x : T, N \triangleright c : \text{com}] \eta \psi = [\Gamma, N \triangleright c : \text{com}] \eta \psi \]
APPENDIX F. PROOFS OF LAWS OF COMMANDS

Proof

\[
\begin{align*}
&[\Gamma; x : T, N \triangleright c : \text{com}] \eta \psi \\
&= [\Gamma; x : T, N \triangleright c[y/x][x/y] : \text{com}] \eta \psi & \text{[by } y \text{ not free in } c]\end{align*}
\]

\[
\begin{align*}
&= ([\Gamma, N \triangleright c[y/x] : \text{com}] \eta \psi[x/y])[x/y] & \text{[by Lemma 13-22]} \\
&= ([\Gamma, N \triangleright c : \text{com}] \eta \psi[x/y])[x/y] & \text{[by } y \text{ not free in } c]\end{align*}
\]

\[
\begin{align*}
&= ([\Gamma, N \triangleright c : \text{com}] \eta \psi)[x/y] & \text{[by } x \text{ not free in } \psi]\end{align*}
\]

\[
\begin{align*}
&= ([\Gamma, N \triangleright c : \text{com}] \eta \psi)[x/y] & \text{[by Lemma 13]}
\end{align*}
\]

\[
\begin{align*}
&= ([\Gamma, N \triangleright le : \text{com}] \eta \psi)[x/y] & \text{[by the semantics of assignment]} \\
&= le \neq error \land \psi[le/le] & \text{[by the semantics of assignment]} \\
&= \psi & \text{[by the hypothesis and a property of substitution]} \\
&= [\Gamma, N \triangleright \text{skip} : \text{com}] \eta \psi & \text{[by Lemma 11]}
\end{align*}
\]

\[
\begin{align*}
&= ([\Gamma, N \triangleright le := e, le_1 := e_1 : \text{com}] \eta \psi)[x/y] & \text{[by the semantics of assignment]} \\
&= e \neq error \land le_1 \neq error \land \psi[e, le_1/le, le_1] & \text{[by the semantics of assignment]} \\
&= e \neq error \land \psi[e/le] & \text{[by the hypothesis and a property of substitution]} \\
&= [\Gamma, N \triangleright le := e : \text{com}] \eta \psi & \text{[by the semantics of assignment]}
\end{align*}
\]

\[
\begin{align*}
&= ([\Gamma, N \triangleright le := e_i : \text{com}] \eta \psi)[x/y] & \text{[by the semantics of assignment]} \\
&= le_i \neq error \land \psi[e_i/le_i] & \text{[by the semantics of assignment]} \\
&= le_{\pi(i)} \neq error \land \psi[e_{\pi(i)}/le_{\pi(i)}] & \text{[by a property a substitution]}
\end{align*}
\]

\[
\begin{align*}
&= ([\Gamma, N \triangleright le := e_i : \text{com}] \eta \psi)[x/y] & \text{[by the semantics of assignment]} \\
&= le_i \neq error \land \psi[e_i/le_i] & \text{[by the semantics of assignment]} \\
&= le_{\pi(i)} \neq error \land \psi[e_{\pi(i)}/le_{\pi(i)}] & \text{[by a property a substitution]}
\end{align*}
\]
F.1. PROOFS OF LAWS

\[ \Gamma, N \triangleright le_{\pi(i)} := c_{\pi(i)} : \text{com}\eta \psi \]  
[by the semantics of assignment]

\[ \blacksquare \]

F.1.2 Conditional Proof F.1.4 Law (if symmetry)

\[ [\Gamma, N \triangleright i \cdot \psi_i \rightarrow c_i \text{ fi : com}\eta \psi \] 
\[ = (\forall i \cdot \psi_i) \land (\forall i \cdot \psi_i \rightarrow [\Gamma, N \triangleright c_i : \text{com}\eta \psi) \]  
[by the semantics of alternation]

\[ = (\forall i \cdot \psi_{\pi(i)}) \land (\forall i \cdot \psi_{\pi(i)} \rightarrow [\Gamma, N \triangleright c_{\pi(i)} : \text{com}\eta \psi) \]  
[by commutativity of disjunction and conjunction]

\[ = [\Gamma, N \triangleright i \cdot \psi_{\pi(i)} \rightarrow c_{\pi(i)} \text{ fi : com}\eta \psi \]  
[by the semantics of alternation]

\[ \blacksquare \]

Proof F.1.5 Law (if true guard)

\[ [\Gamma, N \triangleright \text{ if true } \rightarrow c \text{ fi : com}\eta \psi \] 
\[ = \text{true} \land \text{true} \rightarrow [\Gamma, N \triangleright c : \text{com}\eta \psi \]  
[by the semantics of alternation]

\[ = [\Gamma, N \triangleright c : \text{com}\eta \psi \]  
[by predicate calculus]

\[ \blacksquare \]

Proof F.1.6 Law (if false unity)

\[ [\Gamma, N \triangleright \text{ if false } \rightarrow c \text{ fi : com}\eta \psi \] 
\[ = (\forall i \cdot \psi_i \lor \text{false}) \land (\text{false} \rightarrow [\Gamma, N \triangleright c : \text{com}\eta \psi) \land (\forall i \cdot \psi_i \rightarrow [\Gamma, N \triangleright c_i : \text{com}\eta \psi) \]  
[by the semantics of alternation]

\[ = (\forall i \cdot \psi_i) \land (\forall i \cdot \psi_i \rightarrow [\Gamma, N \triangleright c_i : \text{com}\eta \psi) \]  
[by predicate calculus]

\[ = [\Gamma, N \triangleright \text{ if } \psi_i \rightarrow c_i \text{ fi : com}\eta \psi \]  
[by the semantics of alternation]

\[ \blacksquare \]

Proof F.1.7 Law (if abort unity)

\[ [\Gamma, N \triangleright \text{ if } () \text{ fi : com}\eta \psi \]
Proof F.1.8 Law (if - \lor distrib1)

\[
\begin{split}
\Gamma, N \triangleright \psi_1 \rightarrow c \mid \psi_2 \rightarrow c \mid (i \bullet \psi_1 \rightarrow c_i) \triangleright \com \eta \psi \\
= (\lor i \bullet \psi_1 \lor \psi_1 \lor \psi_2) \land (\psi_1 \Rightarrow [\Gamma, N \triangleright c : \com \eta \psi]) \\
\wedge (\psi_2 \Rightarrow [\Gamma, N \triangleright c : \com \eta \psi]) \\
\quad \wedge (i \bullet \psi_1 \Rightarrow [\Gamma, N \triangleright c_i : \com \eta \psi]) \\
\end{split}
\]

(by the semantics of alternation)

\[
\begin{split}
\Gamma, N \triangleright if (\psi_1 \lor \psi_2) \rightarrow c \mid (i \bullet \psi_1 \rightarrow c_i) : \com \eta \psi \\
= [\Gamma, N \triangleright if (\psi_1 \lor \psi_2) \rightarrow c \mid (i \bullet \psi_1 \rightarrow c_i) : \com \eta \psi] \\
\end{split}
\]

(by the semantics of alternation)

\[
\begin{split}
\Gamma, N \triangleright if \psi \rightarrow (if \psi \rightarrow c \mid (i \bullet \psi_1 \rightarrow c_i)) \triangleright \com \eta \psi \\
= (\lor j \bullet \psi_j \lor \psi) \land (\psi \Rightarrow (if \psi \rightarrow c (i \bullet \psi_1 \rightarrow c_i)) \land (\psi_j \Rightarrow [\Gamma, N \triangleright c_j : \com \eta \psi]) \\
\end{split}
\]

(by the semantics of alternation)

\[
\begin{split}
\Gamma, N \triangleright if \psi \rightarrow (if \psi \rightarrow c \mid (i \bullet \psi_1 \rightarrow c_i)) \triangleright \com \eta \psi \\
= (\lor j \bullet \psi_j \lor \psi) \land (\psi \Rightarrow (if \psi \rightarrow c (i \bullet \psi_1 \rightarrow c_i)) \land (\psi_j \Rightarrow [\Gamma, N \triangleright c_j : \com \eta \psi]) \\
\end{split}
\]

(by the semantics of alternation)

(by predicate calculus)

\[
\begin{split}
\Gamma, N \triangleright if \psi \rightarrow (if \psi \rightarrow c \mid (i \bullet \psi_1 \rightarrow c_i)) \triangleright \com \eta \psi \\
= (\lor j \bullet \psi_j \lor \psi) \land (\psi \Rightarrow (if \psi \rightarrow c (i \bullet \psi_1 \rightarrow c_i)) \land (\psi_j \Rightarrow [\Gamma, N \triangleright c_j : \com \eta \psi]) \\
\end{split}
\]

(by the semantics of alternation)

(by predicate calculus)

\[
\begin{split}
\Gamma, N \triangleright if \psi \rightarrow (if \psi \rightarrow c \mid (i \bullet \psi_1 \rightarrow c_i)) \triangleright \com \eta \psi \\
= (\lor j \bullet \psi_j \lor \psi) \land (\psi \Rightarrow (if \psi \rightarrow c (i \bullet \psi_1 \rightarrow c_i)) \land (\psi_j \Rightarrow [\Gamma, N \triangleright c_j : \com \eta \psi]) \\
\end{split}
\]

(by the semantics of alternation)

(by predicate calculus)

\[
\begin{split}
\Gamma, N \triangleright if \psi \rightarrow (if \psi \rightarrow c \mid (i \bullet \psi_1 \rightarrow c_i)) \triangleright \com \eta \psi \\
= (\lor j \bullet \psi_j \lor \psi) \land (\psi \Rightarrow (if \psi \rightarrow c (i \bullet \psi_1 \rightarrow c_i)) \land (\psi_j \Rightarrow [\Gamma, N \triangleright c_j : \com \eta \psi]) \\
\end{split}
\]

(by the semantics of alternation)

(by predicate calculus)

\[
\begin{split}
\Gamma, N \triangleright if \psi \rightarrow (if \psi \rightarrow c \mid (i \bullet \psi_1 \rightarrow c_i)) \triangleright \com \eta \psi \\
= (\lor j \bullet \psi_j \lor \psi) \land (\psi \Rightarrow (if \psi \rightarrow c (i \bullet \psi_1 \rightarrow c_i)) \land (\psi_j \Rightarrow [\Gamma, N \triangleright c_j : \com \eta \psi]) \\
\end{split}
\]

(by the semantics of alternation)

(by predicate calculus)

\[
\begin{split}
\Gamma, N \triangleright if \psi \rightarrow (if \psi \rightarrow c \mid (i \bullet \psi_1 \rightarrow c_i)) \triangleright \com \eta \psi \\
= (\lor j \bullet \psi_j \lor \psi) \land (\psi \Rightarrow (if \psi \rightarrow c (i \bullet \psi_1 \rightarrow c_i)) \land (\psi_j \Rightarrow [\Gamma, N \triangleright c_j : \com \eta \psi]) \\
\end{split}
\]

(by the semantics of alternation)

(by predicate calculus)

\[
\begin{split}
\Gamma, N \triangleright if \psi \rightarrow (if \psi \rightarrow c \mid (i \bullet \psi_1 \rightarrow c_i)) \triangleright \com \eta \psi \\
= (\lor j \bullet \psi_j \lor \psi) \land (\psi \Rightarrow (if \psi \rightarrow c (i \bullet \psi_1 \rightarrow c_i)) \land (\psi_j \Rightarrow [\Gamma, N \triangleright c_j : \com \eta \psi]) \\
\end{split}
\]

(by the semantics of alternation)

(by predicate calculus)

\[
\begin{split}
\Gamma, N \triangleright if \psi \rightarrow (if \psi \rightarrow c \mid (i \bullet \psi_1 \rightarrow c_i)) \triangleright \com \eta \psi \\
= (\lor j \bullet \psi_j \lor \psi) \land (\psi \Rightarrow (if \psi \rightarrow c (i \bullet \psi_1 \rightarrow c_i)) \land (\psi_j \Rightarrow [\Gamma, N \triangleright c_j : \com \eta \psi]) \\
\end{split}
\]

(by the semantics of alternation)

(by predicate calculus)
(\neg \psi \lor [\Gamma, N \triangleright c : \text{com}\eta \psi] \land ([\psi \rightarrow c_1] \rightarrow \neg \psi_1 \rightarrow (\text{if } \psi_2 \rightarrow c_1 \rightarrow \neg \psi_2 \rightarrow c_2 \rightarrow \text{fi}) : \text{com}\eta \psi)

= (\psi_1 \lor \neg \psi_1) \land (\psi_1 \rightarrow [\Gamma, N \triangleright c_1 : \text{com}\eta \psi] \land
(\neg \psi_1 \rightarrow [\Gamma, N \triangleright \text{if } \psi_2 \rightarrow c_1 \rightarrow \neg \psi_2 \rightarrow c_2 \rightarrow \text{fi} : \text{com}\eta \psi])

= (\psi_1 \rightarrow [\Gamma, N \triangleright c_1 : \text{com}\eta \psi] \land (\neg \psi_1 \rightarrow ((\psi_2 \lor \neg \psi_2) \land (\psi_2 \rightarrow [\Gamma, N \triangleright c_1 : \text{com}\eta \psi] \land
(\neg \psi_2 \rightarrow [\Gamma, N \triangleright c_2 : \text{com}\eta \psi])))

= (\psi_1 \rightarrow [\Gamma, N \triangleright c_1 : \text{com}\eta \psi] \land (\neg \psi_1 \rightarrow ((\neg \psi_2 \lor \neg \psi_2 \lor [\Gamma, N \triangleright c_1 : \text{com}\eta \psi] \land
(\neg \psi_2 \lor [\Gamma, N \triangleright c_2 : \text{com}\eta \psi]))

= (\neg \psi_1 \lor [\Gamma, N \triangleright c_1 : \text{com}\eta \psi] \land (\psi_1 \lor ((\neg \psi_1 \lor [\Gamma, N \triangleright c_1 : \text{com}\eta \psi] \land
(\psi_2 \lor [\Gamma, N \triangleright c_2 : \text{com}\eta \psi]))

= ([\Gamma, N \triangleright c_1 : \text{com}\eta \psi(\neg \psi_1 \lor (\psi_1 \land \neg \psi_2))) \land (\psi_1 \lor \psi_2 \lor [\Gamma, N \triangleright c_2 : \text{com}\eta \psi])

\[ \square \]
APPENDIX F.  PROOFS OF LAWS OF COMMANDS

\[ ([\Gamma, N \triangleright c_1 : \text{com}] \eta \psi) \lor ((\neg \psi_1 \land \psi_1) \lor (\neg \psi_1 \land \neg \psi_2)) \land (\psi_1 \lor \psi_2 \lor [\Gamma, N \triangleright c_2 : \text{com}] \eta \psi) \]

[by predicate calculus]

\[ ([\Gamma, N \triangleright c_1 : \text{com}] \eta \psi) \lor (\neg (\psi_1 \lor \psi_2)) \land (\psi_1 \lor \psi_2 \lor [\Gamma, N \triangleright c_2 : \text{com}] \eta \psi) \]

[by predicate calculus]

\[ ([\psi_1 \lor \psi_2) \Rightarrow [\Gamma, N \triangleright c_1 : \text{com}] \eta \psi) \land (\neg (\psi_1 \lor \psi_2) \Rightarrow [\Gamma, N \triangleright c_2 : \text{com}] \eta \psi) \]

[by predicate calculus]

\[ [\Gamma, N \triangleright \text{if} (\psi_1 \lor \psi_2) \rightarrow c_1 ]\neg (\psi_1 \lor \psi_2) \rightarrow c_2 \text{ fi : com}] \eta \psi \]

[by the semantics of alternation]

\[ \Box \]

**Proof F.1.11 Law (if- \land distrib)**

\[ [\Gamma, N \triangleright \text{if} \psi_1 \rightarrow (\text{if} \psi_2 \rightarrow c_1] \neg \psi_2 \rightarrow c_2 \text{ fi}) ] \neg \psi_1 \rightarrow c_2 \text{ fi : com}] \eta \psi \]

\[ (\neg \psi_1 \Rightarrow [\Gamma, N \triangleright \text{if} \psi_2 \rightarrow c_1] \neg \psi_2 \rightarrow c_2 \text{ com}] \eta \psi \]

[by the semantics of alternation]

\[ (\neg \psi_1 \land (\neg \psi_2 \lor [\Gamma, N \triangleright c_1 : \text{com}] \eta \psi) \land (\psi_2 \lor [\Gamma, N \triangleright c_2 : \text{com}] \eta \psi)) \land \]

[by predicate calculus]

\[ (\psi_1 \lor [\Gamma, N \triangleright c_2 : \text{com}] \eta \psi) \]

[by predicate calculus]

\[ (\neg \psi_1 \lor (\neg \psi_2 \lor [\Gamma, N \triangleright c_1 : \text{com}] \eta \psi) \land (\neg \psi_1 \lor \psi_2 \lor [\Gamma, N \triangleright c_2 : \text{com}] \eta \psi)) \land \]

[by predicate calculus]

\[ (\psi_1 \lor [\Gamma, N \triangleright c_2 : \text{com}] \eta \psi) \]

[by predicate calculus]

\[ (\neg (\psi_1 \land \psi_2) \lor [\Gamma, N \triangleright c_1 : \text{com}] \eta \psi) \land (\neg (\psi_1 \land \psi_2) \lor [\Gamma, N \triangleright c_2 : \text{com}] \eta \psi) \]

[by predicate calculus]

\[ (\neg (\psi_1 \land \psi_2) \lor [\Gamma, N \triangleright c_1 : \text{com}] \eta \psi) \land (\neg (\psi_1 \land \psi_2) \lor [\Gamma, N \triangleright c_2 : \text{com}] \eta \psi) \]

[by predicate calculus]

\[ [\Gamma, N \triangleright \text{if} (\psi_1 \land \psi_2) \rightarrow c_1] \neg (\psi_1 \land \psi_2) \rightarrow c_2 \text{ fi : com}] \eta \psi \]

[by the semantics of alternation]
Proof F.1.12 Law \( \text{if gc intro} \)

\[
\begin{align*}
\Gamma, N \triangleright \text{if } [i \cdot \psi_i \rightarrow c_i \text{ fi : com}] \eta \psi \\
= (& i \cdot \psi_i) \land (\land i \cdot \psi_i \Rightarrow [\Gamma, N \triangleright c_i : \text{com}] \eta \psi) \\
\Rightarrow (\lor i \cdot \psi_i \land (\land i \cdot \psi_i \Rightarrow [\Gamma, N \triangleright c_i : \text{com}] \eta \psi) \\
= [\Gamma, N \triangleright \text{if } ([i \cdot \psi_i \rightarrow c_i] \lor \psi \rightarrow c \text{ fi : com}] \eta \psi)
\end{align*}
\]
[by the semantics of alternation]
[by the hypothesis]
[by the semantics of alternation]

Proof F.1.13 Law \( \text{if gc elim} \)

\[
\begin{align*}
\Gamma, N \triangleright \text{if } \psi \rightarrow c \lor ([i \cdot \psi_i \rightarrow c_i] \lor \psi) \\
= (\psi \Rightarrow [\Gamma, N \triangleright c : \text{com}] \eta \psi) \\
\land (\lor i \cdot \psi_i \lor (\land i \cdot \psi_i \Rightarrow [\Gamma, N \triangleright c_i : \text{com}] \eta \psi)) \\
\Rightarrow [\Gamma, N \triangleright \text{if } ([i \cdot \psi_i \rightarrow c_i] \lor \psi \rightarrow c \text{ fi : com}] \eta \psi)
\end{align*}
\]
[by the semantics of alternation]
[by the hypothesis]
[by the semantics of alternation]

Proof F.1.14 Law \( \text{if true guard ref} \)

\[
\begin{align*}
\Gamma, N \triangleright \text{if } \text{true} \rightarrow c \lor ([i \cdot \psi_i \rightarrow c_i] \lor \psi) \\
= (\text{true} \Rightarrow [\Gamma, N \triangleright c : \text{com}] \eta \psi) \\
\land (\lor i \cdot \psi_i \lor (\land i \cdot \psi_i \Rightarrow [\Gamma, N \triangleright c_i : \text{com}] \eta \psi)) \\
\Rightarrow [\Gamma, N \triangleright c : \text{com}] \eta \psi
\end{align*}
\]
[by predicate calculus]
[by predicate calculus]

F.1.3 Recursion

Proof F.1.15 Law \( \text{rec fixed point} \)

\[
\begin{align*}
\Gamma, N \triangleright \text{rec } \text{X} \cdot F(\text{X}) : \text{com} \eta \psi
\end{align*}
\]
\[ = [\Gamma, N \triangleright \mu_X \bullet F(X) : \text{com}] \eta \psi \quad \text{[by the semantics of recursion]} \]
\[ = [\Gamma, N \triangleright F(\mu_X \bullet F(X)) : \text{com}] \eta \psi \quad \text{[by unfolding least fixpoint]} \]
\[ = [\Gamma, N \triangleright F(\text{rec}X \bullet F(X)) : \text{com}] \eta \psi \quad \text{[by the semantics of recursion]} \]

\section*{F.1.4 Sequential Composition}

\textbf{Proof F.1.16 Law \langle ; \ -\text{skip \ unit} \rangle}

\[ [\Gamma, N \triangleright \text{skip}; c : \text{com}] \eta \psi \]
\[ = [\Gamma, N \triangleright \text{skip} : \text{com}] \eta ([\Gamma, N \triangleright c : \text{com}] \eta \psi) \quad \text{[by the semantics of sequential composition]} \]
\[ = [\Gamma, N \triangleright : [\text{true, true}] : \text{com}] \eta ([\Gamma, N \triangleright c : \text{com}] \eta \psi) \quad \text{[by the definition of \text{skip}]} \]
\[ = [\Gamma, N \triangleright c : \text{com}] \eta (\text{true} \land \text{true} \Rightarrow \psi) \quad \text{[by the semantics of spec. statement and predicate calculus]} \]
\[ = [\Gamma, N \triangleright c : \text{com}] \eta ([\Gamma, N \triangleright : [\text{true, true}] : \text{com}] \eta \psi) \quad \text{[by predicate calculus]} \]
\[ = [\Gamma, N \triangleright c : \text{com}] \eta (\text{true, true}) \Rightarrow \psi \quad \text{[by the semantics of specification statement]} \]
\[ = [\Gamma, N \triangleright c : \text{com}] \eta (\text{false} \land \text{true} \Rightarrow \psi) \quad \text{[by the definition of \text{skip}]} \]
\[ = [\Gamma, N \triangleright c; \text{skip} : \text{com}] \eta \psi \quad \text{[by the semantics of sequential composition]} \]

\textbf{Proof F.1.17 Law \langle ; \ -\text{abort \ left \ zero} \rangle}

\[ [\Gamma, N \triangleright \text{abort}; c : \text{com}] \eta \psi \]
\[ = [\Gamma, N \triangleright \text{abort} : \text{com}] \eta ([\Gamma, N \triangleright c : \text{com}] \eta \psi) \quad \text{[by the semantics of sequential composition]} \]
\[ = [\Gamma, N \triangleright x : [\text{false, true}] : \text{com}] \eta ([\Gamma, N \triangleright c : \text{com}] \eta \psi) \quad \text{[by the definition of \text{abort}]} \]
\[ = \text{false} \land \text{true} \Rightarrow ([\Gamma, N \triangleright c : \text{com}] \eta \psi) \quad \text{[by the semantics of specification statement]} \]
\[ = \text{false} \quad \text{[by predicate calculus]} \]
\[ = \forall x : T \bullet \text{false} \land \text{true} \Rightarrow \psi \quad \text{[by predicate calculus]} \]
\[ = [\Gamma, N \triangleright x : [\text{false, true}] : \text{com}] \eta \psi \quad \text{[by the semantics of specification statement]} \]
\[ = [\Gamma, N \triangleright \text{abort} : \text{com}] \eta \psi \quad \text{[by the definition of \text{abort}]} \]
Proof F.1.18 Law $\langle ; \neg\text{miracle left zero} \rangle$

$$
\begin{align*}
[\Gamma, N \triangleright \text{miracle}; \ c : \text{com}] \eta \psi
&= [\Gamma, N \triangleright \text{miracle} : \text{com}] \eta ([\Gamma, N \triangleright c : \text{com}] \eta \psi) \\
&= [\Gamma, N \triangleright x : \text{true, false} : \text{com}] \eta ([\Gamma, N \triangleright c : \text{com}] \eta \psi) \\
&= \text{true} \land \text{false} \Rightarrow (\forall x : T \bullet [\Gamma, N \triangleright c : \text{com}] \eta \psi) \\
&= \text{true} \land \text{false} \\
&= \text{true} \land \text{false} \Rightarrow \psi \\
&= [\Gamma, N \triangleright x : \text{true, false} : \text{com}] \eta \psi \\
&= [\Gamma, N \triangleright \text{miracle} : \text{com}] \eta \psi
\end{align*}
$$

[by the semantics of sequential composition]

[by the definition of miracle]

[by the semantics of specification statement]

[by predicate calculus]

[by predicate calculus]

[by predicate calculus]

[by the definition of miracle]

\qedhere

Proof F.1.19 Law $\langle ; \text{assoc} \rangle$

$$
\begin{align*}
[\Gamma, N \triangleright (c_1 ; c_2); c_3 : \text{com}] \eta \psi
&= [\Gamma, N \triangleright c_1; c_2 : \text{com}] \eta ([\Gamma, N \triangleright c_3 : \text{com}] \eta \psi) \\
&= ([\Gamma, N \triangleright c_1 : \text{com}] \eta ([\Gamma, N \triangleright c_2 : \text{com}] \eta \psi)) ([\Gamma, N \triangleright c_3 : \text{com}] \eta \psi) \\
&= [\Gamma, N \triangleright c_1 : \text{com}] \eta ([\Gamma, N \triangleright c_2 ; c_3 : \text{com}] \eta \psi) \\
&= [\Gamma, N \triangleright c_1 ; (c_2 ; c_3) : \text{com}] \eta \psi
\end{align*}
$$

[by the semantics of sequential composition]

[by associativity]

[by the semantics of sequential composition]

[by the semantics of sequential composition]

\qedhere

Proof F.1.20 Law $\langle ; \neg\text{if left dist} \rangle$

$$
\begin{align*}
[\Gamma, N \triangleright \text{if} \ [i \bullet \psi_i \rightarrow c_i \ fi; \ c : \text{com}] \eta \psi
&= [\Gamma, N \triangleright \text{if} \ [i \bullet \psi_i \rightarrow c_i \ fi : \text{com}] \eta ([\Gamma, N \triangleright c : \text{com}] \eta \psi) \\
&= (\forall i \bullet \psi_i) \land (\forall i \bullet \psi_i \Rightarrow [\Gamma, N \triangleright c_i : \text{com}] \eta ([\Gamma, N \triangleright c : \text{com}] \eta \psi)) \\
&= (\forall i \bullet \psi_i) \land (\forall i \bullet \psi_i \Rightarrow [\Gamma, N \triangleright c_i ; c : \text{com}] \eta \psi)
\end{align*}
$$

[by the semantics of sequential composition]

[by the semantics of alternation]

[by the semantics of sequential composition]
APPENDIX F. PROOFS OF LAWS OF COMMANDS

\[ \begin{align*}
= \Gamma, N \triangleright \text{if } \downarrow [i \bullet \psi_i \rightarrow c_i : \text{com}] \eta \psi & \quad [\text{by the semantics of alternation}] \\
\square
\end{align*} \]

Proof F.1.21 Law \( \langle ; \text{ \text{-if \ selection} \rangle} \)

\[ \begin{align*}
[\Gamma, N \triangleright [\psi_j] ; \text{if } \downarrow [i \bullet \psi_i \rightarrow c_i : \text{com}] \eta \psi & = \Gamma, N \triangleright [\psi_j] : \text{com} \eta \Gamma, N \triangleright \text{if } \downarrow [i \bullet \psi_i \rightarrow c_i : \text{com}] \eta \psi & [\text{by the semantics of sequential composition]} \\
& = \Gamma, N \triangleright : \text{true, } \psi_j : \text{com} \eta \Gamma, N \triangleright \text{if } \downarrow [i \bullet \psi_i \rightarrow c_i : \text{com}] \eta \psi & [\text{definition of coercion}] \\
& = \Gamma, N \triangleright : \text{true, } \psi_j : \text{com} \eta \Gamma, N \triangleright \text{if } \downarrow c_i : \text{com} \eta \psi & [\text{by the semantics of alternation}] \\
& = \psi_j \Rightarrow (\lor i \bullet \psi_i) \wedge \psi_i \Rightarrow \Gamma, N \triangleright c_i : \text{com} \eta \psi & [\text{by the semantics of specification statement}] \\
& = \psi_j \Rightarrow \psi_i \Rightarrow \Gamma, N \triangleright c_i : \text{com} \eta \psi & [\text{by predicate calculus}] \\
& = (\psi_j \wedge \psi_i) \Rightarrow \Gamma, N \triangleright c_i : \text{com} \eta \psi & [\text{by predicate calculus}] \\
& = \psi_j \Rightarrow \Gamma, N \triangleright c_j : \text{com} \eta \psi & [\text{by the hypothesis}] \\
& = \Gamma, N \triangleright : \text{true, } \psi_j : \text{com} \eta \Gamma, N \triangleright c_j : \text{com} \eta \psi & [\text{by the semantics of specification statement}] \\
& = \Gamma, N \triangleright [\psi_j] : \text{com} \eta \Gamma, N \triangleright c_j : \text{com} \eta \psi & [\text{definition of coercion}] \\
& = \Gamma, N \triangleright [\psi_j] ; c_j : \text{com} \eta \psi & [\text{by the semantics of sequential composition}] \\
\square
\end{align*} \]

Proof F.1.22 Law \( \langle ; \text{ \text{-:= \ combination} \rangle} \)

\[ \begin{align*}
[\Gamma, N \triangleright (\text{le} := e_1; \text{le} := e_2) : \text{com} \eta \psi & = \Gamma, N \triangleright \text{le} := e_1 : \text{com} \eta \Gamma, N \triangleright \text{le} := e_2 : \text{com} \eta \psi & [\text{by the semantics of sequential composition}] \\
& = \Gamma, N \triangleright \text{le} := e_1 : \text{com} \eta (e_2 \not= \text{error} \land \psi[e_2/\text{le}]) & [\text{by the semantics of assignment}] \\
& = e_1 \not= \text{error} \land (e_2 \not= \text{error} \land \psi[e_2/\text{le}])[e_1/\text{le}] & [\text{by the semantics of assignment}] \\
& = e_1 \not= \text{error} \land e_2 \not= \text{error} \land (\psi[e_2/\text{le}])[e_1/\text{le}] & [\text{by predicate calculus}] \\
& = \Gamma, N \triangleright \text{le} := e_2[e_1/\text{le}] : \text{com} \eta \psi & [\text{by the semantics of assignment}] \\
\square
\end{align*} \]

Proof F.1.23 Law \( \langle := \text{ \text{-< \left \rangle right dist} \rangle} \)

\[ \begin{align*}
\]
F.1. PROOFS OF LAWS

[Γ, N ⊢ le := e; if [i • ψi → ci] fi : com]η ψ


[by the semantics of sequential composition]

= [Γ, N ⊢ le := e : com]η ((∀ i • ψi) ∧ (∀ i • ψi) ⇒ [Γ, N ⊢ ci : com]η ψ))

[by the semantics of alternation]

= e ≠ error ∧ (∀ i • ψi) ∧ (∀ i • ψi) ⇒ [Γ, N ⊢ ci : com]η ψ) [e/le]

[by the semantics of assignment]

= e ≠ error ∧ (∀ i • ψi)[e/le] ∧ (∀ i • ψi) ⇒ [Γ, N ⊢ ci : com]η ψ)[e/le] [le is not free in ψi]

= (∀ i • ψi[e/le]) ∧ (∀ i • ψi[e/le]) ⇒ [Γ, N ⊢ le := e : com]η ([Γ, N ⊢ ci : com]η ψ)

[by the semantics of assignment]

= (∀ i • ψi[e/le]) ∧ (∀ i • ψi[e/le]) ⇒ [Γ, N ⊢ (le := e; ci) : com]η ψ

[by the semantics of sequential composition]

= [Γ, N ⊢ if [i • ψi[e/le] → (le := e; ci) fi : com]η ψ [by the semantics of alternation]

F.1.5 Local Variable Block

Proof F.1.24 Law \texttt{(var symmetry)}

[Γ, N ⊢ var \ x_1 : T_1 • (var \ x_2 : T_2 • c \ end) \ end : com]η ψ

= ∀ x_1 : T_1 • [Γ; x_1 : T_1, N ⊢ var \ x_2 : T_2 • c \ end : com]η ψ

[by the semantics of local blocks]

= ∀ x_1 : T_1 • ∀ x_2 : T_2 • [Γ; x_1, x_2 : T_1, T_2, N ⊢ c : com]η ψ

[by the semantics of local blocks]

= ∀ x_2 : T_2 • ∀ x_1 : T_1 • [Γ; x_1, x_2 : T_1, T_2, N ⊢ c : com]η ψ

[by predicate calculus]

= ∀ x_2 : T_2 • [Γ; x_2 : T_2, N ⊢ var \ x_1 : T_1 • c \ end : com]η ψ

[by the semantics of local blocks]

= [Γ, N ⊢ var \ x_2 : T_2 • (var \ x_1 : T_1 • c \ end) \ end : com]η ψ

[by the semantics of local blocks]

Proof F.1.25 Law \texttt{(var elim)}

[Γ, N ⊢ var \ x : T • c \ end : com]η ψ

= ∀ x : T • [Γ; x : T, N ⊢ c : com]η ψ

[by the semantics of local blocks]

= ∀ x : T • [Γ, N ⊢ c : com]η ψ

[by x not free in c]

= [Γ, N ⊢ c : com]η ψ

[by x not free in c and ψ and predicate calculus]
Proof F.1.26 Law \( \langle \text{rename} \rangle \)

\[
[\Gamma, N \triangleright \text{var } x_1 : T \bullet c \triangleright \text{com}] \eta \psi
\]

\[
\Rightarrow \forall x_1 : T \bullet [\Gamma; x_1 : T, N \triangleright c \triangleright \text{com}] \eta \psi
\]

\[\text{[by the semantics of local blocks]}\]

\[
\Rightarrow \forall x_2 : T \bullet ([\Gamma; x_1 : T, N \triangleright c \triangleright \text{com}] \eta \psi)[x_2/x_1]
\]

\[\text{[by Lemma \textbf{[13]} and predicate calculus]}\]

\[
\Rightarrow \forall x_2 : T \bullet ([\Gamma; x_1 : T, N \triangleright c \triangleright \text{com}] \eta)[x_2/x_1]\psi[x_2/x_1]
\]

\[\text{[by Lemma \textbf{[14]}]}\]

\[
\Rightarrow [\Gamma, N \triangleright \text{var } x_2 : T \bullet c[x_2/x_1] \triangleright \text{com}] \eta \psi
\]

\[\text{[by the semantics of local blocks]}\]

\[
\square
\]

Proof F.1.27 Law \( \langle \text{var } \cdot \text{if dist} \rangle \)

\[
[\Gamma, N \triangleright \text{if } \{ i \bullet \psi_1 \rightarrow (\text{var } x : T \bullet c_1 \triangleright \text{end}) \triangleright \text{fi } \triangleright \text{com}] \eta \psi
\]

\[
= (\forall i \bullet \psi_1) \land (\forall i \bullet \psi_1 \Rightarrow [\Gamma, N \triangleright \text{var } x : T \bullet c_1 \triangleright \text{com}] \eta \psi)
\]

\[\text{[by the semantics of alternation]}\]

\[
= (\forall i \bullet \psi_1) \land (\forall i \bullet \psi_1 \Rightarrow \forall x : T \bullet [\Gamma; x : T, N \triangleright c_1 \triangleright \text{com}] \eta \psi)
\]

\[\text{[by the semantics of local blocks]}\]

\[
= \forall x : T \bullet (\forall i \bullet \psi_1 \Rightarrow [\Gamma; x : T, N \triangleright c_1 \triangleright \text{com}] \eta \psi)
\]

\[\text{[by predicate calculus]}\]

\[
= \forall x : T \bullet [\Gamma; x : T, N \triangleright \text{if } \{ i \bullet \psi_1 \rightarrow c_1 \triangleright \text{fi } \triangleright \text{com}] \eta \psi
\]

\[\text{[by the semantics of alternation]}\]

\[
= [\Gamma, N \triangleright \text{var } x : T \bullet \text{if } \{ i \bullet \psi_1 \rightarrow c_1 \triangleright \text{fi } \end{\triangleright \text{com}] \eta \psi
\]

\[\text{[by the semantics of local blocks]}\]

\[
\square
\]

Proof F.1.28 Law \( \langle \text{var } \cdot \text{; left dist} \rangle \)

\[
[\Gamma, N \triangleright \text{var } x : T \bullet c_1 \triangleright \text{end}; c_2 \triangleright \text{com}] \eta \psi
\]

\[
= [\Gamma, N \triangleright \text{var } x : T \bullet c_1 \triangleright \text{end}; c_2 \triangleright \text{com}] \eta ([\Gamma, N \triangleright c_2 \triangleright \text{com}] \eta \psi)
\]

\[\text{[by the semantics of sequential composition]}\]

\[
= \forall x : T \bullet [\Gamma; x : T, N \triangleright c_1 \triangleright \text{com}] \eta ([\Gamma, N \triangleright c_2 \triangleright \text{com}] \eta \psi)
\]

\[\text{[by the semantics of local blocks]}\]

\[
= \forall x : T \bullet [\Gamma; x : T, N \triangleright c_1 \triangleright \text{com}] \eta \psi[\text{x not free in } c \text{ and } \psi]
\]

\[\text{[by the semantics of sequential composition]}\]

\[
= [\Gamma, N \triangleright \text{var } x : T \bullet c_1 \triangleright \text{end}; c_2 \triangleright \text{com}] \eta \psi
\]

\[\text{[by the semantics of local blocks]}\]

\[
\square
\]
F.1. PROOFS OF LAWS

Proof F.1.29 Law \( <\text{var;}\; \text{right dist}> \)

\[
[\Gamma, N \triangleright c_1; \text{var} \; x : T \bullet c_2 \text{ end} : \text{com}]\eta \psi
= [\Gamma, N \triangleright c_1 : \text{com}]\eta ([\Gamma, N \triangleright \text{var} \; x : T \bullet c_2 \text{ end} : \text{com}]\eta \psi)
\]
[by the semantics of sequential composition]

\[
= [\Gamma, N \triangleright c_1 : \text{com}]\eta (\forall x : T \bullet [\Gamma; \; x : T, N \triangleright c_2 \text{ end} : \text{com}]\eta \psi)
\]
[by the semantics of local blocks]

\[
= \forall x : T \bullet [\Gamma, N \triangleright c_1 : \text{com}]\eta (\forall x : T \bullet [\Gamma; \; x : T, N \triangleright c_2 \text{ end} : \text{com}]\eta \psi)
\]
[by predicate calculus]

\[
= \forall x : T \bullet [\Gamma; \; x : T, N \triangleright c_1 ; c_2 : \text{com}]\eta \psi
\]
[by the semantics of sequential composition]

\[
= [\Gamma, N \triangleright \text{var} \; x : T \bullet c_1 ; c_2 \text{ end} : \text{com}]\eta \psi
\]
[by the semantics of local blocks]

\[
\square
\]

Proof F.1.30 Law \( <\text{var;}\; := \text{final value}> \)

\[
[\Gamma, N \triangleright \text{var} \; x : T \bullet c; \; x := e \text{ end} : \text{com}]\eta \psi
= \forall x : T \bullet [\Gamma; \; x : T, N \triangleright c; \; x := e : \text{com}]\eta \psi
\]
[by the semantics of local blocks]

\[
= \forall x : T \bullet [\Gamma; \; x : T, N \triangleright c : \text{com}]\eta ([\Gamma; \; x : T, N \triangleright x := e : \text{com}]\eta \psi)
\]
[by the semantics of sequential composition]

\[
= \forall x : T \bullet [\Gamma; \; x : T, N \triangleright c : \text{com}]\eta (e \neq \text{error} \land \psi[e/x]) \text{by the semantics of assignment}
\]

\[
= \forall x : T \bullet [\Gamma; \; x : T, N \triangleright c : \text{com}]\eta \psi
\]
[\(x\) is not free in \(\psi\)]

\[
= [\Gamma, N \triangleright \text{var} \; x : T \bullet c \text{ end} : \text{com}]\eta \psi
\]
[by the semantics of local blocks]

\[
\square
\]

Proof F.1.31 Law \( <\text{var;}\; \text{dist}> \)

\[
[\Gamma, N \triangleright \text{var} \; x : T \bullet c_1 \text{ end}; \; \text{var} \; x : T \bullet c_2 \text{ end} : \text{com}]\eta \psi
= [\Gamma, N \triangleright \text{var} \; x : T \bullet c_1 \text{ end} : \text{com}]\eta ([\Gamma, N \triangleright \text{var} \; x : T \bullet c_2 \text{ end} : \text{com}]\eta \psi)
\]
[by the semantics of sequential composition]

\[
= [\Gamma, N \triangleright \text{var} \; x : T \bullet c_1 \text{ end} : \text{com}]\eta (\forall x : T \bullet [\Gamma; \; x : T, N \triangleright c_2 : \text{com}]\eta \psi)
\]
[by the semantics of local blocks]

\[
= \forall x : T \bullet ([\Gamma; \; x : T, N \triangleright c_1 : \text{com}]\eta (\forall x : T \bullet [\Gamma; \; x : T, N \triangleright c_2 : \text{com}]\eta \psi)
\]
[by the semantics of local blocks]

\[
\Rightarrow \forall x : T \bullet ([\Gamma; \; x : T, N \triangleright c_1 : \text{com}]\eta ([\Gamma; \; x : T, N \triangleright c_2 : \text{com}]\eta \psi)
\]
APPENDIX F. PROOFS OF LAWS OF COMMANDS

[by predicate calculus and monotonicity]

\[ \forall x : T \cdot [\Gamma; x : T, N \triangleright c_1; c_2 : \text{com}]\eta \psi \]  

[by the semantics of sequential composition]

\[ = [\Gamma, N \triangleright \text{var } x : T \cdot c_1; c_2 \text{ end} : \text{com}]\eta \psi \]  

[by the semantics of local blocks]

\[ = \exists x_1 : T_1 \cdot [\exists x_2 : T_2 \cdot [\Gamma; x_1 : T_1, N \triangleright c : \text{com}]\eta \psi] \]  

[by predicate calculus]

\[ = [\Gamma, N \triangleright \text{avar } x_2 : T_2 \cdot (\text{avar } x_1 : T_1 \cdot c \text{ end}) \text{ end} : \text{com}]\eta \psi \]  

[by the semantics of angelic blocks]

\[ = [\Gamma, N \triangleright \text{avar } x_2 : T_2 \cdot \text{avar } x_1 : T_1 \cdot c \text{ end} : \text{com}]\eta \psi \]  

[by the semantics of angelic blocks]

\[ = [\Gamma, N \triangleright \text{var } x : T \cdot c \text{ end} : \text{com}]\eta \psi \]  

[by the semantics of local blocks]

\[ = \exists x_2 : T_2 \cdot [\exists x_1 : T_1 \cdot [\Gamma; x_1, x_2 : T_1, T_2, N \triangleright c : \text{com}]\eta \psi] \]  

[by predicate calculus]

\[ = [\Gamma, N \triangleright \text{avar } x_1 : T_1 \cdot (\text{avar } x_2 : T_2 \cdot c \text{ end}) \text{ end} : \text{com}]\eta \psi \]  

[by the semantics of angelic blocks]

\[ = [\Gamma, N \triangleright \text{avar } x_2 : T_2 \cdot (\text{avar } x_1 : T_1 \cdot c \text{ end}) \text{ end} : \text{com}]\eta \psi \]  

[by the semantics of angelic blocks]

\[ = \exists x_2 : T_2 \cdot [\exists x_1 : T_1 \cdot [\Gamma; x_1, x_2 : T_1, T_2, N \triangleright c : \text{com}]\eta \psi] \]  

[by predicate calculus]

\[ = [\Gamma, N \triangleright \text{avar } x_2 : T_2 \cdot \text{avar } x_1 : T_1 \cdot c \text{ end} : \text{com}]\eta \psi \]  

[by the semantics of angelic blocks]

\[ = [\Gamma, N \triangleright \text{var } x : T \cdot c \text{ end} : \text{com}]\eta \psi \]  

[by the semantics of local blocks]

\[ = \exists x_1 : T_1 \cdot [\exists x_2 : T_2 \cdot [\Gamma; x_1 : T_1, N \triangleright c : \text{com}]\eta \psi] \]  

[by predicate calculus]

\[ = [\Gamma, N \triangleright \text{avar } x_2 : T_2 \cdot (\text{avar } x_1 : T_1 \cdot c \text{ end}) \text{ end} : \text{com}]\eta \psi \]  

[by the semantics of angelic blocks]

\[ = [\Gamma, N \triangleright \text{var } x : T \cdot c \text{ end} : \text{com}]\eta \psi \]  

[by the semantics of local blocks]

\[ = \exists x_2 : T_2 \cdot [\exists x_1 : T_1 \cdot [\Gamma; x_1, x_2 : T_1, T_2, N \triangleright c : \text{com}]\eta \psi] \]  

[by predicate calculus]

\[ = [\Gamma, N \triangleright \text{avar } x_2 : T_2 \cdot (\text{avar } x_1 : T_1 \cdot c \text{ end}) \text{ end} : \text{com}]\eta \psi \]  

[by the semantics of angelic blocks]

\[ = [\Gamma, N \triangleright \text{var } x : T \cdot c \text{ end} : \text{com}]\eta \psi \]  

[by the semantics of local blocks]

\[ = \exists x_1 : T_1 \cdot [\exists x_2 : T_2 \cdot [\Gamma; x_1 : T_1, N \triangleright c : \text{com}]\eta \psi] \]  

[by predicate calculus]

\[ = [\Gamma, N \triangleright \text{avar } x_2 : T_2 \cdot (\text{avar } x_1 : T_1 \cdot c \text{ end}) \text{ end} : \text{com}]\eta \psi \]  

[by the semantics of angelic blocks]

\[ = [\Gamma, N \triangleright \text{var } x : T \cdot c \text{ end} : \text{com}]\eta \psi \]  

[by the semantics of local blocks]
Proof F.1.34 Law \(\langle \text{avar elim} \rangle\)

\[
\begin{align*}
[\Gamma, N \triangleright \text{avar } x : T \bullet c \text{ end : com}] & \eta \psi \\
= \exists x : T \bullet [\Gamma; x : T, N \triangleright c : \text{com}] & \eta \psi \\
= \exists x : T \bullet [\Gamma, N \triangleright c : \text{com}] & \eta \psi [x \text{ not free in } c] \\
= [\Gamma, N \triangleright c : \text{com}] & \eta \psi [x \text{ not free in } c \text{ and } \psi]
\end{align*}
\]

\[\square\]

Proof F.1.35 Law \(\langle \text{avar rename} \rangle\)

\[
\begin{align*}
[\Gamma, N \triangleright \text{avar } x_1 : T \bullet c \text{ end : com}] & \eta \psi \\
= \exists x_1 : T \bullet [\Gamma; x_1 : T, N \triangleright c \text{ end : com}] & \eta \psi [x_2/\Gamma] [x_2 \text{ not free in } c] \\
= \exists x_2 : T \bullet ([\Gamma; x_1 : T, N \triangleright c \text{ end : com}] & \eta \psi [x_2/x_1] \text{ by Lemma }[13] \text{ and predicate calculus} \\
= \exists x_2 : T \bullet ([\Gamma; x_2 : T, N \triangleright c \text{ end : com}] & \eta \psi [x_2/x_1] \text{ by } \psi \text{ not free in } c \\
= [\Gamma, N \triangleright \text{avar } x_2 : T \bullet c[x_2/x_1] \text{ end : com}] & \eta \psi [x_2/x_1] \text{ by Lemma }[14] \\
= [\Gamma, N \triangleright \text{avar } x_2 : T \bullet c \text{ end : com}] & \eta \psi [x_2/x_1] \text{ by the semantics of angelic blocks}
\end{align*}
\]

\[\square\]

Proof F.1.36 Law \(\langle \text{avar - if dist} \rangle\)

\[
\begin{align*}
[\Gamma, N \triangleright \text{if } []i \bullet \psi_i \rightarrow \text{avar } x : T \bullet c_i \text{ end} : \text{com}] & \eta \psi \\
= \forall i \bullet \psi_i \land (\forall i \bullet \psi_i \Rightarrow [\Gamma, N \triangleright \text{avar } x : T \bullet c_i : \text{com}] & \eta \psi) \text{ by the semantics of alternation} \\
= \forall i \bullet \psi_i \land (\forall i \bullet \psi_i \Rightarrow \exists x : T \bullet [\Gamma; x : T, N \triangleright c_i \text{ end : com}] & \eta \psi) \text{ by the semantics of angelic blocks} \\
= \exists x : T \bullet (\forall i \bullet \psi_i \Rightarrow [\Gamma; x : T, N \triangleright c_i : \text{com}] & \eta \psi) \text{ by predicate calculus} \\
= \exists x : T \bullet [\Gamma; x : T, N \triangleright \text{if } []i \bullet \psi_i \rightarrow c_i \text{ fi : com}] & \eta \psi [x \text{ not free in } c] \text{ by the semantics of alternation} \\
= [\Gamma, N \triangleright \text{avar } x : T \bullet \text{if } []i \bullet \psi_i \rightarrow c_i \text{ fi end : com}] & \eta \psi [x \text{ not free in } c \text{ and } \psi] \text{ by the semantics of angelic blocks}
\end{align*}
\]

\[\square\]

Proof F.1.37 Law \(\langle \text{avar-; left dist} \rangle\)
\[ \begin{align*}
&[\Gamma, N \triangleright \text{avar } x : T \bullet c_1 \text{ end } : \text{com}] \eta \psi \\
&= [\Gamma, N \triangleright \text{avar } x : T \bullet c_1 \text{ end } : \text{com}] \eta ((\Gamma, N \triangleright c_2 : \text{com}] \eta \psi) \\
&= \exists x : T \bullet [\Gamma; x : T, N \triangleright c_1 : \text{com}] \eta ((\Gamma, x : T, N \triangleright c_2 : \text{com}] \eta \psi) \\
&= \exists x : T \bullet [\Gamma; x : T, N \triangleright c_1 : \text{com}] \eta \psi \\
&= [\Gamma, N \triangleright \text{avar } x : T \bullet c_1 ; c_2 \text{ end } : \text{com}] \eta \psi \\
\end{align*} \]

\[ x \text{ not free in } c \text{ and } \psi \]

\[ \square \]

**Proof F.1.38** Law \(\text{avar} - ; \text{right dist}\)

\[ \begin{align*}
&[\Gamma, N \triangleright c_1 ; \text{avar } x : T \bullet c_2 \text{ end } : \text{com}] \eta \psi \\
&= [\Gamma, N \triangleright c_1 ; \text{com}] \eta ((\Gamma, N \triangleright x : T \bullet c_2 \text{ end } : \text{com}] \eta \psi) \\
&= [\Gamma, N \triangleright c_1 ; \text{com}] \eta (\exists x : T \bullet [\Gamma; x : T, N \triangleright c_2 \text{ end } : \text{com}] \eta \psi) \\
&= \exists x : T \bullet [\Gamma, N \triangleright c_1 : \text{com}] \eta (\exists x : T \bullet [\Gamma; x : T, N \triangleright c_2 \text{ end } : \text{com}] \eta \psi) \\
&= \exists x : T \bullet [\Gamma, N \triangleright c_1 ; c_2 \text{ end } : \text{com}] \eta \psi \\
&= [\Gamma, N \triangleright \text{avar } x : T \bullet c_1 ; c_2 \text{ end } : \text{com}] \eta \psi \\
&= \exists x : T \bullet [\Gamma; x : T, N \triangleright c_1 : \text{com}] \eta \psi \\
&= \exists x : T \bullet [\Gamma; x : T, N \triangleright c_1 ; c_2 \text{ end } : \text{com}] \eta \psi \\
&= [\Gamma, N \triangleright \text{avar } x : T \bullet c_1 ; c_2 \text{ end } : \text{com}] \eta \psi \\
&= [\Gamma, N \triangleright \text{avar } x : T \bullet c_1 ; c_2 \text{ end } : \text{com}] \eta \psi \\
\end{align*} \]

\[ \square \]

**Proof F.1.39** Law \(\text{avar} - := \text{final value}\)

\[ \begin{align*}
&[\Gamma, N \triangleright \text{avar } x : T \bullet c ; x := e \text{ end } : \text{com}] \eta \psi \\
&= \exists x : T \bullet [\Gamma; x : T, N \triangleright c ; x := e : \text{com}] \eta \psi \\
&= \exists x : T \bullet [\Gamma; x : T, N \triangleright c : \text{com}] \eta ((\Gamma; x : T, N \triangleright x := e : \text{com}] \eta \psi) \\
&= \exists x : T \bullet [\Gamma; x : T, N \triangleright c : \text{com}] \eta (e \neq \text{error} \land \psi[e/x]) \\
&= \exists x : T \bullet [\Gamma; x : T, N \triangleright c : \text{com}] \eta \psi \\
&= [\Gamma, N \triangleright \text{avar } x : T \bullet c \text{ end } : \text{com}] \eta \psi \\
\end{align*} \]

\[ x \text{ not free in } \psi \]

\[ \square \]
Proof F.1.40 Law (avar - var relationship)

\[\begin{align*}
\Gamma, N \triangleright (\text{avar } x : T \bullet \{ x = e \}; \ c \ \text{end}) & : \text{com}\eta \psi \\
= \exists x : T \bullet [\Gamma; x : T, N \triangleright \{ x = e \}; \ c : \text{com}]\eta \psi & \quad \text{[by the semantics of angelic blocks]} \\
= \exists x : T \bullet [\Gamma; x : T, N \triangleright \{ x = e \} : \text{com}]\eta \psi ( [\Gamma; x : T, N \triangleright c : \text{com}]\eta \psi) & \\
= \exists x : T \bullet [\Gamma; x : T, N \triangleright : \{ x = e, \text{true} \} : \text{com}]\eta \psi ( [\Gamma; x : T, N \triangleright c : \text{com}]\eta \psi) & \quad \text{[by the semantics of sequential composition]} \\
= \exists x : T \bullet [\Gamma; x : T, N \triangleright c : \text{com}]\eta \psi & \quad \text{[by the definition of assumption]} \\
= \forall x : T \bullet x = e \Rightarrow ([\Gamma; x : T, N \triangleright c : \text{com}]\eta \psi) & \quad \text{[by predicate calculus]} \\
= \forall x : T \bullet [\Gamma; x : T, N \triangleright : \{ \text{true}, x = e \} : \text{com}]\eta \psi ( [\Gamma; x : T, N \triangleright c : \text{com}]\eta \psi) & \quad \text{[by the semantics of specification statement]} \\
= \forall x : T \bullet [\Gamma; x : T, N \triangleright [ x = e ] : \text{com}]\eta \psi ( [\Gamma; x : T, N \triangleright c : \text{com}]\eta \psi) & \quad \text{[by the semantics of coercion]} \\
= \forall x : T \bullet [\Gamma; x : T, N \triangleright [ x = e ] : c : \text{com}]\eta \psi & \quad \text{[by the semantics of sequential composition]} \\
= [\Gamma, N \triangleright (\text{avar } x : T \bullet [ x = e ]; \ c \ \text{end}) : \text{com}]\eta \psi & \quad \text{[by the semantics of angelic blocks]}
\end{align*}\]

Proof F.1.41 Law (avar - ; dist)

\[\begin{align*}
\Gamma, N \triangleright \text{avar } x : T \bullet \ c_1; \ c_2 \ \text{end} & : \text{com}\eta \psi \\
= \exists x : T \bullet [\Gamma; x : T, N \triangleright c_1; \ c_2 : \text{com}]\eta \psi & \quad \text{[by the semantics of angelic blocks]} \\
= \exists x : T \bullet [\Gamma; x : T, N \triangleright c_1 : \text{com}]\eta \psi ( [\Gamma; x : T, N \triangleright c_2 : \text{com}]\eta \psi) & \quad \text{[by the semantics of sequential composition]} \\
\Rightarrow \exists x : T \bullet [\Gamma; x : T, N \triangleright c_1 : \text{com}]\eta \psi ( [\exists x : T \bullet [\Gamma; x : T, N \triangleright c_2 : \text{com}]\eta \psi) & \quad \text{[by predicate calculus and monotonicity]} \\
= \exists x : T \bullet [\Gamma; x : T, N \triangleright c_1 : \text{com}]\eta \psi ( [\Gamma, N \triangleright \text{avar } x : T \bullet c_2 \ \text{end} : \text{com}]\eta \psi) & \quad \text{[by the semantics of angelic blocks]} \\
= [\Gamma, N \triangleright \text{avar } x : T \bullet c_1 \ \text{end} : \text{com}]\eta \psi ( [\Gamma, N \triangleright \text{avar } x : T \bullet c_2 \ \text{end} : \text{com}]\eta \psi) & \quad \text{[by the semantics of angelic blocks]} \\
= [\Gamma, N \triangleright \text{avar } x : T \bullet c_1 \ \text{end}; \ \text{avar } x : T \bullet c_2 \ \text{end} : \text{com}]\eta \psi & \quad \text{[by the semantics of sequential composition]}
\end{align*}\]
APPENDIX F. PROOFS OF LAWS OF COMMANDS

Proof F.1.42 Law \( \langle \text{avar- := initial value} \rangle \)

\[
\begin{align*}
\Gamma, N \triangleright (\text{avar } x : T \cdot x := e; \text{ end}) : \text{com} \eta \psi & \\
= \exists x : T \bullet [\Gamma; x : T, N \triangleright x := e; c : \text{com}] \eta \psi & \text{[by the semantics of angelic blocks]} \\
= \exists x : T \bullet [\Gamma; x : T, N \triangleright x := e : \text{com}] \eta ([\Gamma; x : T, N \triangleright c : \text{com}] \eta \psi) & \text{[by the semantics of sequential composition]} \\
= \exists x : T \cdot e \neq \text{error} \land ([\Gamma; x : T, N \triangleright c : \text{com}] \eta \psi)[e/x] & \text{[by the semantics of assignment]} \\
\Rightarrow \exists x : T \cdot [\exists x : T \bullet [\Gamma; x : T, N \triangleright c : \text{com}] \eta \psi & \text{[by predicate calculus]} \\
= [\Gamma, N \triangleright (\text{avar } x : T \cdot c \text{ end}) : \text{com} \eta \psi & \text{[by the semantics of angelic blocks]} \\
\end{align*}
\]

Proof F.1.43 Law \( \langle \text{var - avar refinement} \rangle \)

\[
\begin{align*}
\Gamma, N \triangleright (\text{avar } x : T \cdot c \text{ end}) : \text{com} \eta \psi & \\
= \forall x : T \bullet [\Gamma; x : T, N \triangleright c : \text{com}] \eta \psi & \text{[by the semantics of angelic blocks]} \\
\Rightarrow \exists x : T \bullet [\exists x : T \bullet [\Gamma; x : T, N \triangleright c : \text{com}] \eta \psi & \text{[by predicate calculus]} \\
= [\Gamma, N \triangleright (\text{avar } x : T \cdot c \text{ end}) : \text{com} \eta \psi & \text{[by the semantics of angelic blocks]} \\
\end{align*}
\]

F.2 Proof of additional command laws

In this section, we present some proofs of laws presented in Section D.7.

Proof F.2.1 Law \( \langle \text{if identical guarded commands} \rangle \)

\[
\begin{align*}
[\Gamma, N \triangleright \text{if } \mid i : 1..n \bullet \psi_i \rightarrow c \text{ fi} : \text{com} \eta \psi & \\
= (\lor i \bullet \psi_i) \land (\land i \bullet \psi_i \Rightarrow [\Gamma, N \triangleright c : \text{com}] \eta \psi) & \text{[by the semantics of alternation]} \\
= (\lor i \bullet \psi_i) \land ((\lor i \bullet \psi_i) \Rightarrow [\Gamma, N \triangleright c : \text{com}] \eta \psi) & \text{[by predicate calculus]} \\
= [\Gamma, N \triangleright c : \text{com} \eta \psi & \text{[by predicate calculus]} \\
\end{align*}
\]
F.2. PROOF OF ADDITIONAL COMMAND LAWS

Proof F.2.2 Law \(\langle\text{iteration guards}\rangle\)

\[
\begin{align*}
\text{do } & \left[ i \cdot g_i \rightarrow c_i \right] \text{ od} \\
= & \text{ Abbreviation} \\
\text{rec } & Y \cdot \text{ if } \left[ i \cdot G_i \rightarrow c_i; Y \left[ \right] \rightarrow (\forall i \cdot G_i) \rightarrow \text{ skip fi end}\right] \\
\leq \langle\text{alternation guards}\rangle \\
\text{rec } & Y \cdot \text{ if } \left[ i \cdot H_i \rightarrow c_i; Y \left[ \right] \rightarrow (\forall i \cdot H_i) \rightarrow \text{ skip fi end}\right]
\end{align*}
\]

= Abbreviation

\[
\begin{align*}
\text{do } & \left[ i \cdot h_i \rightarrow c_i \right] \text{ od}
\end{align*}
\]

For the proof obligation of law \(\langle\text{alternation guards}\rangle\) we have

\[
\begin{align*}
(i) & \quad GG \lor \neg GG \Rightarrow HH \lor \neg HH \\
(ii) & \quad GG \lor \neg GG \Rightarrow (H_i \Rightarrow G_i) \equiv H_i \Rightarrow G_i \\
& \quad GG \lor \neg GG \Rightarrow \neg HH \Rightarrow \neg GG \equiv GG \Rightarrow HH
\end{align*}
\]

\(\square\)

Proof F.2.3 Law \(\langle\text{weakening guards}\rangle\)

\[
\begin{align*}
\text{if } & \left[ i \cdot g_i \land g \rightarrow c_i \right] \text{ fi} \\
\leq \langle\text{alternation guards}\rangle \\
\text{if } & \left[ i \cdot g_i \rightarrow c_i \right] \text{ fi}
\end{align*}
\]

For the proof obligation of law \(\langle\text{alternation guards}\rangle\)

\[
\begin{align*}
& (\forall i \cdot g_i \land g) \\
& \equiv (\forall i \cdot g_i) \land g \\
& \Rightarrow (\forall i \cdot g_i)
\end{align*}
\]

\[
\begin{align*}
& (\forall i \cdot g_i \land g) \land g_i \\
& \equiv (\forall i \cdot g_i) \land g \land g_i
\end{align*}
\]
\[ \Rightarrow g_i \land g \]

**Proof F.2.4 Law (innocuous assumption-reading)**

\[
\begin{align*}
[\Gamma, N \triangleright le_1 := le.x : \text{com}]\eta \psi \\
= le \neq \text{error} \land \psi[le.x/le] & \quad \text{[by the semantics of assignment]} \\
= le \neq \text{null} \land le \neq \text{error} \land le.x \neq \text{error} \land \psi[le.x/le_1] & \quad \text{[by the semantic definition of attribute selection]} \\
= le \neq \text{null} \land le \neq \text{error} \land (le.x \neq \text{error} \land \psi[le.x/le_1]) & \quad \text{[by predicate calculus]} \\
= le \neq \text{null} \land le \neq \text{error} \land \text{true} \Rightarrow (le.x \neq \text{error} \land \psi[le.x/le_1]) & \quad \text{[by predicate calculus]} \\
= le \neq \text{null} \land le \neq \text{error} \land \text{true} \Rightarrow ([\Gamma, N \triangleright le_1 := le.x : \text{com}]\eta \psi) \\
& \quad \text{[by the semantics of assignment]} \\
= [\Gamma, N \triangleright : [le \neq \text{null} \land le \neq \text{error}] : \text{com}]\eta \psi ([\Gamma, N \triangleright le_1 := le.x : \text{com}]\eta \psi) & \quad \text{[by the semantics of specification statement]} \\
= [\Gamma, N \triangleright \{le \neq \text{null} \land le \neq \text{error}\} le_1 := le.x : \text{com}]\eta \psi & \quad \text{[by the semantics of sequential composition]} \\
\end{align*}
\]

**Proof F.2.5 Law (innocuous assumption-writing)**

We assume that \( exp \) is not \text{error}, as our main concern here is the value of the expression \( le \).

\[
\begin{align*}
[\Gamma, N \triangleright le.x := exp : \text{com}]\eta \psi \\
= [\Gamma, N \triangleright le := (le; x : exp) : \text{com}]\eta \psi & \quad \text{[by a syntactic transformation]} \\
= (le; x : exp) \neq \text{error} \land \psi[(le; x : exp)/le] & \quad \text{[by the semantics of assignment]} \\
= le \neq \text{null} \land le \neq \text{error} \land (le; x : exp) \neq \text{error} \land \psi[(le; x : exp)/le] & \quad \text{[by the semantic definition of update expression and assumption]} \\
& \quad \text{[by the semantic definition of attribute selection]} \\
= le \neq \text{null} \land le \neq \text{error} \land [\Gamma, N \triangleright le := (le; x : exp) : \text{com}]\eta \psi & \quad \text{[by the semantics of assignment]} \\
= le \neq \text{null} \land le \neq \text{error} \land \text{true} \Rightarrow [\Gamma, N \triangleright le := (le; x : exp) : \text{com}]\eta \psi & \quad \text{[by predicate calculus]} \\
= [\Gamma, N \triangleright : [le \neq \text{null} \land le \neq \text{error}] : \text{com}]\eta \psi ([\Gamma, N \triangleright le := (le; x : exp) : \text{com}]\eta \psi) & \quad \text{[by the semantics of specification statement]} \\
= [\Gamma, N \triangleright : [le \neq \text{null} \land le \neq \text{error}] ; le := (le; x : exp) : \text{com}]\eta \psi & \quad \text{[by the semantics of sequential composition]} \\
= [\Gamma, N \triangleright \{le \neq \text{null} \land le \neq \text{error}\} le := (le; x : exp) : \text{com}]\eta \psi \\
\end{align*}
\]
F.2. PROOF OF ADDITIONAL COMMAND LAWS

247

\[ \{ le \neq \text{null} \land le \neq \text{error} \} \leq x := \text{exp} : \text{com} \| \eta \psi \]  

[by the definition of assumption]

\[ \varnothing \]

\[ \Box \]

Proof F.2.6 Law (assumption guard)

\[ [\Gamma, N \triangleright] \text{if } [] i \bullet \psi_i \rightarrow c_i \cdot \text{fi} : \text{com} \| \eta \psi \]

= \( (\forall i \bullet \psi_i) \land (\land i \bullet \psi_i \Rightarrow [\Gamma, N \triangleright c : \text{com} \| \eta \psi) \)  

[by the semantics of alternation]

= \( (\forall i \bullet \psi_i) \land (\land i \bullet \psi_i \land \psi_i \Rightarrow [\Gamma, N \triangleright c : \text{com} \| \eta \psi) \)  

[by predicate calculus]

= \( (\forall i \bullet \psi_i) \land (\land i \bullet \psi_i \Rightarrow \psi_i \Rightarrow (\Gamma, N \triangleright c_i \cdot \text{fi} : \text{com} \| \eta \psi)) \)  

[by predicate calculus]

= \( (\forall i \bullet \psi_i) \land (\land i \bullet \psi_i \Rightarrow [\Gamma, N \triangleright : [\psi_i, \text{true}] \text{com} \| ((\Gamma, N \triangleright c_i \cdot \text{fi} : \text{com} \| \eta \psi)) \)  

[by the semantics of specification statement]

= \( (\forall i \bullet \psi_i) \land (\land i \bullet \psi_i \Rightarrow [\Gamma, N \triangleright : \{ \psi_i \} c_i \cdot \text{fi} : \text{com} \| \eta \psi) \)  

[by definition of assumption]

= \( [\Gamma, N \triangleright \text{if } [] i \bullet \psi_i \rightarrow \{ \psi_i \} c_i \cdot \text{fi} : \text{com} \| \eta \psi \)  

[by the semantics of sequential composition]

[by the semantics of alternation]

\[ \varnothing \]

\[ \Box \]

Proof F.2.7 Law (var block absorb assumption)

\[ [\Gamma, N \triangleright \{ \psi \} \text{ var } x : T \cdot c \text{ end } : \text{com} \| \eta \psi \]

= \( [\Gamma, N \triangleright : [\phi, \text{true}] ; \text{ var } x : T \cdot c \text{ end } : \text{com} \| \eta \psi \)  

[by definition of assumption]

= \( [\Gamma, N \triangleright : [\phi, \text{true}] : \text{com} \| ((\Gamma, N \triangleright \text{ var } x : T \cdot c \text{ end } : \text{com} \| \eta \psi) \)  

[by the semantics of sequential composition]

= \( \phi \land \text{true} \Rightarrow [\Gamma, N \triangleright \text{ var } x : T \cdot c \text{ end } : \text{com} \| \eta \psi \)  

[by the semantics of specification statement]

= \( \phi \land [\Gamma, N \triangleright \text{ var } x : T \cdot c \text{ end } : \text{com} \| \eta \psi \)  

[by predicate calculus]

= \( \phi \land (\forall x : T \bullet [\Gamma ; x : T, N \triangleright c : \text{com} \| \eta \psi) \)  

[by the semantics of local blocks]

= (\( (\forall x : T \bullet \phi) \land (\forall x : T \bullet [\Gamma ; x : T, N \triangleright c : \text{com} \| \eta \psi) \)  

[by hypothesis]

= (\( \forall x : T \bullet \phi \land [\Gamma ; x : T, N \triangleright c : \text{com} \| \eta \psi \)  

[by predicate calculus]

= (\( \forall x : T \bullet \phi \land \text{true} \Rightarrow ([\Gamma ; x : T, N \triangleright c : \text{com} \| \eta \psi) \)  

[by predicate calculus]

= (\( \forall x : T \bullet [\Gamma ; x : T, N \triangleright : [\phi, \text{true}] : \text{com} \| ((\Gamma ; x : T, N \triangleright c : \text{com} \| \eta \psi) \)  

[by the semantics of specification statement]

= (\( \forall x : T \bullet [\Gamma ; x : T, N \triangleright : [\phi, \text{true}] ; c : \text{com} \| \eta \psi \)  

[by the semantics of sequential composition]
APPENDIX F. PROOFS OF LAWS OF COMMANDS

\[ \forall x : T \bullet [\Gamma; x : T, N \triangleright \{ \phi \} c : \text{com}] \eta \psi \]
\[ = [\Gamma, N \triangleright \{ \phi \} c : \text{com}] \eta \psi \]
\[ \text{[by definition of assumption]} \]
\[ = [\Gamma, N \triangleright \{ \phi \} c : \text{com}] \eta \psi \]
\[ \text{[by the semantics of local blocks]} \]

\( \square \)

**Proof F.2.8 Law (alternation absorb assumption)**

\[
[\Gamma, N \triangleright \{ \phi \} \text{ if } \psi_i \rightarrow c_i \text{ fi : com}] \eta \psi
\]
\[
= [\Gamma, N \triangleright \{ \phi, \text{true} \}; \text{ if } \psi_i \rightarrow c_i \text{ fi : com}] \eta \psi \]
\[
= [\Gamma, N \triangleright \{ \phi, \text{true} \}; (\Gamma, N \triangleright \text{ if } \psi_i \rightarrow c_i \text{ fi : com}] \eta \psi) \]
\[ \text{[by the semantics of sequential composition]} \]
\[
= \phi \land \text{true} \Rightarrow ((\Gamma, N \triangleright \text{ if } \psi_i \rightarrow c_i \text{ fi : com}] \eta \psi) \]
\[ \text{[by the semantics of specification statement]} \]
\[
= \phi \land ((\Gamma, N \triangleright \text{ if } \psi_i \rightarrow c_i \text{ fi : com}] \eta \psi) \]
\[ \text{[by predicate calculus]} \]
\[
= \phi \land ((\lor i \bullet \psi_i) \land (\land i \bullet \psi_i \Rightarrow ((\Gamma, N \triangleright c_i \text{ : com}] \eta \psi))) \]
\[ \text{[by the semantics of alternation]} \]
\[
= (\phi \land (\lor i \bullet \psi_i) \land (\land i \bullet \phi \land \psi_i \Rightarrow (\Gamma, N \triangleright c_i \text{ : com}] \eta \psi))) \]
\[ \text{[by predicate calculus]} \]
\[
= [\Gamma, N \triangleright \text{ if } \phi \land \psi_i \rightarrow c_i \text{ fi : com}] \eta \psi \]
\[ \text{[by the semantics of alternation]} \]

\( \square \)

**Proof F.2.9 Law (pcom absorb assumption)**

By induction on the structure of parameterised command of the form \( pds \bullet c \).

**Case (\( \bullet c \))**

\[
[\Gamma, N \triangleright \{ \phi \}; (\bullet c()) : \text{com}] \eta \psi
\]
\[
= [\Gamma, N \triangleright \{ \phi \}; c : \text{com}] \eta \psi \]
\[ \text{[by a syntactic transformation]} \]
\[
= [\Gamma, N \triangleright (\bullet \{ \phi \}; c()) : \text{com}] \eta \psi \]
\[ \text{[by a syntactic transformation]} \]

**Case Value parameter.**

\[
[\Gamma, N \triangleright \{ \phi \}; (\text{val } x : T \bullet c)(e) : \text{com}] \eta \psi
\]
\[
= [\Gamma, N \triangleright \{ \phi \}; \text{com}] \eta ((\Gamma, N \triangleright (\text{val } x : T \bullet c)(e) : \text{com}) \eta \psi) \]
\[ \text{[by the semantics of sequential composition]} \]
\[
= [\Gamma, N \triangleright \{ \phi, \text{true} \}; \text{com}] \eta ((\Gamma, N \triangleright (\text{val } x : T \bullet c)(e) : \text{com}) \eta \psi) \]
\[ \text{[by the definition of assumption]} \]
\[
= \phi \land \text{true} \Rightarrow ((\Gamma, N \triangleright (\text{val } x : T \bullet c)(e) : \text{com}) \eta \psi) \]
\[ \text{[by the semantics of specification statement]} \]
F.2. PROOF OF ADDITIONAL COMMAND LAWS

\[ \phi \land [\Gamma, N \triangleright (\text{val } x : T \bullet c)(e) : \text{com}] \eta \psi \]  
[by the predicate calculus]

\[ \phi \land [\Gamma, N \triangleright \text{var } l : T \bullet l := e; \ c[l/x] \text{end} : \text{com}] \eta \psi \]  
[by a syntactic transformation]

\[ \phi \land \forall l : T \bullet e \neq \text{error} \land ([\Gamma; l : T, N \triangleright c[l/x] : \text{com}] \eta \psi)[e/l] \]  
[by the semantics definitions]

\[ \forall l : T \bullet \phi \land \forall l : T \bullet e \neq \text{error} \land ([\Gamma; l : T, N \triangleright c[l/x] \text{end} : \text{com}] \eta \psi)[e/l] \]  
[by the hypothesis and the predicate calculus]

\[ \forall l : T \bullet e \neq \text{error} \land (\phi[l/x] \land ([\Gamma; l : T, N \triangleright c[l/x] \text{end} : \text{com}] \eta \psi)[e/l] \]  
[by the hypothesis and the semantics definitions]

\[ \forall l : T \bullet e \neq \text{error} \land (\phi[l/x] \land \text{true} \Rightarrow ([\Gamma; l : T, N \triangleright c[l/x] \text{end} : \text{com}] \eta \psi)[e/l] \]  
[by the predicate calculus]

\[ \forall l : T \bullet e \neq \text{error} \land ([\Gamma; l : T, N \triangleright \phi[l/x], \text{true} : \text{com}] \eta \]  
[by the semantics of specification statement]

\[ ([\Gamma; l : T, N \triangleright c[l/x] \text{end} : \text{com}] \eta \psi))(e/l) \]  
[by a property of substitution]

\[ \forall l : T \bullet e \neq \text{error} \land ([\Gamma; l : T, N \triangleright (\phi)[l/x] : \text{com}] \eta \]  
[by the semantics of sequential composition]

\[ ([\Gamma; l : T, N \triangleright c[l/x] \text{end} : \text{com}] \eta \psi))(e/l) \]  
[by the definition of assumption]

\[ \forall l : T \bullet e \neq \text{error} \land ([\Gamma; l : T, N \triangleright (\phi)[l/x]; c[l/x] \text{end} : \text{com}] \eta \psi)[e/l] \]  
[by the semantics of sequential composition]

\[ \forall l : T \bullet e \neq \text{error} \land ([\Gamma; l : T, N \triangleright (\phi); c[l/x] \text{end} : \text{com}] \eta \psi)[e/l] \]  
[by the definition of assumption]

\[ \text{Case Result parameter: similar.} \]

\[ \text{Case } (pd; \ pds \bullet c) \]

\[ [\Gamma, N \triangleright \phi]; (pd; \ pds \bullet c)(e, e') : \text{com}] \eta \psi \]  
[by a syntactic transformation]
By induction.

Proof F.2.10 Law \textit{(assumption before or after command)}

By induction.

Case \(x := e\)

\[
\begin{align*}
&= [\Gamma, N \triangleright \{ \phi \} : x := e : \text{com}] \eta \psi \\
&= [\Gamma, N \triangleright : [\phi, \text{true}] : x := e : \text{com}] \eta \psi \\
&= [\Gamma, N \triangleright : [\phi, \text{true}] : \text{com}] \eta ([\Gamma, N \triangleright x := e : \text{com}] \eta \psi)
\end{align*}
\]
[by the induction hypothesis]

\[
\begin{align*}
&= \phi \land \text{true} \Rightarrow ([\Gamma, N \triangleright x := e : \text{com}] \eta \psi) \\
&= \phi \land [\Gamma, N \triangleright x := e : \text{com}] \eta \psi \\
&= \phi \land e \neq \text{error} \land \psi[e/x] \\
&= e \neq \text{error} \land \psi[e/x] \land \phi \\
&= e \neq \text{error} \land \psi[e/x] \land \phi[e/x] \\
&= e \neq \text{error} \land (\phi \land \psi)[e/x]
\end{align*}
\]
[by predicate calculus]

\[
\begin{align*}
&= [\Gamma, N \triangleright x := e : \text{com}] \eta (\phi \land \psi) \\
&= [\Gamma, N \triangleright x := e : \text{com}] \eta (\phi \land \text{true} \Rightarrow \psi) \\
&= [\Gamma, N \triangleright x := e : \text{com}] \eta ([\Gamma, N \triangleright : [\phi, \text{true}] : \text{com}] \eta \psi)
\end{align*}
\]
[by predicate calculus]

\[
\begin{align*}
&= [\Gamma, N \triangleright x := e : \text{com}] \eta \psi \\
&= [\Gamma, N \triangleright x := e : \text{com}] \eta \psi \\
&= [\Gamma, N \triangleright x := e : \text{com}] \eta \psi
\end{align*}
\]
[by the semantics of specification statement]

Case \(x := [\psi_1, \psi_2]\)

From left to right.

\[
\begin{align*}
&= [\Gamma, N \triangleright \{ \phi \} : x := [\psi_1, \psi_2] : \text{com}] \eta \psi \\
&= [\Gamma, N \triangleright : [\phi, \text{true}] : x := [\psi_1, \psi_2] : \text{com}] \eta \psi \\
&= [\Gamma, N \triangleright : [\phi, \text{true}] : \text{com}] \eta ([\Gamma, N \triangleright x := [\psi_1, \psi_2] : \text{com}] \eta \psi)
\end{align*}
\]
[by definition of assumption]

\[
\begin{align*}
&= \phi \land \text{true} \Rightarrow ([\Gamma, N \triangleright x := [\psi_1, \psi_2] : \text{com}] \eta \psi) \\
&= \phi \land [\Gamma, N \triangleright x := [\psi_1, \psi_2] : \text{com}] \eta \psi \\
&= \phi \land \psi_1 \land (\forall x : T \bullet \psi_2 \Rightarrow \psi) \\
&= \psi_1 \land (\forall x : T \bullet \phi \land (\psi_2 \Rightarrow \psi)) \\
&= \psi_1 \land (\forall x : T \bullet \phi \land \psi)
\end{align*}
\]
[by predicate calculus]
F.2. PROOF OF ADDITIONAL COMMAND LAWS

\[ \Gamma, N \triangleright x : [\psi_1, \psi_2] : \text{com} \eta (\phi \land \psi) \]  

(by the semantics of specification statement)

\[ \Gamma, N \triangleright x : [\psi_1, \psi_2] : \text{com} \eta (\phi \land \text{true} \Rightarrow \psi) \]  

(by predicate calculus)

\[ \Gamma, N \triangleright x : [\psi_1, \psi_2] : \text{com} \eta (\Gamma, N \triangleright : [\phi, \text{true}] : \text{com} \eta \psi) \]  

(by the semantics of specification statement)

\[ \Gamma, N \triangleright x : [\psi_1, \psi_2] : \text{com} \eta (\Gamma, N \triangleright \{\phi\} : \text{com} \eta \psi) \]  

(by definition of assumption)

\[ \Gamma, N \triangleright x : [\psi_1, \psi_2] : \{\phi\} : \text{com} \eta \psi \]  

(by the semantics of sequential composition)

From right to left.

\[ \Gamma, N \triangleright x : [\psi_1, \psi_2] : \{\phi\} : \text{com} \eta \psi \]  

(by the semantics of sequential composition)

\[ \Gamma, N \triangleright x : [\psi_1, \psi_2] : \text{com} \eta (\Gamma, N \triangleright : [\phi, \text{true}] : \text{com} \eta \psi) \]  

(by definition of assumption)

\[ \Gamma, N \triangleright x : [\psi_1, \psi_2] : \text{com} \eta (\phi \land \text{true} \Rightarrow \psi) \]  

(by the semantics of specification statement)

\[ \psi_1 \land (\forall x: T \cdot \psi_2 \Rightarrow (\phi \land \psi)) \]  

(by predicate calculus)

\[ \psi_1 \land (\forall x: T \cdot \psi_2 \Rightarrow (\phi \land \psi)) \]  

(by predicate calculus)

\[ \phi \land \psi_1 \land (\forall x: T \cdot \psi_2 \Rightarrow \psi) \]  

(by predicate calculus)

\[ \phi \land [\Gamma, N \triangleright x : [\psi_1, \psi_2] : \text{com} \eta \psi] \]  

(by the semantics of specification statement)

\[ \phi \land \text{true} \Rightarrow [\Gamma, N \triangleright x : [\psi_1, \psi_2] : \text{com} \eta \psi] \]  

(by predicate calculus)

\[ [\Gamma, N \triangleright : [\phi, \text{true}] : \text{com} \eta (\Gamma, N \triangleright x := e : \text{com} \eta \psi) \]  

(by the semantics of specification statement)

\[ [\Gamma, N \triangleright : [\phi, \text{true}] : x : [\psi_1, \psi_2] : \text{com} \eta \psi \]  

(by the semantics of sequential composition)

\[ [\Gamma, N \triangleright \{\phi\} x : [\psi_1, \psi_2] : \text{com} \eta \psi \]  

(by definition of assumption)

The other cases are consequence of the induction hypothesis.

\[ \Box \]

**Proof F.2.11 Law** *(assumption advance command)*

\[
\{\phi\} c = \{\phi\} \{\phi\} c 
\]  

(by predicate calculus and law *(merge assumptions)*)

\[
\{\phi\} c = \{\phi\} c; \{\phi\} 
\]  

(by law *(assumption before or after command)*)

\[ \Box \]

**Proof F.2.12 Law** *(assumption distribution)*

This is a consequence of the application of laws *(assumption advance command)* (from left to right)
and \((\text{assumption before or after command})\) (from left to right), as many times as the number of commands in the command \(c\).

\[\Box\]

**Proof F.2.13** Law \(\langle\text{new assumption}\rangle\)

\[
x := \text{new} N()
\]
\[
= x : [x = \text{new} N()] \quad \text{[by law \(\langle\text{simple specification}\rangle\)]}
\]
\[
\sqsubseteq x : [x = \text{new} N() \land x \neq \text{null} \land x \neq \text{error}] \quad \text{[by law \(\langle\text{strengthen postcondition}\rangle\)} \text{ and hypothesis]}
\]
\[
\sqsubseteq x : [x = \text{new} N() \land x \neq \text{null} \land x \neq \text{error}] \{x = \text{new} N() \land x \neq \text{null} \land x \neq \text{error}\} \quad \text{[by law \(\langle\text{merge assumption}\rangle\), from right to left]}
\]
\[
= x : [x = \text{new} N() \land x \neq \text{null} \land x \neq \text{error}] \{x \neq \text{null} \land x \neq \text{error}\} \quad \text{[by law \(\langle\text{remove assumption}\rangle\)]}
\]
\[
= x := \text{new} N(); \{x \neq \text{null} \land x \neq \text{error}\} \quad \text{[by law \(\langle\text{assignment}\rangle\)]}
\]

\[\Box\]

**Proof F.2.14** Law \(\langle\text{assumption intro}\rangle\)

\[
w : [\psi_1, \psi_2]
\]
\[
= w : [\psi_1, \psi_2]; [\psi_2] \quad \text{[by predicate calculus and law \(\langle\text{absorb coercion}\rangle\)]}
\]
\[
\sqsubseteq w : [\psi_1, \psi_2]; [\psi_2] \quad \text{[by law \(\langle\text{introduce assumption}\rangle\)]}
\]
\[
= w : [\psi_1, \psi_2] \quad \text{[by law \(\langle\text{absorb coercion}\rangle\)]}
\]

\[\Box\]

**Proof F.2.15** Law \(\langle\text{repeated assignment}\rangle\)

\[
[\Gamma, N \triangleright \text{le} := e : \text{com}] \eta \psi
\]
\[
= e \neq \text{error} \land \psi[e/\text{le}] \quad \text{[by the semantics of assignment]}
\]
\[
= e \neq \text{error} \land (e \neq \text{error} \land \psi[e/\text{le}])[e/\text{le}] \quad \text{[by predicate calculus and a property of substitution]}
\]
\[
= [\Gamma, N \triangleright \text{le} := e : \text{com}] \eta (e \neq \text{error} \land \psi[e/\text{le}]) \quad \text{[by the semantics of assignment]}
\]
Proof F.2.16

\[ \begin{align*}
&= [\Gamma, N \triangleright le := e : \text{com}\eta \psi ] ( [\Gamma, N \triangleright le := e : \text{com}\eta \psi ] ) \quad \text{[by the semantics of assignment]} \\
&= [\Gamma, N \triangleright le := e; le := e : \text{com}\eta \psi ] \quad \text{[by the semantics of sequential composition]}
\end{align*} \]

\[ \square \]

Proof F.2.17 Law \textit{(order independent assignment)}

\[ \begin{align*}
[\Gamma, N \triangleright x := e_1; y := e_2 : \text{com}\eta \psi ] \\
&= [\Gamma, N \triangleright x := e_1 : \text{com}\eta \psi ] ( [\Gamma, N \triangleright y := e_2 : \text{com}\eta \psi ] ) \\
&= [\Gamma, N \triangleright x := e_1 : \text{com}\eta \psi ] ( e_2 \neq \text{error} \land \psi[e_2/y] ) \\
&= e_1 \neq \text{error} \land ( e_2 \neq \text{error} \land \psi[e_2/y][e_1/x] ) \\
&= e_1 \neq \text{error} \land e_2 \neq \text{error} \land \psi[e_2/y][e_1/x] \\
&= e_1 \neq \text{error} \land e_2 \neq \text{error} \land \psi[e_1/x][e_2/y] \\
&= e_2 \neq \text{error} \land e_1 \neq \text{error} \psi[e_1/x][e_2/y] \\
&= e_2 \neq \text{error} \land e_1 \neq \text{error} \psi[e_1/x][e_2/y] \\
&= [\Gamma, N \triangleright y := e_2 : \text{com}\eta \psi ] ( [\Gamma, N \triangleright x := e_1 : \text{com}\eta \psi ] ) \\
&= [\Gamma, N \triangleright y := e_2; x := e_1 : \text{com}\eta \psi ] \\
\end{align*} \]

[by y is not free in e_1 and a property of substitution]

[by predicate calculus]

[by y is not free in e_1]

[by the semantics of assignment]

[by the semantics of assignment]

[by the semantics of sequential composition] 

\[ \square \]

Proof F.2.17 Law \textit{(assignment seq comp exp substitution)}

By induction.

Case \( x := e \), with \( x \neq le \)

\[ \begin{align*}
&[\Gamma, N \triangleright le := exp; x := e_1[exp] : \text{com}\eta \psi ] \\
&= [\Gamma, N \triangleright le := exp : \text{com}\eta \psi ] ( [\Gamma, N \triangleright x := e_1[exp] : \text{com}\eta \psi ] ) \\
&= e_1[exp] \neq \text{error} \land \psi[e_1[exp]/x][exp/le] \\
&= e_1[exp] \neq \text{error} \land \psi[e_1[exp]/x][exp/le] \\
&= e_1[exp] \neq \text{error} \land \psi[e_1[exp]/x][exp/le] \\
&= [\Gamma, N \triangleright le := exp : \text{com}\eta \psi ] ( [\Gamma, N \triangleright x := e_1[le] : \text{com}\eta \psi ] ) \\
\end{align*} \]

[by a property of substitution]

[by a property of substitution]

[by the semantics of assignment]
APPENDIX F. PROOFS OF LAWS OF COMMANDS

[by the semantics of sequential composition]

Case x : [ψ₁[exp], ψ₂[exp]]

[Γ, N : le := exp; x : [ψ₁[exp], ψ₂[exp]] : com]η ψ
[by the semantics of sequential composition]
= exp \neq \text{error} \land (Γ, N : x : [ψ₁[exp], ψ₂[exp]] : com]η ψ)[exp/le]
[by the semantics of assignment]
= exp \neq \text{error} \land (ψ₁[exp][exp/le] \land (\forall x : T \bullet ψ₂[exp] \Rightarrow ψ))[exp/le]
[by the semantics of specification statement]
= exp \neq \text{error} \land (ψ₁[exp][exp/le] \land (\forall x : T \bullet ψ₂[exp][exp/le] \Rightarrow ψ[exp/le]))
[by a property of substitution]
= exp \neq \text{error} \land (ψ₁[le][exp/le] \land (\forall x : T \bullet ψ₂[le][exp/le] \Rightarrow ψ[exp/le]))
[by a property of substitution]
= exp \neq \text{error} \land (ψ₁[le][exp/le] \land (\forall x : T \bullet ψ₂[le] \Rightarrow ψ)[exp/le])
[by a property of substitution]
= exp \neq \text{error} \land (ψ₁[le] \land (\forall x : T \bullet ψ₂[le] \Rightarrow ψ))[exp/le]
[by a property of substitution]
= exp \neq \text{error} \land (Γ, N : x : [ψ₁[le], ψ₂[le]] : com]η ψ)[exp/le]
[by the semantics of specification statement]
[by the semantics of assignment]
= [Γ, N : le := exp; x : [ψ₁[le], ψ₂[le]] : com]η ψ
[by the semantics of sequential composition]

Case c; c'

[Γ, N : le := exp; (c; c')[exp] : com]η ψ
[by the semantics of sequential composition]
= exp \neq \text{error} \land (Γ, N : (c; c')[exp] : com]η ψ)[exp/le]
[by the semantics of assignment]
= exp \neq \text{error} \land (Γ, N : (c; c')[le] : com]η ψ)[exp/le]
[by the induction hypothesis]
[by the semantics of assignment]
= [Γ, N : le := exp; (c; c')[le] : com]η ψ
[by the semantics of sequential composition]

Case if \[i \bullet ψ_i \rightarrow c_i fi
\[ \Gamma, N \triangleright le \ := \ exp \ (if \ [i \bullet \psi_i \rightarrow c_i \ fi][exp] : com) \eta \ \psi \]

\[ = [\Gamma, N \triangleright le \ := \ exp \ : com] \eta \ ((\Gamma, N \triangleright (if \ [i \bullet \psi_i \rightarrow c_i \ fi][exp] : com) \eta \ \psi)) \]

[by the semantics of sequential composition]

\[ = \ exp \neq error \land ([\Gamma, N \triangleright (if \ [i \bullet \psi_i \rightarrow c_i \ fi][exp] : com) \eta \ \psi)][exp/le] \]

[by the semantics of assignment]

\[ = [\Gamma, N \triangleright le \ := \ exp \ : com] \eta \ ((\Gamma, N \triangleright (if \ [i \bullet \psi_i \rightarrow c_i \ fi][le] : com) \eta \ \psi)) \]

[by the induction hypothesis]

\[ = [\Gamma, N \triangleright le \ := \ exp \ (if \ [i \bullet \psi_i \rightarrow c_i \ fi][le] : com) \eta \ \psi) \]

[by the semantics of assignment]

\[ = [\Gamma, N \triangleright le \ := \ exp \ (if \ [i \bullet \psi_i \rightarrow c_i \ fi][le] : com) \eta \ \psi) \]

[by the semantics of sequential composition]

Case \((val \ x : T \bullet c)(e)\)

\[ [\Gamma, N \triangleright le \ := \ exp \ ((val \ x : T \bullet c)(e))[exp] : com] \eta \ \psi \]

\[ = [\Gamma, N \triangleright le \ := \ exp \ : com] \eta \ ((\Gamma, N \triangleright ((val \ x : T \bullet c)(e))[exp] : com) \eta \ \psi) \]

[by the semantics of sequential composition]

\[ = [\Gamma, N \triangleright le \ := \ exp \ : com] \eta \ ((\Gamma, N \triangleright (var \ l : T \bullet l := e; c[l/x] end)[exp] : com) \eta \ \psi) \]

[by a syntactic transformation]

\[ [\Gamma, N \triangleright le \ := \ exp \ : com] \eta \ ((\forall \ l : T \bullet \Gamma; l : T, N \triangleright (l := e; c[l/x])[exp] : com) \eta \ \psi) \]

[by the semantics of local blocks]

\[ = [\Gamma, N \triangleright le \ := \ exp \ : com] \eta \ ((\forall \ l : T \bullet \Gamma; l : T, N \triangleright l := e[exp]; c[l/x] : com) \eta \ \psi) \]

[by a property of substitution]

\[ = [\Gamma, N \triangleright le \ := \ exp \ : com] \eta \]

\[ (\forall \ l : T \bullet [\Gamma; l : T, N \triangleright l := e[exp] : com] \eta \ ((\Gamma; l : T, N \triangleright c[l/x] : com) \eta \ \psi)) \]

[by the semantics of sequential composition]

\[ = [\Gamma, N \triangleright le \ := \ exp \ : com] \eta \]

\[ (\forall \ l : T \bullet e[exp] \neq error \land ([\Gamma; l : T, N \triangleright c[l/x] : com] \eta \ \psi)[e[exp]/l]) \]

[by the semantics of assignment]

\[ = \ exp \neq error \land (\forall \ l : T \bullet e[exp] \neq error \land ([\Gamma; l : T, N \triangleright c[l/x] : com] \eta \ \psi)[e[exp]/l][exp/le]) \]

[by the semantics of assignment]

\[ = \ exp \neq error \land (\forall \ l : T \bullet e[exp][exp/le] \neq error \land ([\Gamma; l : T, N \triangleright c[l/x] : com] \eta \ \psi)[e[exp]/l][exp/le]) \]

[by a property of substitution]

\[ = \ exp \neq error \land (\forall \ l : T \bullet e[l][exp/le] \neq error \land ([\Gamma; l : T, N \triangleright c[l/x] : com] \eta \ \psi)[e[l][exp/le]]) \]

[by a property of substitution]
Case \(\text{var } x : T \cdot c \text{ end.}\)

\[
\begin{align*}
&= \text{exp} \neq \text{error} \land \\
&\quad (\forall l : T \cdot e[l] \neq \text{error} \land (\Gamma; l : T, N \triangleright c[l/x] : \text{com}}\eta \psi)[e[l]/l][\text{exp}/l]) \\
&= [\Gamma, N \triangleright \text{le} := \text{exp} : \text{com}}\eta \\
&\quad (\forall l : T \cdot e[l] \neq \text{error} \land (\Gamma; l : T, N \triangleright c[l/x] : \text{com}}\eta \psi)[e[l]/l]) \\
&= [\Gamma, N \triangleright \text{le} := \text{exp} : \text{com}}\eta \\
&\quad (\forall l : T \cdot \Gamma; l : T, N \triangleright l := e[l] : \text{com}\eta \psi ((\Gamma; l : T, N \triangleright c[l/x] : \text{com}}\eta \psi)) \\
&= [\Gamma, N \triangleright \text{le} := \text{exp} : \text{com}}\eta \\
&\quad (\Gamma, N \triangleright \text{var } l : T \cdot l := e[l] ; c[l/x] \text{ end} : \text{com}}\eta \psi) \\
&= [\Gamma, N \triangleright \text{le} := \text{exp} : \text{com}}\eta \\
&\quad (\Gamma, N \triangleright (\text{val } x : T \cdot c)(e[l]) \text{ end} : \text{com}}\eta \psi) \\
&= [\Gamma, N \triangleright \text{le} := \text{exp} ; (\text{val } x : T \cdot c)(e[l]) \text{ end} : \text{com}}\eta \psi) \\
&= \text{by the semantics of sequential composition}
\end{align*}
\]

\[\text{Case } \text{var } x : T \cdot c \text{ end.}\]

\[
\begin{align*}
&= \text{exp} \neq \text{error} \land (\Gamma, N \triangleright (\text{var } x : T \cdot c \text{ end})[\text{exp}] : \text{com}}\eta \psi \\
&= [\Gamma, N \triangleright \text{le} := \text{exp} ; (\text{var } x : T \cdot c \text{ end})[\text{exp}] : \text{com}}\eta \psi) \\
&= [\Gamma, N \triangleright \text{le} := \text{exp} ; (\text{var } x : T \cdot c \text{ end})[\text{exp}] : \text{com}}\eta \psi) \\
&= \text{by the semantics of sequential composition}
\end{align*}
\]
Case $avar x : T \bullet c$ end: similar. □

Proof F.2. Law $\langle \text{var dec separation} \rangle$

\[
\begin{align*}
\Gamma, N \triangleright var \; x, x' : T, T' \bullet c \text{ end } : \text{com} \eta \psi \\
= \forall x, x' : T, T' \bullet [\Gamma; x, x' : T, T', N \triangleright c : \text{com}] \eta \psi & \quad \text{[by the semantics of local blocks]} \\
= \forall x : T \bullet \forall x' : T' \bullet [\Gamma; x, x' : T, T', N \triangleright c : \text{com}] \eta \psi & \quad \text{[by predicate calculus]} \\
= \forall x : T \bullet [\Gamma; x : T, N \triangleright \text{var } x' : T' \bullet c \text{ end } : \text{com}] \eta \psi & \quad \text{[by the semantics of local blocks]} \\
= [\Gamma, N \triangleright \text{var } x : T \bullet \text{var } x' : T' \bullet c \text{ end } : \text{com}] \eta \psi & \quad \text{[by the semantics of local blocks]}
\end{align*}
\]

Lemma 15 If $l$ is not free in $c$ and $\psi$; $x$ is not on the left-hand side of assignments and it is not a method call target, result argument, nor it occurs in the frame of specification statements in $c$, then

\[
[\Gamma, N \triangleright c : \text{com}] \eta \psi = ([\Gamma; l : T, N \triangleright c[l/x] : \text{com}] \eta \psi)[x/l]
\]

Proof By induction.

Case $y := e$, with $y \neq x$

\[
\begin{align*}
[\Gamma, N \triangleright y := e : \text{com}] \eta \psi \\
&= e \neq \text{error} \land \psi[e/y] & \quad \text{[by the semantics of assignment]} \\
&= (e \neq \text{error})[l/x][x/l] \land \psi[e[l/x]/y][l/x][x/l] & \quad \text{[by a property of substitution]} \\
&= (e[l/x] \neq \text{error})[x/l] \land \psi[e[l/x]/y][l/x][x/l] & \quad \text{[by a property of substitution]} \\
&= (e[l/x] \neq \text{error} \land \psi[e[l/x]/y][x/l]) & \quad \text{[by a property of substitution]} \\
&= ([\Gamma; l : T, N \triangleright y := e[l/x] : \text{com}] \eta \psi)[x/l] & \quad \text{[by the semantics of assignment]} \\
&= ([\Gamma; l : T, N \triangleright (y := e)[l/x] : \text{com}] \eta \psi)[x/l] & \quad \text{[by a property of substitution]}
\end{align*}
\]

Case $y : [\psi_1, \psi_2]$, with $y \neq x$

\[
\begin{align*}
[\Gamma; y : T, N \triangleright y : [\psi_1, \psi_2] : \text{com}] \eta \psi \\
&= \psi_1 \land (\forall y : T \bullet \psi_2 \Rightarrow \psi) & \quad \text{[by the semantics of specification statement]} \\
&= \psi_1 \land (\forall y : T \bullet \psi_2 \Rightarrow \psi) & \quad \text{[by the hypothesis and predicate calculus]}
\end{align*}
\]
APPENDIX F. PROOFS OF LAWS OF COMMANDS

\[ = \psi_1 \land (\forall y : T \bullet \psi_2 \Rightarrow \psi[x/l]) \quad \text{[by a property of substitution]}\]
\[ = \psi_1[l/x][x/l] \land (\forall y : T \bullet \psi_2[l/x][x/l] \Rightarrow \psi[x/l]) \quad \text{[by a property of substitution]}\]
\[ = \psi_1[l/x][x/l] \land (\forall y : T \bullet \psi_2[l/x] \Rightarrow \psi)[x/l] \quad \text{[by a property of substitution]}\]
\[ = (\psi_1[l/x] \land (\forall y : T \bullet \psi_2[l/x] \Rightarrow \psi))[x/l] \quad \text{[by a property of substitution]}\]
\[ = ([\Gamma; y : T, N \triangleright y : [\psi_1[l/x], \psi_2[l/x]] : \text{com}]\eta \psi)[x/l] \quad \text{[by the semantics of specification statement]}\]
\[ = ([\Gamma; y : T, N \triangleright (x : [\psi_1, \psi_2])[l/x] : \text{com}]\eta \psi)[x/l] \quad \text{[by a property of substitution]}\]

Case \( c, c' \)

\[ [\Gamma, N \triangleright c, c' : \text{com}]\eta \psi \]
\[ = ([\Gamma, N \triangleright c : \text{com}]\eta ([\Gamma, N \triangleright c' : \text{com}]\eta \psi)) \quad \text{[by the semantics of sequential composition]}\]
\[ = ([\Gamma; l : T, N \triangleright c[l/x] : \text{com}]\eta ([\Gamma, N \triangleright c'[l/x] : \text{com}]\eta \psi))[x/l] \quad \text{[by the induction hypothesis]}\]
\[ = ([\Gamma; l : T, N \triangleright c[l/x] : c'[l/x] : \text{com}]\eta \psi)[x/l]] \quad \text{[by the semantics of sequential composition]}\]

Case if \( []i \bullet \psi_1 \rightarrow c_i \text{ fi} \)

\[ = [\Gamma, N \triangleright \text{if } []i \bullet \psi_i \rightarrow c_i \text{ fi} : \text{com}]\eta \psi \]
\[ = (\forall i \bullet \psi_i) \land (\land i \bullet \psi_i \Rightarrow [\Gamma, N \triangleright c_i : \text{com}]\eta \psi) \quad \text{[by the semantics of alternation]}\]
\[ = ([\Gamma; l : T, N \triangleright (\text{if } []i \bullet \psi_i \rightarrow c_i)[l/x] \text{ fi} : \text{com}]\eta \psi)[x/l] \quad \text{[by the induction hypothesis]}\]

Case rec \( Y \bullet c \text{ end} \)

\[ = [\Gamma, N \triangleright \text{rec } Y \bullet c \text{ end} : \text{com}]\eta \psi \]
\[ = [\Gamma, N \triangleright \mu Y \text{ c end} : \text{com}]\eta \psi \quad \text{[by the semantics of recursion]}\]
\[ = ([\Gamma; l : T, N \triangleright \text{µ}_Y c[l/x] \text{ end} : \text{com}]\eta \psi)[x/l] \quad \text{[by the induction hypothesis]}\]
\[ = ([\Gamma; l : T, N \triangleright (\text{rec } Y \bullet c \text{ end})[l/x] : \text{com}]\eta \psi)[x/l] \quad \text{[by semantics of recursion]}\]

Case var \( y : T \bullet c \text{ end} \)

\[ [\Gamma, N \triangleright \text{var } y : T \bullet c \text{ end} : \text{com}]\eta \psi \]
\[ = \forall y : T' \bullet [\Gamma; y : T, N \triangleright c : \text{com}]\eta \psi \quad \text{[by the semantics local blocks]}\]
\[ = (\forall y : T' \bullet [\Gamma; l : T; y : T, N \triangleright c[l/x] : \text{com}]\eta \psi)[x/l] \quad \text{[by the induction hypothesis]}\]
\[ = ([\Gamma; l : T, N \triangleright (\text{var } y : T \bullet c \text{ end})[l/x] : \text{com}]\eta \psi)[x/l]] \quad \text{[by the semantics of local blocks]}\]

Case avar \( y : T' \bullet c \text{ end} \): similar.
Case \( y.\text{z}.m(e), \) with \( y \neq x \)

\[
\begin{align*}
\Gamma; \ l : T, N & \triangleright y.\text{z}.m(e) : \text{com}\eta \psi \\
\Gamma; \ l : T, N & \triangleright (\eta N' m)((N')y.\text{z}, e) : \text{com}\eta \psi \\
\end{align*}
\]

(by the semantics of method call)

\[
\begin{align*}
\forall N' \leq T, N'' \bullet \ y.\text{z} & \text{ isExactly } N' \land \Gamma; \ l : T, N \triangleright ((\eta N' m)((N')l.\text{z}, e))(l/x) : \text{com}\eta \psi[l/x] \\
\end{align*}
\]

(by the induction hypothesis)

\[
\begin{align*}
\forall N' \leq T, N'' \bullet l.\text{z} & \text{ isExactly } N' \land \\
\Gamma; \ l : T, N & \triangleright (y.\text{z}.m(e))[l/x] : \text{com}\eta \psi[l/x] \\
\end{align*}
\]

(by the semantics of method calls)

\[
\square
\]

**Proof F.2.19** Law \( \langle \text{var block-val} \rangle \)

By induction.

Case \( y := e, \) with \( y \neq x \)

\[
\begin{align*}
\Gamma; \ N & \triangleright y := e : \text{com}\eta \psi \\
\forall l ; \ T & \bullet \ [\Gamma; \ N \triangleright y := e : \text{com}\eta \psi] \\
\forall l ; \ T & \bullet \ [\Gamma; \ l : T, N \triangleright (y := e[l/x]) : \text{com}\eta \psi[l/x] ] \\
\forall l ; \ T & \bullet \ [\Gamma; \ l : T, N \triangleright \eta := e[l/x] : \text{com}\eta \psi[l/x] ] \\
\forall l ; \ T & \bullet \ [\Gamma; \ l : T, N \triangleright l := x : \text{com}\eta \psi[l/x] ] \\
\end{align*}
\]

(by the semantics of assignment)

\[
\begin{align*}
\forall l ; \ T & \bullet \ [\Gamma; \ l : T, N \triangleright \eta := e[l/x] : \text{com}\eta \psi[l/x] ] \\
\Gamma; \ N & \triangleright \text{var } l ; \ T \bullet l := x : \text{end } \text{com}\eta \psi [\text{by the semantics of local blocks}]
\end{align*}
\]

Case \( y : [\psi_1, \psi_2], \) with \( y \neq x \)

\[
\begin{align*}
\Gamma; \ N & \triangleright y : [\psi_1, \psi_2] : \text{com}\eta \psi \\
\forall l ; \ T & \bullet \ [\Gamma; \ l : T, N \triangleright y : [\psi_1, \psi_2] : \text{com}\eta \psi ] \\
\forall l ; \ T & \bullet \ [\Gamma; \ l : T, N \triangleright \psi_1[l/x], \psi_2[l/x] : \text{com}\eta \psi[l/x] ] \\
\forall l ; \ T & \bullet \ [\Gamma; \ l : T, N \triangleright \psi_1[l/x], \psi_2[l/x] : \text{com}\eta \psi[l/x] ] \\
\forall l ; \ T & \bullet \ [\Gamma; \ l : T, N \triangleright \psi_1[l/x], \psi_2[l/x] : \text{com}\eta \psi[l/x] ] \\
\end{align*}
\]

(by the semantics of assignment)

\[
\begin{align*}
\forall l ; \ T & \bullet \ [\Gamma; \ l : T, N \triangleright \psi_1[l/x], \psi_2[l/x] : \text{com}\eta \psi[l/x] ] \\
\end{align*}
\]

(by the semantics of sequential composition)

\[
\begin{align*}
\Gamma; \ l : T, N & \triangleright \text{var } l ; \ T \bullet l := x : \text{end } \text{com}\eta \psi [\text{by the semantics of local blocks}]
\end{align*}
\]
Case $c; c'$

\[
\begin{align*}
&[\Gamma, N \triangleright c; c' : \text{com}] \eta \psi \\
=& \forall l : T \cdot [\Gamma; l : T, N \triangleright c; c' : \text{com}] \eta \psi \quad \text{[by the hypothesis and predicated calculus]} \\
=& \forall l : T \cdot ([\Gamma; l : T, N \triangleright c[l/x]; c'[l/x] : \text{com}] \eta \psi)[x/l] \quad \text{[Lemma 15]} \\
=& \forall l : T \cdot x \neq \text{error} \land ([\Gamma; l : T, N \triangleright c[l/x]; c'[l/x] : \text{com}] \eta \psi)[x/l] \quad \text{[by the hypothesis]} \\
=& \forall l : T \cdot [\Gamma; l : T, N \triangleright l := x : \text{com}] \eta \psi (\Gamma; l : T, N \triangleright c[l/x]; c'[l/x] : \text{com}] \eta \psi) \quad \text{[by the semantics of assignment]} \\
=& \forall l : T \cdot [\Gamma; l : T, N \triangleright l := x; c[l/x]; c'[l/x] : \text{com}] \eta \psi \quad \text{[by the semantics of sequential composition]} \\
=& [\Gamma, N \triangleright \text{var } l : T \cdot l := x; c[l/x]; c'[l/x] \text{ end} : \text{com}] \eta \psi \quad \text{[by the semantics of local blocks]}
\end{align*}
\]

Case if $\text{[}\text{[}\text{if } i \cdot \psi_1 \rightarrow c_i \text{ fi}}$

\[
\begin{align*}
=& [\Gamma, N \triangleright \text{if } \text{[}\text{[}i \cdot \psi_1 \rightarrow c_i \text{ fi} : \text{com}] \eta \psi \\
=& \forall l : T \cdot [\Gamma; l : T, N \triangleright \text{if } [\text{[}i \cdot \psi_1 \rightarrow c_i \text{ fi} : \text{com}] \eta \psi \quad \text{[by the hypothesis and predicated calculus]} \\
=& \forall l : T \cdot ([\Gamma; l : T, N \triangleright \text{if } [\text{[}i \cdot \psi_1 \rightarrow c_i \text{ fi} : \text{com}] \eta \psi)[x/l] \quad \text{[by Lemma 15]} \\
=& \forall l : T \cdot x \neq \text{error} \land ([\Gamma; l : T, N \triangleright \text{if } [\text{[}i \cdot \psi_1 \rightarrow c_i \text{ fi} : \text{com}] \eta \psi)[x/l] \\
=& \forall l : T \cdot [\Gamma; l : T, N \triangleright \text{if } \text{[}\text{[}i \cdot \psi_1 \rightarrow c_i \text{ fi} : \text{com}] \eta \psi \quad \text{[by the semantics of assignment]} \\
=& \forall l : T \cdot [\Gamma; l : T, N \triangleright \text{var } l : T \cdot l := x; \text{[}\text{[}i \cdot \psi_1 \rightarrow c_i \text{ fi} : \text{com}] \eta \psi \quad \text{[by the semantics of sequential composition]} \\
=& [\Gamma; l : T, N \triangleright \text{var } l : T \cdot l := x; \text{[}\text{[}i \cdot \psi_1 \rightarrow c_i \text{ fi} : \text{com}] \eta \psi \quad \text{[by the semantics of local blocks]}
\end{align*}
\]

Case rec $Y \cdot c$ end

\[
\begin{align*}
=& [\Gamma, N \triangleright \text{rec } Y \cdot c \text{ end} : \text{com}] \eta \psi \\
=& \forall l : T \cdot [\Gamma; l : T, N \triangleright \text{rec } Y \cdot c \text{ end} : \text{com}] \eta \psi \quad \text{[by the hypothesis and predicated calculus]} \\
=& \forall l : T \cdot ([\Gamma; l : T, N \triangleright \text{rec } Y \cdot c \text{ end}] \eta \psi)[x/l] \quad \text{[by Lemma 15]} \\
=& \forall l : T \cdot x \neq \text{error} \land ([\Gamma; l : T, N \triangleright \text{rec } Y \cdot c \text{ end}] \eta \psi)[x/l] \quad \text{[by the hypothesis]} \\
=& \forall l : T \cdot [\Gamma; l : T, N \triangleright \text{rec } Y \cdot c \text{ end}] \eta \psi (\Gamma; l : T, N \triangleright \text{rec } Y \cdot c \text{ end}] \eta \psi) \quad \text{[by the semantics of assignment]} \\
=& \forall l : T \cdot [\Gamma; l : T, N \triangleright \text{rec } Y \cdot c \text{ end}] \eta \psi \quad \text{[by the semantics of assignment]} \\
=& \forall l : T \cdot [\Gamma; l : T, N \triangleright \text{rec } Y \cdot c \text{ end}] \eta \psi \quad \text{[by the semantics of assignment]} \\
\end{align*}
\]
Case \( y : T' \cdot c \) end

\[
[\Gamma, N \triangleright \var y : T' \cdot c \ \text{end} : \text{com}] \eta \psi
\]

[by the hypothesis and predicate calculus]

\[
= \forall l : T \cdot ([\Gamma ; l : T, N \triangleright (\var y : T' \cdot c \ \text{end})[l/x]] : \text{com} \eta \psi)[x/l]
\]

[by Lemma 15]

\[
= \forall l : T \cdot x \neq \text{error} \land ([\Gamma ; l : T, N \triangleright (\var y : T' \cdot c \ \text{end})[l/x]] : \text{com} \eta \psi)[x/l]
\]

[by the hypothesis]

\[
= \forall l : T \cdot [\Gamma ; l : T, N \triangleright l := x : \text{com}] \eta ([\Gamma ; l : T, N \triangleright (\var y : T' \cdot c \ \text{end})[l/x]] : \text{com} \eta \psi)
\]

[by the semantics of assignment]

\[
= \forall l : T \cdot [\Gamma ; l : T, N \triangleright l := x ; (\var y : T' \cdot c \ \text{end})[l/x] \ \text{end} : \text{com}] \eta \psi
\]

[by the semantics of sequential composition]

[by the semantics of local blocks]

Case \( \var y : T' \cdot c \) end: similar.

Case \( y.\exists.m(e) \), with \( e \) as a value argument.

\[
= [\Gamma ; l : T, N \triangleright y.\exists.m(e) : \text{com}] \eta \psi
\]

[by the hypothesis and predicate calculus]

\[
= \forall l : T \cdot [\Gamma ; l : T, N \triangleright y.\exists.m(e) : \text{com}] \eta \psi
\]

[by Lemma 15]

\[
= \forall l : T \cdot x \neq \text{error} \land ([\Gamma ; l : T, N \triangleright (y.\exists.m(e))[l/x]] : \text{com} \eta \psi)[x/l]
\]

[by the hypothesis]

\[
= \forall l : T \cdot [\Gamma ; l : T, N \triangleright l := x : \text{com} \eta ([\Gamma ; l : T, N \triangleright (y.\exists.m(e))[l/x]] : \text{com} \eta \psi)
\]

[by the semantics of assignment]

\[
= \forall l : T \cdot [\Gamma ; l : T, N \triangleright l := x ; (y.\exists.m(e))[l/x] \ \text{end} : \text{com} \eta \psi
\]

[by the semantics of sequential composition]

[by the semantics of local blocks]

\[ \square \]

**Proof F.2.20 Law \( \langle \var block-res \rangle \)**

By induction.

Case \( x := e \)
First, we assume that $x$ does not get $\text{error}$ in the specification statement.

\[
[\Gamma, N \triangleright x := e : \text{com}] \eta \psi = \psi_1 \land (\forall x : T \cdot \psi_2 \Rightarrow \psi) \quad \text{[by the semantics of specification statement]}
\]

\[
= \forall l : T \cdot \psi_1 \land (\forall x : T \cdot \psi_2 \Rightarrow \psi) \quad \text{[by predicate calculus]}
\]

\[
= \forall l : T \cdot \psi_1 \land (\forall l : T \cdot \psi_2 \Rightarrow (x \neq \text{error} \land \psi)) \quad \text{[by assumption]}
\]

\[
= \forall l : T \cdot \psi_1[l/x] \land (\forall l : T \cdot \psi_2[l/x] \Rightarrow (l \neq \text{error} \land \psi[l/x])) \quad \text{[by predicate calculus]}
\]

\[
= \forall l : T \cdot [\Gamma; l : T, N \triangleright l : [\psi_1[l/x], \psi_2[l/x]] : \text{com}] \eta (l \neq \text{error} \land \psi[l/x]) \quad \text{[by the semantics of specification statement]}
\]

\[
= \forall l : T \cdot [\Gamma; l : T, N \triangleright l : [\psi_1[l/x], \psi_2[l/x]] : \text{com}] \eta (l \neq \text{error} \land \psi[l/x]) \quad \text{[by the semantics of sequential composition]}
\]

\[
= [\Gamma, N \triangleright \text{var} l : T \cdot l : [\psi_1[l/x], \psi_2[l/x]] ; x := l : \text{com}] \eta \psi \quad \text{[by the semantics of local blocks]}
\]

Case $x : [\psi_1, \psi_2]$

The case in which $x$ gets $\text{error}$ in the specification statement result in program abortion. As $l$ replaces $x$, in the specification statement, and $l$ is assigned to $x$, if $l$ get $\text{error}$ the program aborts.
The other cases are consequence of the induction hypothesis.

\[\square\]

**Proof F.2.21 Law \(\langle \text{pcom elimination-val} \rangle\)**

\[
\begin{align*}
[\Gamma, N \triangleright (\text{val } vl : T \bullet c)(x) : \text{com}] \eta \psi \\
= [\Gamma, N \triangleright \text{var } l : T \bullet l := x; \ c[l/vl] \end : \text{com}] \eta \psi \\
&\quad \text{[by a syntactic transformation]}
\end{align*}
\]

\[\square\]

**Proof F.2.22 Law \(\langle \text{pcom elimination-res} \rangle\)**

\[
\begin{align*}
[\Gamma, N \triangleright (\text{res } vl : T \bullet c)(x) : \text{com}] \eta \psi \\
= [\Gamma, N \triangleright \text{var } l : T \bullet c[l/vl]; \ x := l \end : \text{com}] \eta \psi \\
&\quad \text{[by a syntactic transformation]}
\end{align*}
\]

\[\square\]
Appendix G

Proof of Laws of Classes

In this appendix we present the proofs of the programming laws related to classes. We concentrate on proof in the dynamic semantics of ROOL. So, we assume programs in the proof to be well-typed. Proofs in the static semantics of ROOL would involved verifying against the grammar of ROOL and typing rules.

G.1 Normal Form Laws

G.1.1 Class Declaration

Lemma 16 Consider a class \( N \), and typing environments \( \Gamma \) and \( \Gamma' \) such that \( N \in \Gamma.\mathit{cnames} \) and \( \Gamma'.\mathit{cnames} = \Gamma.\mathit{cnames} \setminus \{ N \} \). Let \( \eta \) and \( \eta' \) be environments compatible with \( \Gamma \) and \( \Gamma' \), respectively. If \( N \) is not used in \( \Gamma \) in anywhere, except in \( \mathit{cnames} \), nor in \( \eta \), then

\[
[\Gamma, A \triangleright c : \mathit{com}]\eta \psi = [\Gamma', A \triangleright c : \mathit{com}]\eta' \psi
\]

Proof Straightforward induction. Here, we consider only the case of a call to a method \( m \) that is not declared by \( N \). A call to a method of \( N \) is not well-typed.

Case \( \mathit{le}.m(e) \), with \( m \notin \text{dom}(\Gamma.\mathit{meth} N) \)

\[
[\Gamma, D \triangleright \mathit{le}.m(e) : \mathit{com}]\eta \psi
= (\forall N' \leq \Gamma.C \bullet \mathit{le} \ \mathbf{isExactly} \ N' \wedge [\Gamma, D \triangleright (\eta N' \ m)((N') \ \mathit{le}, e) : \mathit{com}]\eta \psi)
\]

[by the semantics of method call]

\[
= (\forall N' \leq \Gamma.C \bullet \mathit{le} \ \mathbf{isExactly} \ N' \wedge [\Gamma', D \triangleright (\eta N' \ m)((N') \ \mathit{le}, e) : \mathit{com}]\eta' \psi)
\]

[by the assumption]

\[
= [\Gamma', D \triangleright \mathit{le}.m(e) : \mathit{com}]\eta' \psi
\]

[by the semantics of method calls]

\[\square\]
Proof G.1.1 Law \textit{(class elimination)}

\[
\begin{align*}
[\emptyset; x : T \triangleright cdsx1 \bullet c : \text{program}] \eta \psi &= [\Gamma, \text{main} \triangleright c : \text{com}] \eta \psi & \text{[by the semantics of programs]} \\
&= [\Gamma', \text{main} \triangleright c : \text{com}] \eta' \psi & \text{[by Lemma 16]} \\
&= [\emptyset; x : T \triangleright cds \bullet c : \text{program}] \eta' \psi & \text{[by the semantics of programs]}
\end{align*}
\]

The environments \(\Gamma, \Gamma', \eta,\text{ and } \eta'\) are as in Lemma 16. The provisos of the law \textit{(class elimination)} guarantee that the programs are well-typed.

\[\square\]

G.1.2 Attribute Declaration

Proof G.1.2 Law \textit{(attribute elimination)}

Law \textit{(attribute elimination)} can be proved by applying law \textit{(private attribute-coupling invariant)}; the attribute \(a\) should be regarded as the abstract attribute, there should be no concrete attributes, and the coupling invariant should be true.

\[\square\]

Lemma 17 Let \(\Gamma\) and \(\Gamma'\) be typing environments such that \(\Gamma.\text{vis } N \ a = \text{prot}\) and \(\Gamma'.\text{vis } N = \Gamma.\text{vis } N \oplus \{a \mapsto \text{pub}\}\), but are otherwise identical.

\[
[\Gamma, N \triangleright c : \text{com}] \eta \psi = [\Gamma', N \triangleright c : \text{com}] \eta \psi
\]

Proof By induction. Since the semantics is defined with basis on the extended typing system, which does not enforce the visibility constraints, the difference between \(\Gamma\) and \(\Gamma'\) is irrelevant.

\[\square\]

Proof G.1.3 Law \textit{(change visibility: from protected to public)}

\[
\begin{align*}
[\emptyset; x : T \triangleright \text{class } C \text{ extends } D \text{ prot } a : T; \text{ ads ops end } cds \bullet c : \text{program}] \eta \psi &= [\Gamma, \text{main} \triangleright c : \text{com}] \eta \psi & \text{[by the semantics of programs]} \\
&= [\Gamma', \text{main} \triangleright c : \text{com}] \eta \psi & \text{[by Lemma 17 and the law proviso]} \\
&= [\emptyset; x : T \triangleright \text{class } C \text{ extends } D \text{ pub } a : T; \text{ ads ops end } cds \bullet c : \text{program}] \eta \psi & \text{[by the semantics of programs]}
\end{align*}
\]

\[\square\]
Lemma 18 Let $\Gamma$ and $\Gamma'$ be typing environments such that $\Gamma.vis N a = \text{pri}$ and $\Gamma'.vis N = \Gamma.vis N \oplus \{a \mapsto \text{pub}\}$, but are otherwise identical.

\[
[\Gamma, N \triangleright c : \text{com}] \eta \psi = [\Gamma', N \triangleright c : \text{com}] \eta \psi
\]

Proof By induction. Since the semantics is defined with basis on the extended typing system, which does not enforce the visibility constraints, the difference between $\Gamma$ and $\Gamma'$ is irrelevant.

Proof G.1.4 Law \langle change visibility: from private to public \rangle

\[
[\emptyset; x : T \triangleright \text{class } C \text{ extends } D \text{ pri } a : T; \text{ ads } \text{ ops } \text{ end } \text{ cds } \bullet c : \text{program}] \eta \psi
\]

\[
= [\Gamma, \text{main} \triangleright c : \text{com}] \eta \psi \quad \text{[by the semantics of programs]}
\]

\[
= [\Gamma', \text{main} \triangleright c : \text{com}] \eta \psi \quad \text{[by Lemma 18 and the law proviso]}
\]

\[
= [\emptyset; x : T \triangleright \text{class } C \text{ extends } D \text{ pub } a : T; \text{ ads } \text{ ops } \text{ end } \text{ cds } \bullet c : \text{program}] \eta \psi
\]

\[
= [\emptyset; x : T \triangleright \text{class } C \text{ extends } D \text{ pub } a : T; \text{ ads } \text{ ops } \text{ end } \text{ cds } \bullet c : \text{program}] \eta \psi
\]

Lemma 19 Let $\Gamma$ and $\Gamma'$ be typing environments such that $\Gamma'.\text{attr } N = \Gamma.\text{attr } N \oplus \{a \mapsto T\}$, but are otherwise identical. For a type $T''$, such that $T'' \leq_{\Gamma} T$, if every non-assignable occurrence of $a$ in expressions in $\eta$ is cast with type $T''$, every expression assigned to $a$ in the environment $\eta$ has type $T''$, and every use of $a$ as result argument in $\eta$ is for a formal parameter of type $T''$, such that $T'' \leq_{\Gamma} T$, and $T \leq_{\Gamma} T'$, then

\[
[\Gamma, N \triangleright c : \text{com}] \eta \psi = [\Gamma', N \triangleright c : \text{com}] \eta \psi
\]

Proof By induction. Here we present the cases of assignment and the application of a parameterised command with a result parameter to the argument $a$.

Case $a := e$

\[
[\Gamma, N \triangleright a := e : \text{com}] \eta \psi
\]

\[
= e \neq \text{error } \wedge \psi[e/a] \quad \text{[by the semantics of assignment]}
\]

\[
= [\Gamma', N \triangleright a := e : \text{com}] \eta \psi \quad \text{[by the semantics of assignment and } a := e \text{ is well-typed]}
\]

The assignment $a := e$ is well-type because the type of $a$ is changed to $T$ and the type of $e$ is $T''$, such that $T'' \leq_{\Gamma} T$, according to the hypothesis.
Case \((\text{res } x : T \cdot c)(a)\)

\[
[\Gamma, N \triangleright (\text{res } x : T \cdot c)(a) : \text{com}] \eta \psi = [\Gamma, N \triangleright \text{var } l : T \cdot l := a; c[l/x]; a := l \text{ end} : \text{com}] \eta \psi \quad \text{[by a syntactic transformation]}
\]

\[
= \forall l : T. [\Gamma; l : T, N \triangleright l := a; c[l/x]; a := l \text{ end} : \text{com}] \eta \psi \quad \text{[by the semantics of local blocks]}
\]

\[
= [\Gamma', N \triangleright \text{var } l : T \cdot l := a; c[l/x]; a := l \text{ end} : \text{com}] \eta \psi
\]

[by the semantics of local blocks and parameterised command application to \(a\) is well-typed]

\[
= [\Gamma', N \triangleright (\text{res } x : T \cdot c)(a) : \text{com}] \eta \psi \quad \text{[by a syntactic transformation]}
\]

The type of \(a\) is changed from \(T'\) to \(T\) and every use of \(a\) is for a corresponding formal parameter of type \(T\) or any subtype of \(T\), according the hypothesis. Consequently, the parameterised command application is well-typed.

\[\square\]

**Proof G.1.5 Law** \(\langle\text{change attribute type}\rangle\)

\[
[\emptyset; x : T_1 \triangleright \text{class } C \text{ extends } D \text{ pub } a : T'; \text{ ads ops end } cds \cdot c : \text{program}] \eta \psi
\]

\[
= [\Gamma, \text{main } \triangleright c : \text{com}] \eta \psi \quad \text{[by the semantics of programs]}
\]

\[
= [\Gamma', \text{main } \triangleright c : \text{com}] \eta \psi \quad \text{[by Lemma \[19\] ]}
\]

\[
= [\emptyset; x : T_1 \triangleright \text{class } C \text{ extends } D \text{ pub } a : T; \text{ ads ops end } cds \cdot c : \text{program}] \eta \psi
\]

[by the semantics of programs]

The provisos of the law \(\langle\text{change attribute type}\rangle\) guarantee that the program is well-formed and the typing environments \(\Gamma\) and \(\Gamma'\) are as in the hypothesis of Lemma \[19\].

\[\square\]

**Lemma 20** Let \(\Gamma\) and \(\Gamma'\) be typing environments such that

\[
\Gamma'.\text{attr } B = \Gamma.\text{attr } B \oplus \{a \mapsto T\}
\]

and

\[
\Gamma'.\text{vis } B = \Gamma.\text{vis } B \oplus \{a \mapsto \text{pub}\}
\]

but are otherwise identical. If \(a \notin \text{dom}(\Gamma.\text{attr } D)\) for any class \(D\) such that \(D \leq_\Gamma B, D \nsubseteq_\Gamma C,\) and \(C \leq_\Gamma B,\) then

\[
[\Gamma, N \triangleright c : \text{com}] \eta \psi = [\Gamma', N \triangleright c : \text{com}] \eta \psi
\]
Proof By induction. The environment $\Gamma'$ is relevant when dealing with accesses to the attribute $a$. For instance, when assigning to $a$. We consider this assignment through an expression of type $C$ or of any subtype of $C$.

Case $le.a := e$, with $le : M$ such that $M \leq_{\Gamma} C$

$$\begin{align*}
&[\Gamma, N \triangleright le.a := e : \text{com}] \eta \psi \\
= & [\Gamma, N \triangleright le := (le; a : e) : \text{com}] \eta \psi \\
= & (le; a : e) \neq \text{error} \land \psi[(le; a : e)/le] \\
= & [\Gamma', N \triangleright le := (le; a : e) : \text{com}] \eta \psi \\
= & [\Gamma', N \triangleright le.a := e : \text{com}] \eta \psi
\end{align*}$$

(by a syntactic transformation)

(by the semantics of assignment)

(by the semantics of assignment and the hypothesis)

(by a syntactic transformation)

The expression $le.a := e$ is well-typed because the attribute $a$ is still visible inside $C$ or any of its subclasses.

\[ \square \]

Lemma 21

\begin{align*}
[\emptyset; x : T_1 \triangleright & \text{class } B \text{ extends } A \text{ ads ops end} \\
& \text{class } C \text{ extends } B \text{ pub } a : T; \text{ ads' ops' end} \text{ cds } c : \text{program}] \psi \\
= & [\emptyset; x : T_1 \triangleright \text{class } B \text{ extends } A \text{ pub } a : T; \text{ ads ops end} \\
& \text{class } C \text{ extends } B \text{ ads' ops' end} \text{ cds } c : \text{program}] \psi
\end{align*}

Proof

\begin{align*}
[\emptyset; x : T_1 \triangleright & \text{class } B \text{ extends } A \text{ ads ops end} \\
& \text{class } C \text{ extends } B \text{ pub } a : T; \text{ ads' ops' end} \text{ cds } c : \text{program}] \psi \\
= & [\Gamma, \text{main } \triangleright c : \text{com}] \eta \psi \\
= & [\Gamma', \text{main } \triangleright c : \text{com}] \eta \psi \\
[\emptyset; x : T_1 \triangleright & \text{class } B \text{ extends } A \text{ pub } a : T; \text{ ads ops end} \\
& \text{class } C \text{ extends } B \text{ ads' ops' end} \text{ cds } c : \text{program}] \psi
\end{align*}

(by the semantics of programs)

(by Lemma 20)

(by the semantics of programs)

The proviso of the law, from left to right, guarantees that the program is well-formed and the typing environments $\Gamma$ and $\Gamma'$ are as in the hypothesis of Lemma 20.

\[ \square \]

Lemma 22 Let $\Gamma$ and $\Gamma'$ be typing environments such that

$$\Gamma'.\text{attr } B = \Gamma.\text{attr } B \setminus \{a \mapsto T\},$$
\( \Gamma'.\text{vis} \ B = \Gamma.\text{vis} \ B \setminus \{a \mapsto \text{pub}\} \),

and

\( \Gamma'.\text{attr} \ C = \Gamma.\text{attr} \ C \oplus \{a \mapsto T\} \), \( \Gamma'.\text{vis} \ C = \Gamma.\text{vis} \ C \oplus \{a \mapsto \text{pub}\} \),

but are otherwise identical. If \( le.a \) does not appear in \( \eta \), for any expression \( le \) of type \( D \), such that \( D \preceq \Gamma \ B \), \( D \prec \Gamma \ C \), and \( C \preceq \Gamma \ B \), then

\[
[\Gamma, N \triangleright c : \text{com}] \psi = [\Gamma', N \triangleright c : \text{com}] \psi
\]

\textbf{Proof} By induction.

\textbf{Proof} By induction. The environment \( \Gamma' \) is relevant when dealing with accesses to the attribute \( a \). For instance, when assigning to \( a \). We consider this assignment through an expression of type \( C \) or of any subtype of \( C \).

\textbf{Case} \( le.a := e \), with \( le : M \) such that \( M \preceq \Gamma \ C \)

\[
[\Gamma, N \triangleright le.a := e : \text{com}] \eta \psi
= [\Gamma, N \triangleright le := (le; a : e) : \text{com}] \eta \psi \quad \text{[by a syntactic transformation]}
= (le; a : e) \neq \text{error} \land \psi[(le; a : e) / le] \quad \text{[by the semantics of assignment]}
= [\Gamma', N \triangleright le := (le; a : e) : \text{com}] \eta \psi \quad \text{[by the semantics of assignment and the hypothesis]}
= [\Gamma', N \triangleright le.a := e : \text{com}] \eta \psi \quad \text{[by a syntactic transformation]}
\]

\[ \Box \]

\textbf{Lemma 23}

\[
[\emptyset; x : T_1 \triangleright \text{class} \ B \text{ extends } A \text{ pub } a : T; \text{ ads ops } \text{ end} \\
\text{class } C \text{ extends } B \text{ ads' ops' end cds } \bullet c : \text{ program}] \psi \\
= [\emptyset; x : T_1 \triangleright \text{class} \ B \text{ extends } A \text{ ads ops } \text{ end} \\
\text{class } C \text{ extends } B \text{ pub } a : T; \text{ ads' ops' end cds } \bullet c : \text{ program}] \psi
\]

\textbf{Proof}

\[
[\emptyset; x : T_1 \triangleright \text{class} \ B \text{ extends } A \text{ pub } a : T; \text{ ads ops } \text{ end} \\
\text{class } C \text{ extends } B \text{ ads' ops' end cds } \bullet c : \text{ program}] \psi \\
= [\Gamma, \text{main} \triangleright c : \text{com}] \eta \psi \quad \text{[by the semantics of programs]}
= [\Gamma', \text{main} \triangleright c : \text{com}] \eta \psi \quad \text{[by Lemma 22]}
\]
G.1. NORMAL FORM LAWS

= [∅: x : T₁ ⊳ class B extends A ads ops end
class C extends B pub a : T; ads' ops' end cds • c : program] ψ

[by the semantics of programs]

□

Proof G.1.6 Law (move attribute to superclass)
The proof of this law is given by Lemmas [21] and [23]

□

G.1.3 Method Declaration

Lemma 24 Consider and typing environment Γ, and let η and η' be environments such that

η B m = (vres me : B; Γ.meth B m • c),

η C m = (vres me : C; Γ.meth C m • c), and

η' C = η C ⊕ {m ↦ (vres me : C; Γ.meth C m • super.m(α(pds)))},

with C ≤ Γ B, and m ∉ (dom(Γ.meth C) \ dom(Γ.meth B)). For all other classes and methods, η and η' are equal. Then,

[Γ, N ⊢ c : com]η ψ = [Γ, N ⊢ c : com]η' ψ

Proof By induction. The case of calls to the method m is relevant as the definition for this method is being modified. We assume that the type of le has exact type C. The proof for the other cases follows from an argument that is similar.

Case le.m(e)

[Γ, D ⊢ le.m(e) : com]η ψ
= (∨₅N' ≤ Γ C • le isExactly N' ∧ [Γ, D ⊢ (η N' m) ((N')le, e) : com]η ψ

[by the semantics of method call]

Here, as said above, we concentrate on the case in which le is exactly an object of type C. This requires that le is not null or error.

= le ≠ null ∧ le ≠ error ∧ [Γ, D ⊢ (η C m) ((C)le, e) : com]η ψ
= [Γ, D ⊢ (η C m) ((C)le, e) : com]η ψ
By the construction of the environment, we know that $\eta \ C \ m$ contains a modified version of $m$.

$$
[\Gamma, \Delta \triangleright (\eta \ C \ m) ((C)le, e) : \text{com}] \eta \psi
$$

$$
= [\Gamma, \Delta \triangleright \mu (vres \ me : C; \ pds \bullet \ meI \Gamma D m) ((C)le, e) : \text{com}] \eta \psi \quad \text{[by the definition of $\eta$]}
$$

$$
= [\Gamma, \Delta \triangleright (vres \ me : C; \ pds \bullet \ meI \Gamma D m) ((C)le, e) : \text{com}] \eta \psi
$$

[by the unfold property of fixed points]

$$
= [\Gamma, \Delta \triangleright \text{var} \ me : C * \text{me} := (C)le; \ (pds \bullet \ meI \Gamma D m) (e); \ (C)le := \ me
$$

end : \text{com}] \eta \psi

[by a syntactic transformation]

$$
= [\Gamma, \Delta \triangleright (pds \bullet \ meI \Gamma D m) \text{[le/me](e) : com}] \eta \psi
$$

[by the semantics of programs]

$$
= [\Gamma, \Delta \triangleright \text{super.} \ m[\text{le/me}(e) : \text{com}] \eta \psi
$$

[by a syntactic transformation]

$$
= [\Gamma, \Delta \triangleright \text{super.} \ m \text{[e]} : \text{com}] \eta \psi
$$

$me$ is not free in $\text{super.} \ m$

$$
\square
$$

**Proof G.1.7** Law (*introduce method redefinition*)

$$
[\emptyset; \ x : T \triangleright \text{class } B \text{ extends } A \text{ ads } \text{meth } m \cong (pds \bullet c) \text{ ops end}
$$

$$
\text{class } C \text{ extends } B \text{ ads'} \text{ops'} \text{ end } \text{cds} \bullet c : \text{program}] \eta \psi
$$

$$
= [\Gamma, \text{main} \triangleright c : \text{com}] \eta \psi
$$

[by the semantics of programs]

$$
= [\Gamma, \text{main} \triangleright c : \text{com}] \eta' \psi
$$

[by Lemma 24]

$$
= [\emptyset; x : T \triangleright \text{class } B \text{ extends } A \text{ ads } \text{meth } m \cong (pds \bullet c) \text{ ops end}
$$

$$
\text{class } C \text{ extends } B \text{ ads'} \text{meth } m \cong (pds \bullet \text{super.} \ m(\alpha \text{pds})) \text{ ops'} \text{ end}
$$

$$
\text{cds} \bullet c : \text{program}] \eta' \psi
$$

[by the semantics of programs]

$$
\square
$$

**Lemma 25** Consider a typing environment $\Gamma$, and environments $\eta$ and $\eta'$ such that

$$
\eta \ B \ m = (vres \ me : B; \ pds \bullet \ meI \Gamma B m b)
$$

and

$$
\eta \ C \ m = (vres \ me : C; \ pds \bullet \ meI \Gamma C m b')
$$
Moreover,
\[
\eta' B m = (vres \ me : B; \ pds \bullet \ if \ \neg(me \ is \ C) \to \ me \Gamma \ B \ m \ b' \ \\
[me \ is \ C \to \ me \Gamma \ B \ m \ b' \fi)
\]

and
\[
\eta' C m = (vres \ me : C; \ pds \bullet \ if \ \neg(me \ is \ C) \to \ me \Gamma \ C \ m \ b' \\
[me \ is \ C \to \ me \Gamma \ C \ m \ b' \fi)
\]

For all other classes and methods, \(\eta\) and \(\eta'\) are equal. If \(B, C, b,\) and \(b'\) are as in law \(\langle\text{move redefined method to superclass}\rangle\), then, for all classes \(N\),
\[
[\Gamma, N \triangleright c : \text{com}] \eta \psi = [\Gamma, N \triangleright c : \text{com}] \eta' \psi
\]

**Proof** By induction. The different environments potentially affect the semantics of method calls. If the method called is not \(m\), then the semantics recorded in \(\eta\) and \(\eta'\) are the same, so the result is trivial. The case we consider below is that of a call to \(m\).

**Case** \( le.m(e) \)
\[
[\Gamma, N \triangleright le.m(e) : \text{com}] \eta \psi = \bigvee_{N' \leq \Gamma N''} le \ \text{isExactly} \ N'' \wedge \\
[\Gamma, N \triangleright (\eta N' m') ((N')le, e) : \text{com}] \eta \psi
\]

[semantics]

In this definition, \(le \ \text{isExactly} \ N''\) holds for exactly one subclass \(N''\) of the type \(N\) of \(le\). If \(N''\) is \(B\) or \(C\), then \(\eta\) and \(\eta'\) record different semantics for \(m\). Otherwise, the semantics are the same, and the result is trivial.

If the exact type of \(le\) is \(B\), we can proceed as follows.
\[
\bigvee_{N' \leq \Gamma N''} le \ \text{isExactly} \ N'' \wedge \\
[\Gamma, N \triangleright (\eta N' m') ((B)le, e) : \text{com}] \eta' \psi
\]

[assumption]
\[
= [\Gamma, N \triangleright (\eta B m) ((B)le, e) : \text{com}] \eta' \psi
\]

[semantics]
\[
= [\Gamma, N \triangleright vres \ me : B; \ pds \bullet \ meI \Gamma B m b)((B)le, e) : \text{com}] \eta' \psi
\]

[assumption]
\[
= [\Gamma, N \triangleright \text{var} \ me : B \bullet \\
me := (B)le; \ pds \bullet \ meI \Gamma B m b(e); \ (B)le := me \\
end : \text{com}] \eta' \psi
\]
Some of the above steps are justified by properties of commands. They are formalised by standard command laws.

If the exact type of \( le \) is \( C \), the proof is similar.

\[ \square \]

**Proof G.1.8** Law \( \langle \text{move redefined method to superclass} \rangle \)

\[
[\emptyset; x : T \triangleright \\
\text{class } B \text{ extends } A \\
\quad ads \\
\quad \text{meth } m \equiv (pds \bullet b); \ ops \\
\text{end} \\
\text{class } C \text{ extends } B \\
\quad ads' \\
\quad \text{meth } m \equiv (pds \bullet b') \\
\quad ops' ]
\]
G.1. NORMAL FORM LAWS

end
cds • c : program]η ψ
= [Γ, main ⊢ c : com]η ψ
= [Γ, main ⊢ c : com]η′ ψ
[∅; x : T ⊢
class B extends A

class C extends B

cds • c : program]η ψ

[by the semantics of programs]

[by Lemma 25]

The semantics of super is given by a copy-rule, but, since super is not present in b′ and ops′, the different definitions of m do not affect the result of applying such a rule. The typing environments defined by the programs are the same, since the methods available in the classes B and C are the same in both of them. The provisos guarantee that they are well-typed.

The environments η and η′ defined by the programs in law \{move redefined method to superclass\} are as in Lemma 25. The only final detail is that, in Lemma 25, we did not consider the fact that, if a subclass of C does not redefine m, then its semantics in the environment is also affected by the change. This generalisation of Lemma 25 is rather lengthier, but its proof is similar to that presented above.

Lemma 26 Let Γ and Γ′ be typing environments such that Γ′.meth N = Γ.meth N \ {m ↦ pds′}, but are otherwise identical. Consider also environments η and η′ such that
η′ N = η N \ {m ↦ (vres me : N; pds′ • meI Γ′ N m c′)}. If m is not used in η, then

[Γ, N ⊢ (η N′ m′) ((N′)le, e) : com]η ψ = [Γ′, N ⊢ (η′ N′ m′) ((N′)le, e) : com]η ψ

Proof By induction. The environments Γ′ and η′ are relevant when dealing with method calls. In this case, the called method cannot call the method that is being removed.
Case \( le.m'(e) \), with \( m' \neq m \)

\[
[\Gamma, N \triangleright le.m'(e) : com] \eta \psi \\
= (\forall N' \leq_N \cdot le \ isExactly \ N'' \wedge [\Gamma, N \triangleright (\eta N' m') ((N')le, e) : com] \eta \psi
\]

[by the semantics of method call]

\[
= (\forall N' \leq_N \cdot le \ isExactly \ N'' \wedge [\Gamma', N \triangleright (\eta' N' m') ((N')le, e) : com] \eta' \psi
\]

[by the hypothesis]

\[
= [\Gamma', N \triangleright le.m'(e) : com] \eta' \psi
\]

[by the semantics of method call]

\[\Box\]

**Proof G.1.9 Law** \( \langle method \ elimination \rangle \)

\[
[\emptyset; x : T \triangleright class \ C \ extends \ D \ ads; \ \text{meth} \ m \triangleq pc \ end; \ \text{ops} \ end \ \text{cds} \bullet \ c : \text{program}] \eta \psi
\]

\[
= [\Gamma; \ \text{main} \triangleright c : \text{com}] \eta \psi
\]

[by the semantics of programs]

\[
= [\Gamma'; \ \text{main} \triangleright c : \text{com}] \eta' \psi
\]

[by Lemma 26]

\[
= [\emptyset; x : T \triangleright class \ C \ extends \ D \ ads; \ \text{ops} \ end \ \text{cds} \bullet \ c : \text{program}] \eta \psi
\]

[by the semantics of programs]

The provisos of the law \( \langle method \ elimination \rangle \) guarantee that the program is well-formed and the environments \( \eta \) and \( \eta' \) are as in the hypothesis of Lemma 26.

\[\Box\]

**Lemma 27** Consider that \( B \) and \( C \) are classes such that \( C \leq_B B \), and let \( \Gamma, \Gamma', \eta, \) and \( \eta' \) are environments such that

\[
\eta \ C \ m = (vres \ me : C; \ pds \bullet \ meI \Gamma \ C \ m \ c),
\]

\[
\Gamma'.meh \ B = \Gamma.meh \ B \oplus \{m \mapsto pds\},
\]

and

\[
\eta' \ B = \eta \ B \oplus \{m \mapsto (vres \ me : B; \ pds \bullet c)\},
\]

but are otherwise identical. If

\[
m \notin \text{dom}(\Gamma.meh A), \text{ for any } A, \text{ such that } B \leq_B A, \text{ and } m \notin \text{dom}(\Gamma.meh B),
\]

\[
\Gamma.meh \ C \ m = \Gamma.meh \ D \ m, \text{ for any } D, \text{ such that } D \leq_B B \text{ and } D \not\leq_B C,
\]

and
G.1. NORMAL FORM LAWS

\[ m \not\in \text{dom}(\Gamma'.\text{meth}\ C \setminus \Gamma'.\text{meth}\ B), \]

then

\[ [\Gamma, N \triangleright c : \text{com}]_\eta \psi = [\Gamma', N \triangleright c : \text{com}]_{\eta'} \psi \]

**Proof** By induction.

The environments \( \eta \) and \( \eta' \) are relevant when dealing with method calls. They potentially affect the semantics of method calls. If the method called is not \( m \), then the semantics recorded in \( \eta \) and \( \eta' \) are the same; thus the result is trivial. We consider the case of a call to \( m \). The other cases are straightforward.

**Case** \( \text{le}.m(e) \)

\[
[\Gamma, N \triangleright \text{le}.m(e) : \text{com}]_\eta \psi \\
= (\forall N' \leq \eta \cdot \text{le isExactly } N'' \land [\Gamma, N \triangleright (\eta N' m) ((N')le, e) : \text{com}]_\eta \psi \\
\quad \text{[by the semantics of method call]} \\
= (\forall N' \leq \eta' \cdot \text{le isExactly } N'' \land [\Gamma', N \triangleright (\eta' N' m) ((N')le, e) : \text{com}]_{\eta'} \psi \quad \text{[by hypothesis]} \\
= [\Gamma', N \triangleright \text{le}.m(e) : \text{com}]_{\eta'} \psi \\
\quad \text{[by the semantics of method call]}
\]

\[ \square \]

**Proof G.1.10** Law \langle move original method to superclass \rangle

\[
= [\emptyset; x : T \triangleright \text{class } B \text{ extends } A \text{ ads ops end} \\
\quad \text{class } C \text{ extends } B \text{ ads'} \text{ meth } m \equiv (pds \bullet c) \text{ ops' end} \\
\quad \text{cds} \bullet c : \text{program}]_\eta \psi \\
= [\Gamma, \text{main} \triangleright c : \text{com}]_\eta \psi \quad \text{[by the semantics of programs]} \\
= [\Gamma', \text{main} \triangleright c : \text{com}]_{\eta'} \psi \quad \text{[by Lemma 27]} \\
= [\emptyset; x : T \triangleright \text{class } B \text{ extends } A \text{ ads meth } m \equiv (pds \bullet c) \text{ ops end} \\
\quad \text{class } C \text{ extends } B \text{ ads' ops' end} \\
\quad \text{cds} \bullet c : \text{program}]_\eta \psi \\
\quad \text{[by the semantics of programs]}
\]

Since \( \text{super} \) is not present in \( (pds \bullet c) \) and \( m \) is not declared by \( B \) or any superclass of \( B \), we can move the definition of \( m \) to the superclass. The environments \( \Gamma, \Gamma', \eta, \) and \( \eta' \) are as defined in Lemma 27. The provisos guarantee that the environments \( \Gamma \) and \( \Gamma' \) are well-typed.

\[ \square \]
G.1.4 Parameter type

**Lemma 28** Let $\Gamma$ and $\Gamma'$ be typing environments such that $(\text{val } x : T'; \ pds) \in \Gamma$.meth $C$ $m$, and $(\text{val } x : T; \ pds) \in \Gamma'$.meth $C$ $m$, but are otherwise identical, where $T \leq \Gamma T'$. For a type $T''$, such that $T'' \leq \Gamma T$, if every non-assignable occurrence of $x$ in expressions in $\eta C m$ is cast with $T''$, every actual parameter associated to $x$ in $\eta C m$ is of type $T''$, every expression assigned to $x$ in the $\eta C m$ has type $T''$, and every use of $x$ as result argument in $\eta C m$ is for a formal parameter of type $T''$, then

$$[[\Gamma, C \triangleright c : \text{com}] \eta \psi = [[\Gamma', C \triangleright c : \text{com}] \eta \psi$$

**Proof** By induction. Here we present the cases of application of a parameterised command with a value parameter to the argument $a$, with $a$ of type $T$ or of any subtype of $T$

**Case** $(\text{val } x : T \bullet b)(a)$

$$[[\Gamma, C \triangleright (\text{val } x : T' \bullet b)(a) : \text{com}] \eta \psi$$

$$= [[\Gamma, C \triangleright \text{var } l : T' \bullet l := a; b[l/x] \text{ end : com}] \eta \psi$$

$$= \forall l : T' \bullet [[\Gamma; l : T', C \triangleright l := a; b[l/x] : \text{com}] \eta \psi$$

$$= \forall l : T \bullet [[\Gamma'; l : T, C \triangleright l := a; b[l/x] : \text{com}] \eta \psi$$

$$= [[\Gamma', C \triangleright \text{var } l : T \bullet l := a; b[l/x] \text{ end : com}] \eta \psi$$

$$= [[\Gamma', C \triangleright (\text{val } x : T \bullet b)(a) : \text{com}] \eta \psi$$

The semantics of a parameterised command does not depend on the type of its parameters, but on it being well-typed. Therefore, the main risk in the change of a parameter type is to render method calls ill-typed. If this is not the case, then the semantics of the calls depend on the value of the arguments. These are not changed in Lemma 28.

**Proof G.1.11 Law (change value parameter type)**

$$[[\emptyset; x : T_1 \triangleright \text{class } C \text{ extends } D \text{ ads } \text{meth } \triangleq (\text{val } x : T'; \ pds \bullet b) \text{ ops end}$$

$$c_ds \bullet c : \text{program}] \psi$$

$$= [[\Gamma; \text{main } \triangleright c : \text{com}] \eta \psi$$

$$= [[\Gamma'; \text{main } \triangleright c : \text{com}] \eta \psi$$

$$= [[\emptyset; x : T_1 \triangleright \text{class } C \text{ extends } D \text{ ads } \text{meth } \triangleq (\text{val } x : T; \ pds \bullet b) \text{ ops end}$$

$$c_ds \bullet c : \text{program}] \psi$$
The environments $\Gamma$ and $\Gamma'$ are as in Lemma 28. The provisos of the law guarantee that the programs are well-typed.

**Lemma 29** Let $\Gamma$ and $\Gamma'$ be typing environments such that $(\text{res } x : T; \text{pds}) \in \Gamma.\text{meth } C \text{ m}$, and $(\text{res } x : T'; \text{pds}) \in \Gamma'.\text{meth } C \text{ m}$, but are otherwise identical, where $T \leq \Gamma T'$. For a type $T''$, such that $T'' \leq \Gamma T$, if every non-assignable occurrence of $x$ in expressions in $\eta C m$ is cast with $T''$, and every actual parameter associated to $x$ in $\eta$ is of type $T'$ or a any supertype of $T'$, then

$$[[\Gamma, C \triangleright c : \text{com}] \eta \psi = [[\Gamma', C \triangleright c : \text{com}] \eta \psi]$$

**Proof** By induction. Here we present the case of an application of a parameterised command with a result parameter to the argument $a$.

**Case** $(\text{res } x : T \bullet b)(le)$, with $le$ of type $T'$ or of any supertype of $T'$.

$$[[\Gamma, C \triangleright (\text{res } x : T \bullet b)(a) : \text{com}] \eta \psi$$

$$= [[\Gamma, C \triangleright \text{var } l : T \bullet b[l/x]; le := l \,:= \text{com}] \eta \psi \quad \text{[by a syntactic transformation]}$$

$$= \forall l : T' \bullet [[\Gamma'; l : T, C \triangleright b[l/x]; le := l \,:= \text{com}] \eta \psi \quad \text{[by the semantics of local blocks]}$$

$$= \forall l : T' \bullet [[\Gamma'; l : T', C \triangleright b[l/x]; le := l \,:= \text{com}] \eta \psi \quad \text{[by the hypothesis]}$$

$$= [[\Gamma', C \triangleright \text{var } l : T' \bullet b[l/x]; le := l \,:= \text{com}] \eta \psi \quad \text{[by the semantics of local blocks]}$$

$$= [[\Gamma', C \triangleright (\text{res } x : T' \bullet b)(le) : \text{com}] \eta \psi \quad \text{[by a syntactic transformation]}$$

The semantics of a parameterised command does not depend on the type of its parameters, but on it being well-typed. Therefore, the main risk in the change of a parameter type is to render method calls ill-typed. If this is not the case, then the semantics of the calls depend on the value of the arguments. These are not changed in Lemma 29.

**Proof** G.1.12 Law ⟨change result parameter type⟩

$$[[\emptyset; x : T_1 \triangleright \text{class } C \text{ extends } D \text{ ads } \text{meth} \triangleq (\text{res } x : T; \text{pds } \bullet b) \text{ ops end})$$

$$\text{cds } \bullet c : \text{program}] \eta \psi \quad \text{[by the semantics of programs]}$$

$$= [[\Gamma, \text{main } \triangleright c : \text{com}] \eta \psi \quad \text{[by the semantics of programs]}$$

$$= [[\Gamma', \text{main } \triangleright c : \text{com}] \eta \psi \quad \text{[by Lemma 29]}$$

$$= [[\emptyset; x : T_1 \triangleright \text{class } C \text{ extends } D \text{ ads } \text{meth} \triangleq (\text{res } x : T'; \text{pds } \bullet b) \text{ ops end})$$

$$\text{cds } \bullet c : \text{program}] \eta \psi \quad \text{[by the semantics of programs]}$$
The environments $\Gamma$ and $\Gamma'$ are as in Lemma 29. The provisos guarantee that the programs are well-typed.

\end{proof}

G.1.5 Method calls

**Proof G.1.13 Law** (eliminate super)

Let $\Gamma$ and $\eta$ be environments such that

$$\Gamma.\text{supcls } C = B,$$

and

$$\eta \ B \ m = (\text{vres } m : B; \ pds \bullet \ meI \ \Gamma \ B \ m \ c),$$

then

$$[[\Gamma, \ C \triangleright super.m(e) : \text{com}]_\eta \ \psi = [[\Gamma, \ C \triangleright (pds \bullet c)(e) : \text{com}]_\eta \ \psi$$

[by a syntactic transformation]

The proof of Law (eliminate super) is a direct consequence of the semantics of super, which is given by the copy rule. Dynamic binding does not apply in this case. Arguments to which super.m is applied remain the same, they are not touched by this law. As required in the law proviso, super and private attributes do not appear of the superclass of $C$ do not appear in the parameterised command that defines the method $m$.

\end{proof}

For the proofs of Lemmas 30, 31, and 32, we assume that the type of $le$ is $C$. In the law (method call elimination); it can be any of its subclasses, but the proof is the same.

**Lemma 30** If the method $m$ is not redefined in any subclass of $C$, then

$$\forall_{N' \leq C} le \text{ isExactly } N' \land [\Gamma, A \triangleright (\eta \ N' \ m) ((N')le, e) : \text{com}]_\eta =$$

$$\forall_{N' \leq C} le \text{ isExactly } N' \land [\Gamma, A \triangleright (\eta \ C \ m) ((C)le, e) : \text{com}]_\eta$$

**Proof** By the construction the environment $\eta$. 

\end{proof}
Lemma 31 For an environment \( \eta \) and a method \( m \) in this environment, we have that

\[
[\Gamma, A \triangleright (\eta \ C \ m)((C)le, e) : \text{com}] \eta \psi
= [\Gamma, A \triangleright \mu(vres \ me : C; \ pds \cdot meI \Gamma, A \ m \ c)((C)le, e) : \text{com}] \eta \psi
\]

Proof By the definition of the environment \( \eta \), \( \eta \ C \ m \) contains the method \( m \) modified with the extra value-result parameter \( me \).

Lemma 32

\[
[\Gamma, A \triangleright \mu(vres \ me : C; \ pds \cdot meI \Gamma, A \ m \ c)((C)le, e) : \text{com}] \eta \psi
= [\Gamma, A \triangleright (pds \cdot c)[le/me](e) : \text{com}] \eta \psi
\]

Proof

\[
[\Gamma, A \triangleright \mu(vres \ me : C; \ pds \cdot meI \Gamma, A \ m \ c)((C)le, e) : \text{com}] \eta \psi
= [\Gamma, A \triangleright (vres \ me : C; \ pds \cdot meI \Gamma, A \ m \ c)((C)le, e) : \text{com}] \eta \psi
\]

\[
= [\Gamma, A \triangleright \text{var} \ me : C \cdot
  \quad me := (C)le; \ (pds \cdot meI \Gamma, A \ m \ c)(e); \ (C)le := me
  \quad \text{end} : \text{com}] \eta \psi
\]

\[
= [\Gamma, A \triangleright (pds \cdot meI \Gamma, A \ m \ c)[le/me](e) : \text{com}] \eta \psi
\]

\[
= [\Gamma, A \triangleright (pds \cdot c)[le/self](e) : \text{com}] \eta \psi
\]

[by a syntactic transformation] [by the semantics] [by the definition of \( meI \)]

Proof G.1.14 Law \( \langle \text{method call elimination} \rangle \)

\[
[\Gamma, A \triangleright le.m(e) : \text{com}] \eta \psi
= (\forall N' \leq \Gamma \cdot C \cdot le \ \text{isExactly} \ N' \land [\Gamma, A \triangleright (\eta \ N' \ m)((N') \ le, e) : \text{com}] \eta \psi)
\]

[by the semantics of method call]

\[
= le \neq \text{null} \land le \neq \text{error} \land [\Gamma, A \triangleright (\eta \ C \ m)((C)le, e) : \text{com}] \eta \psi
\]

[by Lemma 30 and the semantics of \( \text{isExactly} \)]

\[
= le \neq \text{null} \land le \neq \text{error} \land [\Gamma, A \triangleright \mu(vres \ me : C; \ pds \cdot meI \Gamma, A \ m \ c)((C)le, e) : \text{com}] \eta \psi
\]

[by Lemma 31]

\[
= le \neq \text{null} \land le \neq \text{error} \land [\Gamma, A \triangleright (pds \cdot c)[le/self](e) : \text{com}] \eta \psi
\]

[by Lemma 32]
\begin{align*}
= le \neq \text{null} \land le \neq \text{error} \land (\text{true} \Rightarrow ([\Gamma, A \triangleright (\text{pds} \bullet c)[le/\text{self}])(e) : \text{com}]\eta \psi)) \\
= [\Gamma, A \triangleright : [le \neq \text{null} \land le \neq \text{error}, \text{true}] : \text{com}]\eta([\Gamma, A \triangleright (\text{pds} \bullet c)[le/\text{self}])(e) : \text{com}]\eta \psi) \\
= [\Gamma, A \triangleright : [le \neq \text{null} \land le \neq \text{error}, \text{true}] ; (\text{pds} \bullet c)[le/\text{self}])(e) : \text{com}]\eta \psi \\
= [\Gamma, A \triangleright \{le \neq \text{null} \land le \neq \text{error} \}; (\text{pds} \bullet c)[le/\text{self}])(e) : \text{com}]\eta \psi \\
\end{align*}

\[\Box\]

\section*{G.1.6 Casts}

\textbf{Proof G.1.15 Law \textit{(eliminate cast of method call)}}

If \(c ds, A \triangleright e : B, C \leq_\Gamma B\), and \(m \in \text{dom } \Gamma.\text{meth } B\) or \(m \in \text{dom } \Gamma.\text{meth } N\), such that \(B \leq_\Gamma N\), then

First, we assume that \(e \notin \{\text{null, error}\}\) and the object it holds is of type \(C\) or of any subtype of \(C\).

\[\begin{align*}
= ([\forall N' \leq_\Gamma N \bullet ((C) e) \text{Exactly } N' \land [\Gamma, D \triangleright (\eta \ N' \ m) ((N') ((C) e), e') : \text{com}]\eta \psi) \\
= ((C) e) \text{Exactly } C \land [\Gamma, A \triangleright (\eta \ C \ m)((C) (C) e), e') : \text{com}]\eta \psi \\
= \text{true} \land ((C) e) \text{Exactly } C \land [\Gamma, A \triangleright (\eta \ C \ m)((C) (C) e), e') : \text{com}]\eta \psi \\
= e \text{ is } C \land (e \text{ Exactly } C \land [\Gamma, A \triangleright (\eta \ C \ m)((C) e), e') : \text{com}]\eta \psi) \\
= e \text{ is } C \land (\Gamma, A \triangleright e \text{.m(e')} : \text{com}]\eta \psi) \\
= e \text{ is } C \land \text{true} \Rightarrow ([\Gamma, A \triangleright e \text{.m(e')} : \text{com}]\eta \psi) \\
= [\Gamma, A \triangleright : e \text{ is } C, \text{ true} ; e \text{.m(e')} : \text{com}]\eta \psi \\
= [\Gamma, A \triangleright \{e \text{ is } C\} ; e \text{.m(e')} : \text{com}]\eta \psi
\end{align*}\]

The case in which \(e \in \{\text{null, error}\}\) or \(e\) holds an object whose type is not \(C\) or a subtype
of $C$ is equivalent to program abortion, because if the value of a cast expression does not have the required type, it evaluation results in error, by the semantics of casts, and command \((C)\mathit{e}).\mathit{m}(\mathit{e}')\) aborts. If the value of $\mathit{e}$ is null, then the value of the boolean expression $\mathit{e} \text{ is } C$ is false. In this case, the assumption \{\mathit{e} is $C$\} behaves like abort. If the value of $\mathit{e}$ is not null or error, but it is an object that is not of type $C$ or of any subtype of $C$, the test $\mathit{e} \text{ is } C$ also gives false. Consequently, the assumption \{\mathit{e} is $C$\} is equivalent to abort.

\[\blacksquare\]

\textbf{Proof G.1.16 Law} \langle introduce trivial cast in expressions \rangle

If $\mathit{cds}, \mathit{A} \triangleright \mathit{e} : C$, then

\[
\begin{align*}
\quad &\quad \quad [\Gamma, \mathit{A} \triangleright \mathit{e} : \mathit{com}]\eta \psi \\
&= [\Gamma, \mathit{A} \triangleright (C)\mathit{e} \mathit{com}]\eta \psi &\text{[by the hypothesis and the semantics of casts]}
\end{align*}
\]

\[\blacksquare\]

\textbf{Proof G.1.17 Law} \langle eliminate cast of expressions \rangle

If $\mathit{cds}, \mathit{A} \triangleright \mathit{le} : B, \mathit{e} : B', C \leq_{\Gamma} B' \text{ and } B' \leq_{\Gamma} B$, then

First, we assume that $\mathit{e} \notin \{\text{null, error}\}$ and the object it holds is of type $C$ or of any subtype of $C$.

\[
\begin{align*}
\quad &\quad \quad [\Gamma, \mathit{A} \triangleright \mathit{le} := (C)\mathit{e} : \mathit{com}]\eta \psi \\
&= (C)\mathit{e} \neq \text{error} \land \psi[(C)\mathit{e}/\mathit{le}] &\text{[by the semantics of assignment]} \\
&= \text{true} \land (C)\mathit{e} \neq \text{error} \land \psi[(C)\mathit{e}/\mathit{le}] &\text{[by predicate calculus]} \\
&= (C)\mathit{e} \text{ is } C \land (C)\mathit{e} \neq \text{error} \land \psi[(C)\mathit{e}/\mathit{le}] &\text{[by the semantics of is and assumption]} \\
&= e \text{ is } C \land (e \neq \text{error} \land \psi[e/\mathit{le}]) &\text{[by predicate calculus]} \\
&= e \text{ is } C \land (\mathit{A} \triangleright \mathit{le} := e : \mathit{com})\eta \psi &\text{[by the semantics of casts and assumption]} \\
&= e \text{ is } C \land \mathit{A} \triangleright \mathit{le} := e : \mathit{com} \eta \psi &\text{[by predicate calculus]} \\
&= [\Gamma, \mathit{A} \triangleright \mathit{le} := e : \mathit{com}]\eta \psi &\text{[by the semantics of the specification statement]} \\
&= [\Gamma, \mathit{A} \triangleright \mathit{le} := e : \mathit{com}]\eta \psi &\text{[by the semantics of sequential composition]} \\
&= [\Gamma, \mathit{A} \triangleright \{e \text{ is } C\}; \mathit{le} := e : \mathit{com}]\eta \psi &\text{[by the definition of assumption]}
\end{align*}
\]

The comments presented in the proof of law \langle eliminate cast of method call \rangle are applicable here. 

\[\blacksquare\]

\textbf{G.1.7 Commands and expressions}

\textbf{Lemma 33} Let $\Gamma$ and $\Gamma'$ be typing environments such that
\[ \Gamma . \text{locals } x = T', \]

and

\[ \Gamma' . \text{locals } = \Gamma . \text{locals} \oplus \{ x \mapsto T \}, \]

but are otherwise identical, where \( T \leq \Gamma T' \). For a type \( T'' \), such that \( T'' \leq \Gamma T \), if every non-assignable occurrence of \( x \) in expressions is cast with \( T'' \), every expression assigned to \( x \) in the \( \eta \ C m \) has type \( T'' \), and every use of \( x \) as result argument in for a formal parameter of type \( T'' \), then

\[ [ \Gamma, A \triangleright c : \text{com}] \eta \psi = [ \Gamma', A \triangleright c : \text{com}] \eta \psi \]

**Proof** By induction.

**Case** \( x := e \), with \( e \) of type \( T \) or any subtype of \( T \).

\[
[ \Gamma, A \triangleright x := e : \text{com}] \eta \psi \\
= e \neq \text{error} \land \psi[e/x] \quad \text{[by the semantics of assignment]} \\
= [\Gamma', A \triangleright x := e : \text{com}] \eta \psi \quad \text{[by the hypothesis and the semantics of assignment]}
\]

The assignment \( x := e \) is well-type because the type of \( x \) is changed from \( T' \) to \( T \) and the type of \( e \) is \( T'' \), such that \( T'' \leq \Gamma T \), according to the hypothesis.

**Case** \( (\text{res } y : T' \bullet c)(x) \)

\[
[\Gamma, N \triangleright (\text{res } y : T' \bullet c)(x) : \text{com}] \eta \psi \\
= [\Gamma, N \triangleright \text{var } l : T' \bullet c[l/y]; x := l \text{ end : com}] \eta \psi \quad \text{[by a syntactic transformation]} \\
= \forall l : T' \bullet [\Gamma; l : T', N \triangleright c[l/y]; x := l : \text{com}] \eta \psi \quad \text{[by the semantics of local blocks]} \\
= \forall l : T' \bullet [\Gamma; l : T', N \triangleright c[l/y] : \text{com}] \eta ([\Gamma'; l : T', N \triangleright x := l : \text{com}] \eta \psi) \quad \text{[by the semantics of sequential composition]} \\
= \forall l : T \bullet [\Gamma'; l : T, N \triangleright c[l/y] : \text{com}] \eta ([\Gamma'; l : T, N \triangleright x := l : \text{com}] \eta \psi) \quad \text{[by the hypothesis]} \\
= \forall l : T \bullet [\Gamma'; l : T, N \triangleright c[l/y]; x := l : \text{com}] \eta \psi \quad \text{[by the semantics of sequential composition]} \\
= [\Gamma', N \triangleright \text{var } l : T \bullet c[l/y]; x := l \text{ end : com}] \eta \psi \quad \text{[by the semantics of local blocks]} \\
= [\Gamma', N \triangleright (\text{res } y : T \bullet c)(x) : \text{com}] \eta \psi \quad \text{[by a syntactic transformation]} 
\]
The case in which \( x \) is used as result argument is similar to that of assignment as in the semantics of a parameterised command with a result parameter, there is an assignment of a local variable, whose type is the same as the one of the result parameter, to the argument that is applied to the parameterised command. As required by the law, the type of the formal result parameter is \( T \) or any subtype of \( T \).

\[ \text{Proof G.1.18 Law } \langle \text{change variable type} \rangle \]

\[
\begin{align*}
\left[ \Gamma, A \triangleright \text{var } x : T' \bullet c \text{ end : com} \right] \eta \psi \\
= \forall x : T' \bullet \left[ \Gamma; x : T' \triangleright c : \text{com} \right] \eta \psi &[\text{by the semantics of local blocks}] \\
= \forall x : T \bullet \left[ \Gamma; x : T \triangleright c : \text{com} \right] \eta \psi &[\text{by Lemma} \ 33] \\
= \left[ \Gamma, A \triangleright \text{var } x : T \bullet c \text{ end : com} \right] \eta \psi &[\text{by the semantics of local blocks}]
\end{align*}
\]

The environments \( \Gamma \) and \( \Gamma' \) are as in Lemma 33. The provisos of the law guarantee that the programs are well-typed.

\[ \text{Proof G.1.19 Law } \langle \text{change angelic variable type} \rangle \]

The proof of law \( \langle \text{change angelic variable type} \rangle \) is similar to that of law \( \langle \text{change variable type} \rangle \).

\[ \text{Proof G.1.20 Law } \langle \text{is test true} \rangle \]

If \( N \leq_{\text{cds}} M \), then

\[
\begin{align*}
\left[ \Gamma, N \triangleright \text{self is } M : \text{com} \right] \eta \psi \\
= \left[ \Gamma, N \triangleright N \text{ is } M : \text{com} \right] \eta \psi &[\text{by typing of self}] \\
= \left[ \Gamma, N \triangleright \text{true : com} \right] \eta \psi &[\text{by the hypothesis and the semantics of is}]
\end{align*}
\]

\[ \text{Proof G.1.21 Law } \langle \text{is test false} \rangle \]

If \( N \nleq_{\text{cds}} M \) and \( M \nleq_{\text{cds}} N \), then

\[
\begin{align*}
\left[ \Gamma, N \triangleright \text{self is } M : \text{com} \right] \eta \psi \\
= \left[ \Gamma, N \triangleright N \text{ is } M : \text{com} \right] \eta \psi &[\text{by typing of self}] \\
= \left[ \Gamma, N \triangleright \text{false : com} \right] \eta \psi &[\text{by the hypothesis and the semantics of is}]
\end{align*}
\]
APPENDIX G. PROOF OF LAWS OF CLASSES

G.2 Further object-oriented programming laws

In Lemma [34] in what follows, we abuse of notation and write \( \eta[\text{new } A / \text{new } B] \) to denote that, in environment \( \eta \), expression \text{new } B is replaced with \text{new } A.

**Lemma 34** Let \( \Gamma \) be a typing environment in which \( \Gamma.\supcls B = A \), \( \Gamma.\attr B \setminus \Gamma.\attr A = \emptyset \), and \( \Gamma.\meth B \setminus \Gamma.\meth A = \emptyset \). Let \( \eta \) and \( \eta' \) be environments such that \( \eta' = \eta[\text{new } A / \text{new } B] \), then

\[
[\Gamma, D \triangleright e : \text{com}] \eta \psi = [\Gamma, D \triangleright e : \text{com}] \eta' \psi
\]

**Proof** By induction. Here, we consider the case of assignments and method calls. The other cases are consequence of the induction hypothesis.

**Case** \( x := e \). Assignment of \text{new } B is of special interest, since other expressions remain unchanged.

\[
[\Gamma, N \triangleright x := \text{new } B : \text{com}] \eta \psi = \text{new } B \neq \text{error} \land \psi[\text{new } B / x] \quad \text{[by the semantics of assignment]}
\]

\[
[\Gamma, N \triangleright x := \text{new } A : \text{com}] \eta' \psi \quad \text{[by the hypothesis and definition of } \eta' \text{]}
\]

\[
[\Gamma, N \triangleright x := \text{new } B : \text{com}] \eta \psi = [\Gamma, D \triangleright e : \text{com}] \eta' \psi \quad \text{[by the semantics of assignment]}
\]

**Case** \( le.m(e) \). We consider the case of a call on an object of dynamic type \( B \).

\[
[\Gamma, D \triangleright le.m(e) : \text{com}] \eta \psi
\]

\[
= \bigvee_{N' \leq \Gamma.N} le \text{ isExactly } N' \land [\Gamma, D \triangleright (\eta N' m) ((N')le, e) : \text{com}] \eta \psi \quad \text{[by the semantics of method call]}
\]

We consider the case in which \( le \) is exactly an object of type \( B \). Since \( m \) is not declared in \( B \), then, by the way the environment \( \eta \) is constructed, we can deduce that \( \eta B m \) is equal to \( \eta A m \), except for the type of the extra \( me \) parameter. The method \( m \) may be redefined in the subclasses of \( B \), but this is irrelevant for the proof.

\[
(\bigvee_{N' \leq \Gamma.N} le \text{ isExactly } N' \land [\Gamma, D \triangleright (\eta N' m) ((N')le, e) : \text{com}] \eta \psi
\]

\[
= le \text{ isExactly } B \land [\Gamma, D \triangleright (\eta B m) ((B)le, e) : \text{com}] \eta \psi \quad \text{[by assumption]}
\]

An object of exact type \( A \) also calls the same method that is called on an object of dynamic type \( B \), since class \( B \) redefines no methods. By the definition of the environment \( \eta' \) occurrences of \text{new } B are replaced with \text{new } A.

\[
le \text{ isExactly } B \land [\Gamma, D \triangleright (\eta B m) ((B)le, e) : \text{com}] \eta \psi
\]

\[
= le \text{ isExactly } A \land [\Gamma, D \triangleright (\eta A m) ((A)le, e) : \text{com}] \eta \psi \quad \text{[by definition of } \eta' \text{]}
\]

\[
= [\Gamma, D \triangleright le.m(e) : \text{com}] \eta \psi \quad \text{[by the semantics of method call]}
\]
Lemma 35 Let $\Gamma$ be a typing environment in which $\Gamma.\text{supcls} B = A$, $\Gamma.\text{attr} \setminus \Gamma.\text{attr} A = \emptyset$, and $\Gamma.\text{meth} \setminus \Gamma.\text{meth} A = \emptyset$. Let $\eta$, and $\eta'$ be environments such that $\eta' = \eta[\text{new} A/\text{new} B]$. If $c' \equiv c[\text{new} A/\text{new} B]$, then

$$[\Gamma, \text{main} \triangleright c : \text{com}] \eta \psi = [\Gamma, \text{main} \triangleright c' : \text{com}] \eta' \psi$$

Proof Similar to Lemma 34.

Proof G.2.1 Law $\langle \text{new superclass} \rangle$

$$[\emptyset; x : T \triangleright \text{class A extends C ads}_a \text{ mts}_a \text{ end} \\
\text{class B extends A end} \bullet c : \text{program}] \eta \psi
= [\Gamma, \text{main} \triangleright c : \text{com}] \eta \psi  \quad \text{[by the semantics of programs]}
= [\Gamma, \text{main} \triangleright c' : \text{com}] \eta' \psi  \quad \text{[by Lemmas 34 and 35]}
= [\emptyset; x : T \triangleright \text{class A extends C ads}_a \text{ mts}_a \text{ end} \\
\text{class B extends A end} \bullet c' : \text{program}] \eta' \psi  \quad \text{[by the semantics of programs]}
$$

The reason for using Lemma 35 in the proof of Law $\langle \text{new superclass} \rangle$ is that there may be occurrences of $\text{new} B$ in the main command $c$. Replacements of $\text{new} B$ with $\text{new} A$ in $c$ are not captured in Lemma 34 because class $\text{main}$ is not in $\Gamma.\text{cnames}$. The environments $\Gamma$, $\eta$, and $\eta'$ are as in Lemmas 34 and 35. The provisos of the law guarantee that the programs are well-typed.

Lemma 36 Consider a typing environment $\Gamma$ in which $\Gamma.\text{supcls} B = A$. Consider also an environment $\eta$ such that

$$\eta A m = (\text{vres} \ me : A; \text{pds} \bullet \text{meI} \ \Gamma \ A \ m \ c),$$

and

$$\eta B m = (\text{vres} \ me : B; \text{pds} \bullet \text{meI} \ \Gamma \ B \ m \ c),$$

for all method $m$, with $m \in \text{dom} (\Gamma.\text{meth} A)$ and $m \in \text{dom} (\Gamma.\text{meth} B)$. Let $\eta$ and $\eta'$ be environments such that $\eta' = \eta[\text{new} B/\text{new} A]$, but are otherwise identical, then

$$[\Gamma; D \triangleright c : \text{com}] \eta \psi = [\Gamma; D \triangleright c : \text{com}] \eta' \psi$$
Proof By induction. Here, we consider the case of assignments and method calls. The other cases are consequence of the induction hypothesis.

Case \( x := e \). Assignment of new \( A \) is of special interest, since other expressions remain unchanged.

\[
\begin{align*}
[\Gamma, N \triangleright x := \text{new } A : \text{com}] & \eta \psi \\
= & \text{new } A \neq \text{error} \wedge \psi[\text{new } A/x] & \text{[by the semantics of assignment]} \\
= & [\Gamma, N \triangleright x := \text{new } B : \text{com}] \eta' \psi & \text{[by the hypothesis and definition of } \eta' \text{]} \\
= & \text{new } B \neq \text{error} \wedge \psi[\text{new } B/x]
\end{align*}
\]

Case \( le.m(e) \). We consider the case of a call on an object of dynamic type \( A \).

\[
\begin{align*}
[\Gamma, D \triangleright le.m(e) : \text{com}] & \eta \psi \\
= & (\forall N' \leq N. \bullet le \text{ isExactly } N' \wedge [\Gamma, D \triangleright (\eta N' m) ((N')le, e) : \text{com}] \eta \psi & \text{[by the semantics of method call]} \\
= & le \text{ isExactly } A \wedge [\Gamma, D \triangleright (\eta A m) ((A)le, e) : \text{com}] \eta \psi & \text{[by assumption]}
\end{align*}
\]

Since \( m \) is not redefined in \( B \), then, by the way the environment \( \eta \) is constructed, we deduce that \( \eta B m \) is equal to \( \eta A m \), except for the type of the extra \( me \) parameter. The method \( m \) may be redefined in the subclasses of \( B \), but this is irrelevant for the proof.

An object of exact type \( A \) also calls the same method that is called on an object of dynamic type \( B \), since class \( B \) redefines no methods of \( A \). By the definition of the environment \( \eta' \) occurrences of new \( A \) are replaced with new \( B \).

\[
\begin{align*}
le \text{ isExactly } A \wedge [\Gamma, D \triangleright (\eta A m) ((A)le, e) : \text{com}] \eta \psi \\
= & le \text{ isExactly } B \wedge [\Gamma, D \triangleright (\eta B m) ((B)le, e) : \text{com}] \eta \psi & \text{[by definition of } \eta' \text{]} \\
= & [\Gamma, D \triangleright le.m(e) : \text{com}] \eta \psi & \text{[by the semantics of method call]}
\end{align*}
\]

Lemma 37 Consider a typing environment \( \Gamma \) in which \( \Gamma.\supcls B = A \). Consider also an environment \( \eta \) such that

\[
\eta A m = (vres \ me : A; \ pds \bullet meI \Gamma A m c),
\]

and

\[
\eta B m = (vres \ me : B; \ pds \bullet meI \Gamma B m c),
\]
for all method m, with $m \in \text{dom} (\Gamma.\text{meth } A)$ and $m \in \text{dom} (\Gamma.\text{meth } B)$. Let $\eta$ and $\eta'$ be environments such that $\eta' = \eta[\text{new } B/\text{new } A]$, but are otherwise identical. If $c' \equiv c[\text{new } B/\text{new } A]$, then

$$[\Gamma, \text{main} \triangleright c : \text{com}]\eta \psi = [\Gamma, \text{main} \triangleright c' : \text{com}]\eta' \psi$$

**Proof** Similar to Lemma 36.

---

**Proof G.2.2 Law $\langle \text{new superclass} \rangle$**

$$[\emptyset; x : T \triangleright \text{class } A \text{ extends } C \text{ ads } a \text{ mts } a \text{ end}
\text{ class } B \text{ extends } A \text{ end } \bullet c : \text{program}]\eta \psi$$

$$= [\Gamma, \text{main} \triangleright c : \text{com}]\eta \psi$$

[by the semantics of programs]

$$= [\Gamma, \text{main} \triangleright c' : \text{com}]\eta' \psi$$

[by Lemmas 36 and 37]

$$= [\emptyset; x : T \triangleright \text{class } A \text{ extends } C \text{ ads } a \text{ mts } a \text{ end}
\text{ class } B \text{ extends } A \text{ end } \bullet c' : \text{program}]\eta' \psi$$

[by the semantics of programs]

The reason for using Lemma 37 in the proof of aw $\langle \text{new subclass} \rangle$ is that there may be occurrences of $\text{new } A$ in the main command $c$. Replacements of $\text{new } A$ with $\text{new } B$ in $c$ are not captured in Lemma 36 because class $\text{main}$ is not in $\Gamma.\text{cnames}$. The environments $\Gamma$, $\eta$, and $\eta'$ are as in Lemmas 36 and 37. The provisos of the law guarantee that the programs are well-typed.

---

**G.2.1 Changing a superclass**

Lemma 38 in what follows states that, if the environment $\eta$ remains unchanged, the semantics of programs also remains unchanged, in spite of changes in the typing environment.

**Lemma 38** Let $\Gamma$ and $\Gamma'$ be environments such that a command $c$ is well-typed for both $\Gamma$ and $\Gamma'$ and a class $N$, and let $\eta$ be an environment compatible with both $\Gamma$ and $\Gamma'$. Then,

$$[\Gamma, N \triangleright c : \text{com}]\eta = [\Gamma', N \triangleright c : \text{com}]\eta$$

**Proof** Straightforward induction. 

---
Proof G.2.3 Law \( ⟨\text{change superclass: from object to any class}⟩ \)

\[
[\emptyset; x : T \triangleright \text{class } C \text{ extends object } ads_C mts_C \text{ end}]
\text{cds} \bullet c : \text{program}] \eta \psi
\]
\[
= [\Gamma, \text{main} \triangleright c : \text{com}] \eta \psi \quad \text{[by the semantics of programs]}
\]
\[
= [\Gamma', \text{main} \triangleright c : \text{com}] \eta' \psi \quad \text{[by Lemma 38]}
\]
\[
[\emptyset; x : T \triangleright \text{class } C \text{ extends } B \text{ ads}_C mts_C \text{ end}]
\text{cds} \bullet c : \text{program}] \eta \psi
\]

The environment \( \eta \) does not change. The provisos guarantee that the programs are well-typed.

\(\square\)

Proof G.2.4 Law \( ⟨\text{change superclass: from an empty class to immediate superclass}⟩ \)

\[
[\emptyset; x : T \triangleright \text{class } B \text{ extends } A \text{ end}]
\text{class } C \text{ extends } B \text{ ads}_C \text{ ops}_C \text{ end}
\text{cds} \bullet c : \text{program}] \eta \psi
\]
\[
= [\Gamma, \text{main} \triangleright c : \text{com}] \eta \psi \quad \text{[by the semantics of programs]}
\]
\[
= [\Gamma', \text{main} \triangleright c : \text{com}] \eta' \psi \quad \text{[by Lemma 38]}
\]
\[
[\emptyset; x : T \triangleright \text{class } B \text{ extends } A \text{ end}]
\text{class } C \text{ extends } A \text{ ads}_C \text{ ops}_C \text{ end}
\text{cds} \bullet c : \text{program}] \eta \psi
\]

The environment \( \eta \) does not change, as class \( B \) is an empty class. The provisos guarantee that the programs are well-typed.

\(\square\)

G.2.2 Class invariant

Proof G.2.5 Law \( ⟨\text{introduce class invariant}⟩ \)

From left to right.

For the proof of this law from left to right, we proceed with data refinement of class \( A \). The abstract and concrete variables are the same. The coupling invariant is the class invariant \( inv \).

Case \( x := e \)

\[
x := e
\]
= x : [x = e] [by law (simple specification)]
\preceq x : [inv, x = e \land inv] [by law (augment specification)]
= x : [inv, x = e]; [inv] [by law (absorb coercion), from right to left]
\subseteq x : [inv, x = e]; [inv]; \{inv\} [by law (introduce assumption)]
\subseteq x := e; \{inv\} [by law (assignment)]

Case \(x : [\psi_1, \psi_2]\)

\(x : [\psi_1, \psi_2]\)
\preceq x : [\psi_1 \land inv, \psi_2 \land inv] [by law (augment specification)]
= x : [\psi_1 \land inv, \psi_2]; [inv] [by law (absorb coercion), from right to left]
\subseteq x : [\psi_1 \land inv, \psi_2]; [inv]; \{inv\} [by law (introduce assumption)]
= x : [\psi_1 \land inv, \psi_2 \land inv]; \{inv\} [by law (absorb coercion), from left to right]
\subseteq x : [\psi_1, \psi_2]; \{inv\} [since inv is class invariant and \(w : [inv, inv] \subseteq x : [\psi_1, \psi_2]\)]

The last case, in terms of data refinement, is guard augmentation. In this case, we assume \(G \land inv\) to be \(G'\) in law (augment guard). This deals with the refinement of alternation.

For sequential composition, recall that data refinement distributes over it. For the case of method calls, we have to eliminate calls before applying data refinement. The case of application of parameterised commands is similar. The case of local variable blocks and recursion are also consequence of data refinement.

From right to left.

c; \{inv\}
\subseteq c [by laws (remove assumption) and \(\vdash \text{-skip unit}\)]
Appendix H

Typing Rules

\[
\begin{array}{l}
N \neq \text{main} \quad N' \in \Gamma.cnames \\
\Gamma, N \triangleright \text{self} : N \\
N' \in \Gamma.cnames \\
\Gamma, N \triangleright \text{null} : N'
\end{array}
\]

\[
T \in \Gamma.cnames \text{ or } T \text{ is a primitive type} \\
\Gamma, N \triangleright \text{error} : T
\]

\[
(\Gamma; x : T) \triangleright x : T \\
\Gamma \triangleright e : T' \\
\Gamma \triangleright f(e) : U
\]

\[
\begin{array}{c}
\Gamma, N \triangleright e : N' \\
\Gamma \triangleright \text{attr } N' \; x = T \\
\Gamma \triangleright e' : T'
\end{array}
\]

\[
\begin{array}{c}
\Gamma, N \triangleright (e; x : e') : N'
\end{array}
\]

Table H.1: Typing of Expressions
**APPENDIX H. TYPING RULES**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \triangleright e : \text{bool}$</td>
<td>$\Gamma \triangleright \psi_i : \text{pred}$ for all $i$</td>
</tr>
<tr>
<td>$\Gamma \triangleright e : \text{pred}$</td>
<td>$\Gamma \triangleright \psi_1 \Rightarrow \psi_2 : \text{pred}$</td>
</tr>
<tr>
<td>$\Gamma \triangleright \psi_1 : \text{pred}$</td>
<td>$\Gamma ; x : T \triangleright \psi : \text{pred}$</td>
</tr>
<tr>
<td>$\Gamma \triangleright (\forall i \cdot \psi_i) : \text{pred}$</td>
<td>$\Gamma \triangleright \forall x : T \cdot \psi : \text{pred}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma, N \triangleright e : N'$</td>
<td>$N'' \leq N'$</td>
</tr>
<tr>
<td>$\Gamma, N \triangleright e$ isExactly $N'' : \text{pred}$</td>
<td></td>
</tr>
</tbody>
</table>

Table H.2: Typing of Predicates

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(\Gamma; x : N''), N \triangleright \psi : \text{pred}$</td>
<td>$N'' \leq \Gamma N'$</td>
</tr>
<tr>
<td>$(\Gamma; x : N'), N \triangleright x , \text{isExactly} , N'' \land \psi : \text{pred}$</td>
<td></td>
</tr>
<tr>
<td>$(\Gamma; x : N''), N \triangleright \psi : \text{pred}$</td>
<td>$N'' \leq \Gamma N'$</td>
</tr>
<tr>
<td>$(\Gamma; x : N'), N \triangleright x , \text{isExactly} , N'' \Rightarrow \psi : \text{pred}$</td>
<td></td>
</tr>
<tr>
<td>$(\Gamma; x : N''), N \triangleright \psi : \text{pred}$</td>
<td>$N'' \leq \Gamma N'$</td>
</tr>
<tr>
<td>$(\Gamma; x : N'), N \triangleright \psi : \text{pred}$</td>
<td>$N'' \leq \Gamma N'$</td>
</tr>
<tr>
<td>$(\Gamma; x : N'), N \triangleright x , \text{is} , N'' \Rightarrow \psi : \text{pred}$</td>
<td></td>
</tr>
</tbody>
</table>

Table H.3: Coercion Rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(\Gamma; x : T) \triangleright c : \text{com}$</td>
<td>$\text{par} \in {\text{val, res, vres}}$</td>
</tr>
<tr>
<td>$\Gamma \triangleright (\text{par} x : T \cdot c) : \text{pcom}(\text{par} x : T)$</td>
<td>$(\Gamma; x : T) \triangleright (\text{pds} \cdot c) : \text{pcom}(\text{pds})$</td>
</tr>
<tr>
<td>$\Gamma \triangleright c : \text{com}$</td>
<td>$\text{par} \in {\text{val, res, vres}}$</td>
</tr>
<tr>
<td>$\Gamma \triangleright (\bullet c) : \text{pcom}()$</td>
<td>$\Gamma, N \triangleright \text{le} : N'$</td>
</tr>
<tr>
<td>$\Gamma, \text{meth} N' m = \text{pds}$</td>
<td></td>
</tr>
<tr>
<td>$\Gamma, N \triangleright \text{le.m} : \text{pcom}(\text{pds})$</td>
<td></td>
</tr>
</tbody>
</table>

Table H.4: Typing of Parameterised Commands
Table H.5: Typing of Commands

\[
\begin{array}{lll}
\Gamma \triangleright le : T & \Gamma \triangleright e : T' & T' \leq_{\Gamma} T \quad \text{sdisjoint } le \\
\hline
\Gamma \triangleright le := e : \text{com} & \hline
(\Gamma; x : T) \triangleright \psi_i : \text{pred} & i \in \{1, 2\} \quad \Gamma \triangleright (\bullet c) : \text{pcom()} \\
(\Gamma; x : T) \triangleright x : [\psi_1, \psi_2] : \text{com} & \Gamma \triangleright (\bullet c)() : \text{com} \hline
\Gamma \triangleright (\text{val } x : T \bullet c) : \text{pcom(val } x : T) & \Gamma \triangleright e : T' & T' \leq_{\Gamma} T \quad \text{sdisjoint } le \\
\hline
\Gamma \triangleright (\text{val } x : T \bullet c)(e) : \text{com} & \hline
\Gamma \triangleright (\text{res } x : T \bullet c) : \text{pcom(res } x : T) & \Gamma \triangleright le : T' & T' \leq_{\Gamma} T' \quad \text{sdisjoint } le \\
\hline
\Gamma \triangleright (\text{res } x : T \bullet c)(le) : \text{com} & \hline
\Gamma \triangleright (\text{vres } x : T \bullet c) : \text{pcom(vres } x : T) & \Gamma \triangleright le : T \quad \text{sdisjoint } le \\
\hline
\Gamma \triangleright (\text{vres } x : T \bullet c)(le) : \text{com} & \hline
\Gamma \triangleright (pd \bullet (pds \bullet c))(e_1)) (e_2) : \text{com} & \text{sdisjoint (rvrargs } (pd; pds) (e_2, e_1)) \\
\hline
\Gamma \triangleright (pd; pds \bullet c)(e_2, e_1) : \text{com} & \hline
\Gamma \triangleright le.m : \text{pcom(pds)} & \Gamma \triangleright e : T & \text{sdisjoint(le, rvrargs pds e)} \quad \text{apttype } \Gamma \ pds e T \\
\hline
\Gamma \triangleright le.m(e) : \text{com} & \hline
\Gamma \triangleright c : \text{com} & \Gamma \triangleright c' : \text{com} \quad \Gamma \triangleright \psi_i : \text{pred} & \Gamma \triangleright c_i : \text{com} \\
\hline
\Gamma \triangleright c; c' : \text{com} & \Gamma \triangleright \text{if } i \bullet \psi_i \rightarrow c_i \text{ fi : com} \\
\hline
\Gamma \triangleright Y : \text{com} \Rightarrow \Gamma \triangleright c : \text{com} & \hline
\Gamma \triangleright \text{rec } Y \bullet c \text{ end : com} & \hline
\end{array}
\]
Bibliography


Index

Laws
Classes
attribute elimination, 214
change angelic variable type, 221
change attribute type, 215
change result parameter type, 218
change superclass: from an empty class to immediate superclass, 224
change superclass: from object to any class, 223
change value parameter type, 218
change variable type, 220
change visibility: from private to public, 214
change visibility: from protected to public, 214
class elimination, 213
eliminate super, 219
eliminate cast of expressions, 220
eliminate cast of method call, 220
introduce class invariant, 224
introduce method redefinition, 216
introduce trivial cast in expressions, 220
is test false, 221
is test true, 221
method call elimination, 219
method elimination, 217
move attribute to superclass, 215
move original method to superclass, 217
move redefined method to superclass, 216
new subclass, 222

Commands
new superclass, 221

:= identity, 199
:= -▷ right dist, 202
:= skip, 199
:= symmetry, 199
; -:= combination, 202
; assoc, 201
▷ abort left zero, 201
▷ if left dist, 202
▷ if selection, 202
▷ miracle left zero, 201
▷ skip unit, 201
alternation absorb assumption, 206
assignment seq comp exp substitution, 208
assumption advance command, 207
assumption before or after command, 206
assumption distribution, 207
assumption guard, 206
assumption intro, 207
avar- := initial value, 204
avar-: dist, 204
avar elim, 203
avar- := final value, 204
avar-if dist, 204
avar-: left dist, 204
avar rename, 203
avar-: right dist, 204
avar symmetry, 203
avar - var relationship, 204
command refinement-class refinement, 38
   209
if abort unity, 200
if \(-\land distrib\), 200
if elim, 200
if false unity, 200
if gc elim, 201
if gc intro, 201
if identical guarded commands, 205
if \(-\lor distrib\), 200
if \(-\lor distrib2\), 200
if symmetry, 199
if true guard, 199
if true guard ref, 201
innocuous assumption-reading, 206
innocuous assumption-writing, 206
iteration guards, 205
new assumption, 207
order independent assignment, 207
pcom absorb assumption, 206
pcom elimination-res, 209
pcom elimination-val, 208
pcom merge, 209
rec fixed point, 201
repeated assignment, 207
var- := dist, 203
var- := initial value, 203
var- avar refinement, 205
var block absorb assumption, 206
var block-res, 208
var block-val, 208
var dec separation, 208
var elim, 202
var- := final value, 203
var- if dist, 203
var- := left dist, 203
var rename, 202
var- := right dist, 203
var symmetry, 202
weakening guards, 205
Data refinement and simulation
  Data refinement—variable blocks with initialisation, 35, 211
  private attribute-coupling invariant, 34, 225
  superclass attribute-coupling invariant, 34, 225
Morgan
  absorb coercion, 210
  assignment, 210
  augment assignment, 210
  augment guards, 210
  augment specification, 210
  diminish assignment, 210
  diminish specification, 210
  merge assumptions, 211
  merge coercions, 211
  remove assumption, 211
  simple specification, 211
  strengthen postcondition, 211
  weaken precondition, 211

Lemmas
  Eliminate super and Trivial Methods in Hierarchy, 194
  Eliminate super in Hierarchy, 193
  method call elimination-self, 187
  pcom result-argument, 197
  pcom value-argument, 196
  Program cast elimination, 196
  Pull Down Method in the Whole Hierarchy, 190
  Pull Up Method in the Whole Hierarchy, 189
  Self Non-Null Non-Error, 194
  Type changes in program, 195
List of Refactorings

4.1 ⟨Extract/Inline Method⟩  .............................................................................. 43
4.2 ⟨Move Method⟩  ......................................................................................... 48
4.3 ⟨Move Attribute⟩  ......................................................................................... 54
4.4 ⟨Pull Up/Push Down Method⟩ ................................................................. 62
4.5 ⟨Replace Parameter with Method⟩ ............................................................ 66
4.6 ⟨Extract Class⟩  ............................................................................................ 72
4.7 ⟨Clientship Elimination⟩ .............................................................. 76
4.8 ⟨Delegation Elimination⟩ ............................................................................. 81
4.9 ⟨Inline Class⟩  ............................................................................................... 82
4.10 ⟨Self Encapsulate Field⟩ ......................................................................... 84
4.11 ⟨Decompose Conditional⟩ ....................................................................... 85
4.12 ⟨Introduce Explaining Variable⟩ ............................................................... 86
4.13 ⟨Consolidate Conditional Expressions⟩ .................................................. 87
4.14 ⟨Consolidate Duplicate Conditional Fragments⟩ ..................................... 88
5.1 ⟨Change clientship: from subclass to superclass⟩ .................................... 93
5.2 ⟨Pull Up/Push Down Field⟩ .......................................................... 98
5.3 ⟨Extract Superclass⟩ ................................................................................. 102
5.4 ⟨Collapse Hierarchy - Superclass⟩ ............................................................. 106
5.5 ⟨Collapse Hierarchy - Subclass⟩ ............................................................... 111
5.6 ⟨Rename Method⟩ ...................................................................................... 115
5.7 ⟨Parameterise Method⟩ ............................................................................. 120
5.8 ⟨Encapsulate Field⟩ .................................................................................. 124
5.9 ⟨Add/Remove Parameter⟩ ....................................................................... 129
5.10 ⟨Separate Query from Modifier⟩ .............................................................. 130
5.11 ⟨Encapsulate Downcast⟩ .......................................................................... 131
6.1 ⟨Interface Clientship⟩ .............................................................................. 138
Tese de Doutorado apresentada por Márcio Lopes Cornélio a Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título "Refactorings como Refinamentos Formais", orientada pela Profa. Ana Lúcia Caneca Cavalcanti e aprovada pela Banca Examinadora formada pelos professores:

Prof. Paulo Henrique Monteiro Borba  
Departamento de Sistemas e Computação – CIn / UFPE

Prof. André Luís de Medeiros Santos  
Departamento de Sistemas e Computação – CIn / UFPE

Prof. Hermano Perrelli de Moura  
Departamento de Informação e Sistemas - CIn / UFPE

Prof. Arnaldo Vieira Moura  
Instituto de Computação / UNICAMP

Prof. Patrícia Duarte de Lima Machado  
Departamento de Computação e Sistemas / UFCG

Visto e permitida a impressão.  
Recife, 15 de março de 2004

Prof. Jackson Freire Brelaz de Castro  
Coordenador da Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco.