University of Magdeburg

School of Computer Science

Diplomarbeit

A Machine-Checked Proof for a Product-Line–Aware Type System

Author:

Thomas Thüm

January 15, 2010

Advisors:

Prof. Dr. rer. nat. habil. Gunter Saake
Dipl.-Wirt.-Inform. Christian Kästner

Department of Technical & Business Information Systems

Prof. Dr. rer. nat. habil. Jürgen Dassow

Department of Knowledge and Language Engineering

Thüm, Thomas:
A Machine-Checked Proof for a Product-Line–Aware Type System
Diplomarbeit, University of Magdeburg, 2009.

Abstract

A software product line is a set of software-intensive systems that share a common code base. Program variants are generated using a feature selection, where only the code for the selected features is generated. Since the number of program variants grows exponentially with the number of features, we cannot type-check each variant on its own. Recently, product-line–aware type systems were proposed to efficiently type-check the software product line. Type soundness proofs show that no ill-typed program variants can be generated from a well-typed product line. We present a formal type soundness proof for Colored Featherweight Java in Coq. Furthermore, we present a simplified type system for Colored Featherweight Java and share our experiences with machine-checked proofs using the proof assistant Coq.

Acknowledgements

I would like to thank Christian Kästner for the perfect assistance throughout the last three years. We had many productive discussions that I would not want to miss. His careful reading of my drafts and his critical view substantially helped to improve this thesis.

Many thanks to Jürgen Dassow who supported me in mathematical issues. He also gave me many helpful comments on earlier versions of this thesis.

Special thanks to the people who answered me on the Coq mailing list, especially Adam Chlipala, Cedric Auger, Bruno de Fraine, Edsko de Vries, and Pierre-Yves Strub. Without their assistance, many proofs would not be completed yet.

Finally, I would like to thank my fellow students Tom Brosch and Frederik Dornemann for their help with LaTeX and my family for their encouragement.

Contents

List of Figures x

List of Tables xi

List of Code Listings xiv

List of Acronyms xv

1 Introduction 1

2 Background 5

2.1 Software Product Lines 5
2.1.1 Feature Models 6
2.1.2 Variant Generation 7
2.1.3 CIDE 9
2.2 Type Systems 10
2.2.1 Type Checking 10
2.2.2 Type Soundness 11
2.3 Proof Assistant Coq 13
2.3.1 Gallina 13
2.3.2 Proof Tactics 15

3 Colored Featherweight Java 17

3.1 Featherweight Java 17
3.1.1 Syntax 18
3.1.2 Examples 19
3.1.3 Type System 21
3.2 Colored Featherweight Java 23
3.2.1 Syntax and Annotations 23
3.2.2 Examples 24
3.2.3 Variant Generation 27
3.2.4 Type System 27
3.3 Simplifications of the Type System 31
3.3.1 Casting and Field Access 31
3.3.2 Method Typing 33
3.3.3 Class Typing 33

3.3.4 Revised Type System . . . 35

4 Formalization of Colored Featherweight Java 39
4.1 Building on a FJ Formalization . . . 39
4.1.1 Choosing a FJ Formalization . . . 40
4.1.2 Corrections to the FJ Formalization . . . 40
4.2 Type System . . . 41
4.2.1 Realization of Annotations . . . 41
4.2.2 Subtyping and Auxiliary Rules . . . 44
4.2.3 Typing Rules . . . 47
4.3 Variant Generation . . . 52
4.4 Summary . . . 56

5 Type Soundness of Colored Featherweight Java 59
5.1 Type Soundness Theorem . . . 59
5.1.1 Formalization of the Theorem . . . 60
5.1.2 Splitting the Theorem . . . 61
5.2 Type Soundness Proof . . . 63
5.2.1 The Proof Strategy . . . 63
5.2.2 Problems with Induction . . . 64
5.2.3 Problems with Mutual Induction . . . 67
5.2.4 Further Problems . . . 70
5.3 Summary . . . 73

6 Experiences 75
6.1 Estimated Effort of Our Proof . . . 75
6.2 Challenges with Proof Assistants . . . 78
6.3 Advantages of Formal Proofs . . . 79
6.4 Summary . . . 80

7 Related Work 83

8 Conclusion 85

9 Future Work 87

A FJ Formalization in Coq 89
A.1 Incomplete Class Tables . . . 89
A.2 Method Overriding . . . 91

B CFJ Formalization in Coq 93
B.1 Verification of Our Formalization . . . 93
B.2 Annotation Lookup Functions . . . 94
B.3 Theorems, Lemmas, and Facts . . . 95

Bibliography 101

List of Figures

1.1 Development Costs with Software Product Line Engineering [JKB08] . 2

2.1 A Feature Model Representing a Graph Product Line 6
2.2 Generating Program Variants from a Common Code Base 8
2.3 The Domain of Graph Libraries 8
2.4 Annotations in CIDE [KAK08] 9
2.5 Type Checking a Single System 10
2.6 Type Checking a Software Product Line 11
2.7 Type Soundness Proofs using Coq 14

3.1 The Syntax of Featherweight Java [KA08] 19
3.2 A Minimal FJ Program 20
3.3 A FJ Program that Handles Pairs 20
3.4 Subtyping, Auxiliary, and Typing Rules for FJ 22
3.5 The Syntax of Colored Featherweight Java [KA08] 24
3.6 A CFJ Product Line that Handles Pairs 25
3.7 A CFJ Product Line with Pairs and Single Elements 26
3.8 The Feature Model for our Pair Product Line 26
3.9 The Variant Generation for CFJ [KA08] 27
3.10 Subtyping and Auxiliary Rules for CFJ [KAS] 28
3.11 Typing Rules for CFJ [KAS] 30
3.12 Our Revised Subtyping and Auxiliary Rules for CFJ 36
3.13 Our Revised Typing Rules for CFJ 37

4.1 Overview on Definitions for CFJ 57

4.2 Overview on the Variant Generation for CFJ 57

5.1 Theorems to Prove Generation Preserves Typing 61
5.2 Further Theorems to Prove Generation Preserves Typing 62

6.1 Estimated Effort in Formally Proving Type Soundness for CFJ 76
6.2 Estimated Effort for Each Part of the Formalization 77
6.3 Time Coq Needs to Check Our Formalization 77

A.1 An Example of an Incomplete Class Table 90
A.2 Method Overriding with Differing Return Types 91

B.1 Dependencies Between the FJ and the CFJ Formalization 94

List of Tables

1.1 Type System Usage in IDEs: Support for Efficient Development 2
1.2 Type System Usage in IDEs: Support for Error Detection 3

2.1 Mapping a Feature Model to a Propositional Formula 7

3.1 Meta-Variables in CFJ and Their Meaning 18

4.1 Annotation Lookup at the Class Table 45

B.1 The Files of Our Formalization 93

List of Code Listings

2.1 Inductive Definitions in Gallina 14

2.2 Recursive Functions and Definitions in Gallina 15

2.3 Proofs in Gallina 16

4.1 Formalization of Annotations 42

4.2 Terms in the FJ Formalization [Fra09a] 42

4.3 Formalization of Terms 43

4.4 Formalization of the Class Table 43

4.5 Formalization of Annotation Lookup 44

4.6 Formalization of Subtyping 45

4.7 Formalization of Field Lookup 46

4.8 Formalization of Method Lookup 46

4.9 Formalization of Valid Method Overriding 47

4.10 Reachability Checks for Valid Method Overriding 47

4.11 Formalization of Term Typing 48

4.12 Formalization of Method Typing 49

4.13 Reachability Checks for Method Typing 50

4.14 Formalization of Class Typing 50

4.15 Reachability Checks for Class Typing 51

4.16 Formalization of Well-Typed FJ Programs 52

4.17 Formalization of Well-Typed CFJ Product Lines 52

4.18 Formalization of Well-Typed CFJ Class Tables 52

4.19 Formalization of Parameter List Generation 54

4.20 Formalization of Term Generation 55

4.21 Formalization of Method Generation 55
4.22 Formalization of Class Table Generation 56
5.1 The Type Soundness Theorem—Generation Preserves Typing 60
5.2 Assumptions for Our Proofs 64
5.3 Variant Generation for Parameter Lists is Unique 65
5.4 Method Present with Existential Quantification 66
5.5 Method Present without Existential Quantification 66
5.6 Combined Scheme for Mutual Induction 67
5.7 Variant Generation for Terms is Unique using Mutual Induction 68
5.8 Variant Generation for Terms is Total 69
5.9 Generation Preserves Typing for Terms 70
5.10 Generation Preserves Typing for Terms using Mutual Induction 71
5.11 Variant Generation for Parameter Lists is a Total Function 72
5.12 Axioms of Our Type Soundness Proof 74
6.1 Example for a Mutual Induction Principle in Coq 79
A.1 Proof that an Incomplete FJ Class Table is Well-Typed 90
A.2 A Definition Allowing Incomplete Class Tables [Fra09a] 91
A.3 The Corrected Definition Prohibiting Incomplete Class Tables 91
A.4 A Definition Allowing Different Return Types [Fra09a] 92
A.5 The Corrected Definition Forcing Identical Return Types 92
B.1 Annotation Lookup Functions for CFJ 94
B.2 Theorems, Lemmas, and Facts to Prove Type Soundness for CFJ . . . 95

List of Acronyms

AT	Annotation Table
CFJ	Colored Featherweight Java
CIDE	Colored Integrated Development Environment
FCJ	Featherweight cJ
FFJ	Feature Featherweight Java
FFJ$_{PL}$	Feature Featherweight Java Product Line
FJ	Featherweight Java
FM	Feature Model
IDE	Integrated Development Environment
LFJ	Lightweight Feature Java
LJ	Lightweight Java
SPL	Software Product Line

1. Introduction

A continuous challenge in the field of software engineering is to improve software development techniques to build software faster and cheaper. It can be achieved through a better reuse of software artifacts, since all reused code does not have to be written over and over again. Obviously, this reduces the time needed to build new software and thus drops the development costs as well. State-of-the-art object-oriented programming provides concepts such as inheritance to enable the reuse of classes and avoid duplicated code [GHJV95]. But can we achieve even more software reuse?

Taking a closer look at today's software we do still find up to 20% duplicated code [Bak95, MLM96, KG06]. The reasons are manifold. For instance, we might have written a database management system for a particular device. A very similar database is needed for another device with stronger memory limitations. The probably fastest technique is to copy the code, remove all dispensable routines and adapt it for the new demand. In this way the code is kind of reused, but we run into problems with software maintenance and further adoptions [Joh93, Bak95], e.g., an error in duplicated code need to be fixed multiple times. Recent studies found out that inconsistent code duplicates are often a source for faults or misbehaviors [JDHW09]. Furthermore, how to combine the features of two or more database management system variants to create a new variant?

Software Product Lines

Software product lines aim at efficient development of similar programs using a single code base [PBvdL05]. A common experience is that object-oriented programming is insufficient on its own [KLM$^+$97, Pre97]. Therefore, on the one hand, object-oriented language extensions like aspect-oriented programming [KLM$^+$97] and feature-oriented programming [Pre97] were introduced. On the other hand, preprocessor statements [SH04] and annotations in terms of colors [KAK08] have been proposed. The overall idea is that of generated programming, functionalities of a software are represented in the code, so that a software developer can generate a program code according to special requirements, i.e., by selecting needed functionalities and a kind of software generator [CE00].

The main motivation to use software product line engineering instead of single system engineering is that the costs per software product are lower (see Figure 1.1). This is caused by software reuse strategies that aim on strong commonalities between programs of a certain domain. But, software product lines have a higher upfront investment. The use of software product line engineering amortizes when at least three or four single systems are involved [Beu03, PBvdL05, JKB08]. Hence, for developing more than three similar programs, we should take advantage of software product line engineering.

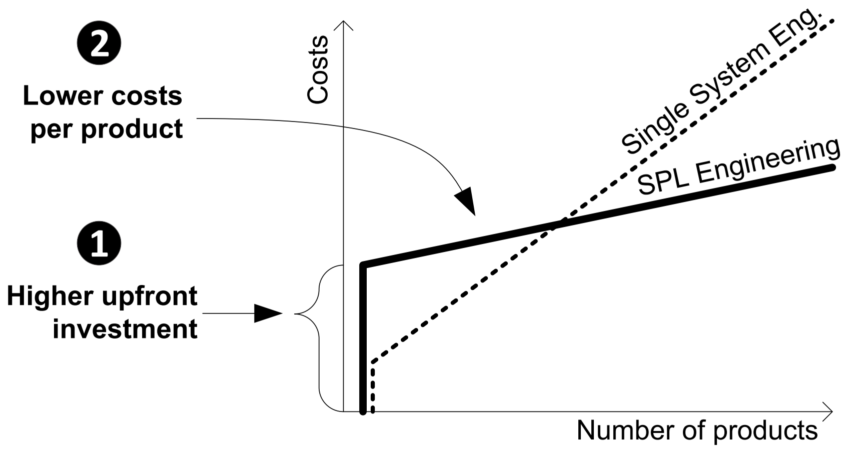


Figure 1.1: Development Costs with Software Product Line Engineering [JKB08]

Type Systems

Modern software engineering uses formal methods to ensure that systems behave correctly with respect to some specification of its desired behavior [Pie02]. A type system is such a formal method beside model checkers and run-time monitoring. Although a lot of software developers already profit from type systems, they do not necessarily know about them. Hence, we start with some examples what type systems are good for in Table 1.1. All screenshots are made in Eclipse Ganymede, a widely-used Integrated Development Environment (IDE).

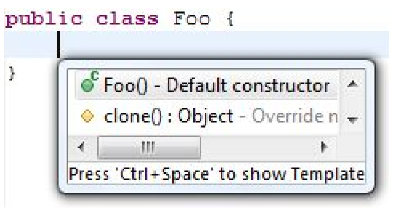
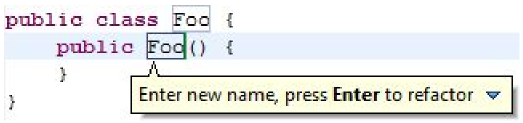
 `public class Foo {` `}` Foo() - Default constructor; clone() : Object - Override n; Press 'Ctrl+Space' to show Template	Content assist and auto completion
 `public class Foo {` `public Foo() {` `}` `}` Enter new name, press Enter to refactor	Refactorings: renaming identifiers and moving declarations

Table 1.1: Type System Usage in IDEs: Support for Efficient Development

First, a content assist provides context sensitive content for the position of the cursor. The content compromises local variables, fields, methods and elements from the super class. Essential is that only references available in the actual context are displayed. Second, we can simply rename an identifier by providing its new name. The IDE finds all occurrences of the name, even in other classes.

The support by type systems is not only for efficient development, it is also used for efficient debugging. Table 1.2 gives examples for errors a type system can detect.

`public class Foo {` `int a = 5 + "3";` `}`	Type mismatch: cannot convert from String to int
`public class Foo {` `void bla(){` `return; int a;` `}` `}`	Unreachable code
`public class Foo {` `void bla() {}` `void bla() {}` `}`	Duplicate method bla in type Foo

Table 1.2: Type System Usage in IDEs: Support for Error Detection

First, the addition of two numbers fails, if one of them is not encoded as a number but as a string. We get a localized type mismatch error at compile time. Without a type system, we would get an error at run time, but only if the class is instantiated, i.e., extensive testing is essential. Second, a type system can detect unreachable code and duplicate methods. It prevents from writing and maintaining unnecessary code.

All these examples point up how type systems can be used to improve the efficiency in software development. Using type systems for software product lines seems worthwhile, since we are interested in efficient development and many programmers are already used to have the functionalities of modern IDEs. In software product line engineering the program variants are generated from a new language or using a generation process, we can only apply an existing type system to the variants and not to the product line itself.

As the number of program variants increases possibly exponentially with the number of features that can be chosen, it is usually not suitable to check all program variants separately [CP06, TBKC07, KA08, AKGL09]. Hence, there is a need for product-line–aware type systems that can efficiently check a whole software product line and guarantee the absence of certain behaviors for all program variants that can be generated.

Machine-Checked Proofs

A type system is a formal method. Given a type system, we also need a proof of its correctness. Such a proof is called type soundness proof and is typically long and hard to verify by humans. One reason is that programming languages getting more and more complex what makes the proofs consisting of many cases.

A proof assistant is an environment to write a proof that can be verified by a machine. There are several reasons to decide to use such a machine-checked proof. First, we cannot simply forget cases or use assumptions that are not given. Second, we need to formalize all concepts, while informal concepts often lead to wrong conclusions. Third, having a machine-checked proof we can trust in the proof assistant and focus more on the theorems themselves.

Goal of this Thesis

The goal of this thesis is to provide a machine-checked type soundness proof of a product-line–aware type system. We decided to prove type soundness for Colored Featherweight Java (CFJ), because it already comes with a product-line–aware type system and a proof of its correctness in informal math [KA08]. For other languages, e.g., Feature Featherweight Java Product Line (FFJ_{PL}), only a proof sketch exists [AKGL09], what may indicate that writing a machine-checked proof is more complicated.

CFJ is a product-line–aware language based on a small functional subset of Java named Featherweight Java (FJ). Variability is achieved in CFJ by annotations, i.e., code fragments can be annotated with colors. A software generator can then produce a program variant by removing code annotated with certain colors.

The choice to do a proof manually or machine-checked is not obvious. Many mathematicians decline machine-checked proofs, as they read as a phone book and not like a mathematical proof. However, communities on proof assistants seem to grow and to give an example, there is still no manual proof of the Four Color Theorem stated in the year 1852, for which since 2004 a machine-checked proof exists [Gon04]. Therefore, a secondary goal of this thesis is that we want to share our experiences with machine-checked proofs.

Structure of the Thesis

Chapter 2 provides the necessary background on software product lines, type systems and the proof assistant Coq. Some further background on CFJ is given in Chapter 3 and we present a simplified type system for CFJ. Chapter 4 covers how we formalized CFJ in the proof assistant Coq. Based on this formalization, we present details on our machine-checked type soundness proof of CFJ in Chapter 5. We share our experiences on machine-checked proofs in Chapter 6. Chapter 7 presents related work. We give a conclusion and point to future work in Chapter 8 and Chapter 9.

2. Background

The subject of this thesis are machine-checked proofs for product-line–aware type system. It combines three subjects from computer science for which this chapter provides the necessary background. First, software product lines are used to efficiently develop similar programs (Section 2.1). Second, type systems syntactically prove the absence of certain undesired program behaviors (Section 2.2). Third, machine-checked proofs are formal proofs whose correctness can be verified by a computer program (Section 2.3).

2.1 Software Product Lines

Pohl et al. define software product line engineering as follows [PBvdL05].

> "*Software product line engineering* is a paradigm to develop software applications (software-intensive systems and software products) using platforms and mass customisation."

The definition covers the development of software as well as software-intensive systems, i.e., hardware systems that come along with software. Mass customization brings together the advantages of mass production and customized products. Applied to software development, the idea is to build programs for specific needs without developing each program from scratch. Instead, parts are build that can be reused.

Software product lines are widespread in today's software, while there is a couple of approaches for realization. The most popular approach is to use #ifdefs and C's preprocessor [SB00, AG01]. Arbitrary parts of the code can be surrounded by an #ifdef construct and a flag indicates the preprocessor to remove the code or not.

Usually, there is more than one flag in a software product line and so we can produce a number of software applications with different functionality. We use features to describe the commonalities and differences of these software variants. Kang et al. defined features as follows [KKL+98].

> "*Features* are any prominent and distinctive aspects or characteristics that are visible to various stakeholders, e.g., end-users, domain experts or developers."

Not all combinations of features are useful. Section 2.1.1 introduces feature models that describe the valid combinations of features. How we generate a variant given a valid combination of features is described in Section 2.1.2. Finally, Section 2.1.3 presents a tool to develop software product lines, on which our work partly relies on.

2.1.1 Feature Models

In 1990, feature models were introduced in the Feature-Oriented Domain Analysis by Kang et al. [KCH+90]. A *feature model* is a hierarchically organized set of features that is used as a compact representation of all possible program variants. The graphical representation of a feature model is a *feature diagram*. We give an example in Figure 2.1.

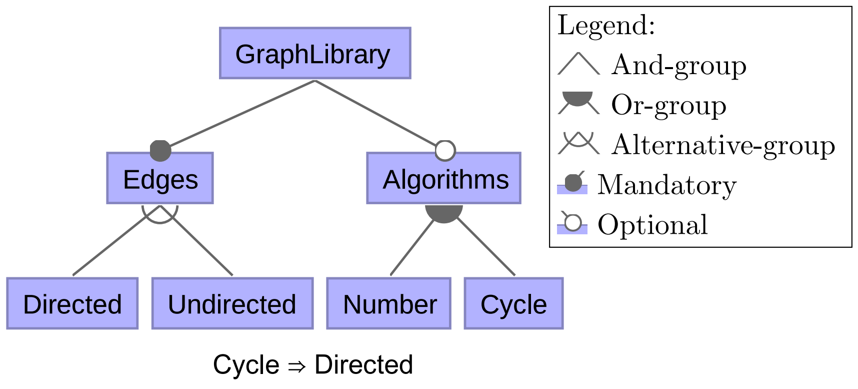


Figure 2.1: A Feature Model Representing a Graph Product Line

Every feature has a parent feature except for one feature that we call the *root feature*. Semantically, we want to express that whenever a feature is contained in a product, we will also find its parent in the same product. Usually, we distinguish between the three group types in that a feature is connected to its children [GFdA98, CE00].

And-groups have mandatory (filled circle) and *optional* features (empty circle). *Mandatory* features are always selected when their parent is selected. The semantic of *Alternative*-groups is that whenever the parent is selected, we have to choose exactly one of its children. *Or* means that we have to choose at least one of the children.

A feature diagram may also contain *cross-tree constraints*. Such constraints may express that one feature requires another or that two features mutually exclude each other. Cross-tree constraints are often drawn as dashed arrows in feature diagrams or written below the diagram.

In our example, a graph always has edges that are either directed or undirected. Our graph library might have algorithms to determine the number of edges or to detect a cycle. The cross-tree constraint states that the cycle detection requires the edges to be directed.

Propositional formulas can be used as a logical representation of feature models [Bat05]. For every feature we have a variable (usually with the same name) and assigning true to a variable means that the corresponding feature is selected. The propositional formula has the truth value true, if and only if the combination of selected features is valid. A feature model can be translated into a propositional formula using the rules given in Table 2.1.

Group Type	Propositional Formula
And	$(P \Rightarrow C_{k_1} \wedge \ldots \wedge C_{k_m}) \wedge (C_1 \vee \ldots \vee C_n \Rightarrow P)$
Or	$P \Leftrightarrow C_1 \vee \ldots \vee C_n$
Alternative	$(P \Leftrightarrow C_1 \vee \ldots \vee C_n) \wedge \text{atmost1}(C_1, \ldots, C_n)$

Table 2.1: Mapping a Feature Model to a Propositional Formula

P is a place holder for the parent feature, C_i for a child feature and n is the number of child features. $C_{k_1}, \ldots, C_{k_m}$ are mandatory features and the term $\text{atmost1}(C_1, \ldots, C_n)$ is equivalent to $\bigwedge_{1 \leq i < j \leq n} (\neg C_i \vee \neg C_j)$. The propositional formulas for each group are connected with the logical and. Additionally, we add all cross-tree constraints and the rule that the root feature is true in all configurations. For example, the propositional formula representing the feature model in Figure 2.1 is as follows:

$$\begin{aligned}
& \text{GraphLibrary} \\
\wedge\ & (\text{GraphLibrary} \Rightarrow \text{Edges}) \wedge (\text{Edges} \vee \text{Algorithms} \Rightarrow \text{GraphLibrary}) \\
\wedge\ & (\text{Edges} \Leftrightarrow \text{Directed} \vee \text{Undirected}) \wedge (\neg\text{Directed} \vee \neg\text{Undirected}) \\
\wedge\ & (\text{Algorithms} \Leftrightarrow \text{Number} \vee \text{Cycle}) \\
\wedge\ & (\text{Cycle} \Rightarrow \text{Directed}).
\end{aligned}$$

2.1.2 Variant Generation

Software product lines take advantage of generative programming, where software is generated from a common code base [CE00]. The software generator gets a valid configuration and a code base as input to generate a program variant (see solid arrows in Figure 2.2). The code base provides a mapping from features to code fragments, so that the generator can produce a program variant by combining the code for all selected features. The feature model describes the valid combinations of features, i.e., it is used to specify whether a given configuration is valid (dashed arrows). Generally, we have multiple configurations each leading to a different program variant.

A *configuration* (also feature selection) assigns a truth value to each variable representing a feature at the feature model. A feature is called *selected* in a particular

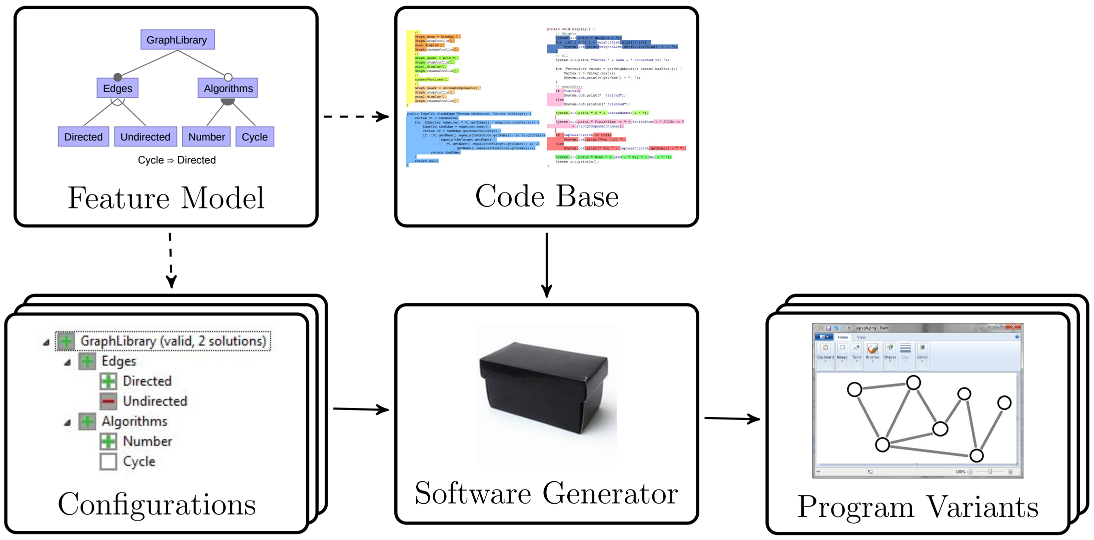


Figure 2.2: Generating Program Variants from a Common Code Base

configuration, if the value assigned to its variable is true. A configuration is called *valid* according to a feature model, if the propositional formula representing the feature model evaluates to true under that configuration. In Figure 2.3, we visualize all valid configurations of our example feature model in Figure 2.1.

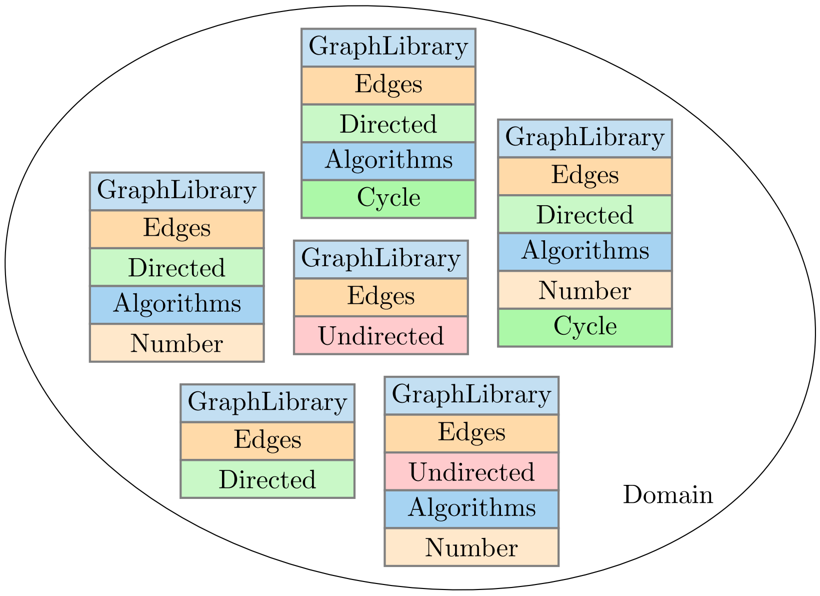


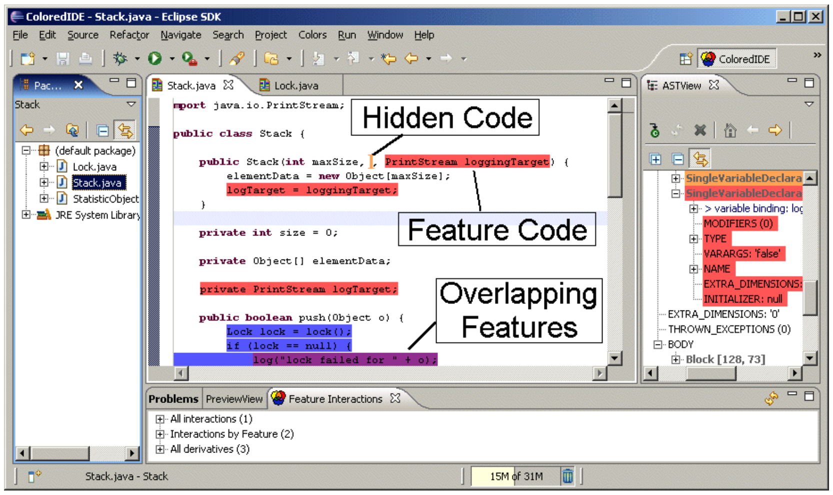
Figure 2.3: The Domain of Graph Libraries

Our sample domain of graph libraries contains six program variants (visualized as blocks). The program variants vary in their features (colored layers), e.g., there are two variants that can detect cycles in graphs and four variants that do not support this algorithm. There is no program variant with directed and undirected edges, because the according features are declared as alternatives at the feature diagram. Furthermore, a graph library with cycle checking on undirected graphs is not valid due to the cross-tree constraint.

In a software product line tool it can easily be determined whether a given configuration is valid using a satisfiability solver. It checks whether a given propositional formula is satisfiable or not. On the other hand, we can also check whether the code base with its mapping from features to code is valid according to the feature model. For this purpose, type checkers can be used to verify that all valid configurations specified by the feature model lead to well-typed program variants (see Section 2.2).

2.1.3 CIDE

In 2008, Kästner et el. presented the Colored Integrated Development Environment (CIDE) as an Eclipse-based prototype tool for software product line development. Compared to other software product line tools it especially focuses on "decomposing legacy applications into features that may have a fine granularity" [KAK08]. We give a screenshot in Figure 2.4.



```
mport java.io.PrintStream;

public class Stack {

    public Stack(int maxSize, , PrintStream loggingTarget) {
        elementData = new Object[maxSize];
        logTarget = loggingTarget;
    }

    private int size = 0;

    private Object[] elementData;

    private PrintStream logTarget;

    public boolean push(Object o) {
        Lock lock = lock();
        if (lock == null) {
            log("lock failed for " + o);
```

Figure 2.4: Annotations in CIDE [KAK08]

Similar to the aforementioned #ifdef of C's preprocessor, CIDE allows to annotate code with features, but in a more disciplined way. First, not arbitrary code parts can be annotated. Instead, we can only annotate elements which are optional in the language's syntax. Second, the annotations are not written in the source code, they consist of colored annotations that are stored separately in an annotation table.

In CIDE, source code can also be annotated with multiple features. In other words, colors may overlap and are drawn using a mixture of colors. When generating program variants, the source code not annotated with any features is always present. Whether annotated code is removed or not, depends on the configuration. Thus, different program variants can be generated.

2.2 Type Systems

Pierce defines a type system as follows [Pie02].

> "A *type system* is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute."

A type system is a syntactic method, since programs are classified using their syntactical elements. Therefore, a type system is usually specific to a certain language. A program variable can assume a range of values during execution of a program and an upper bound of such range is called a *type* of the variable. A language is called *typed* if variables can be given types and typed languages are *explicitly typed* if types are part of the syntax [Car97]. All languages of interest in this thesis are explicitly typed.

In general, a formal type system is the mathematical characterization of an informal type system that is described in a programming language manual. Formal type systems mainly consist of typing rules that classify programs into well-typed and ill-typed programs. Section 2.2.1 explains the process of classifying programs based on a type system called type checking. Proving that a type system is correct means to prove a property named type soundness, as we show in Section 2.2.2.

2.2.1 Type Checking

Typed languages can enforce the absence of certain program errors by performing static checks, i.e., checks at compile time. This process is named *type checking* and the tool that performs this checking is called the *type checker* [Car97]. Hence, a type checker classifies a given program as well-typed or ill-typed based on a type system. Figure 2.5 visualizes the process for a single program.

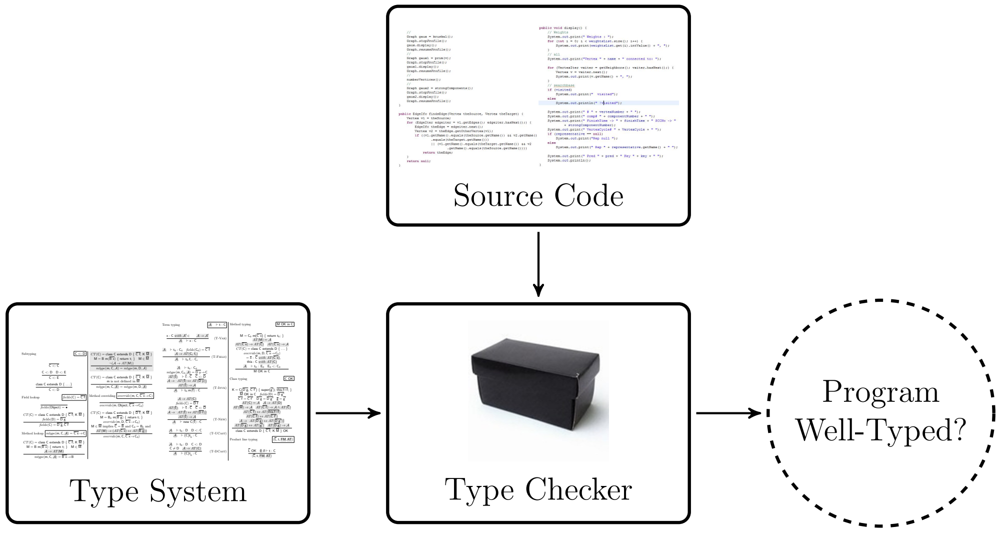


Figure 2.5: Type Checking a Single System

We want to use type checking not only for single system, we want to type-check an entire software product line. A naive approach is to generate all programs, as described in Section 2.1.2, and check each program separately. This is often not feasible, as the number of variants tends to be very high and identical parts of the variants need to be checked multiple times [AKGL09, KA08].

Product-line–aware type systems are type systems that can efficiently type check software product lines. Recently, product-line–aware type systems were proposed for different software product line languages [CP06, TBKC07, KA08, DCB09, AKGL09]. The overall idea is to classify the whole software product line as well-typed or ill-typed and for every well-typed product line, all variants generated using valid configurations are well-type.

Figure 2.6 shows that the type checker based on a product-line–aware type system gets the feature model and the code base (with a mapping from features to code) to classify software product lines into well-typed and ill-typed ones.

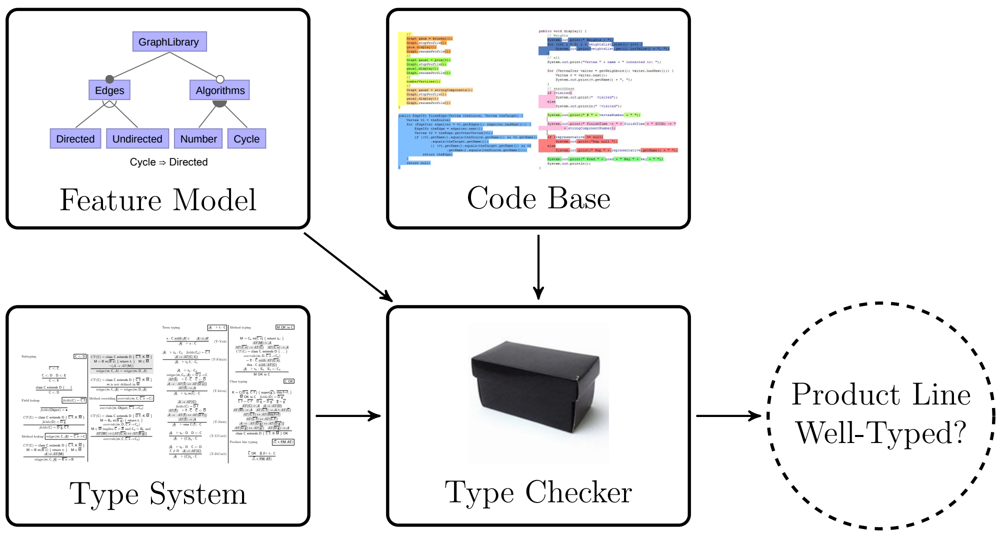


Figure 2.6: Type Checking a Software Product Line

2.2.2 Type Soundness

A *type soundness* theorem states that well-typed terms always evaluate to values [WF94]. If a type soundness theorem holds, we call the type system *sound*. In order to prove type soundness formally, we need to formalize the whole language [Car97], i.e., we need to define a relation that identifies well-typed programs and a relation that defines the semantics by providing evaluation rules.

Typing Rules

For most languages especially types and terms are of interest, where a *term* is a statement, an expression or another program fragment. To type a term, we additionally need the environment specifying the types of variables that may occur in the term of interest. For instance, in a method declaration, we may have parameter variables with specific types and whenever we want to analyze the type of a term in that method, we also need the type of all variables that occur in our term.

The relation that a term t has the type T in the context Γ is written as $\Gamma \vdash \mathsf{t} : \mathsf{T}$. An *context* Γ is a list of type assignments of the form $\mathsf{x} : \mathsf{T}$, meaning that the variable x has type T in the context Γ [WF94]. We use $\emptyset$ to denote an empty context. A type system consists of typing rules that define which terms are in this relation.

Typing rules are basically inference rules, stating that if a potentially empty set of premises is fulfilled, then the conclusion is valid. We could write them as a propositional formula, but a common notation is to write the premises above a line and the conclusion below. We give an example that is often part of type systems.

$$\frac{\mathsf{x} : \mathsf{T} \in \Gamma}{\Gamma \vdash \mathsf{x} : \mathsf{T}} \qquad \text{(T-VAR)}$$

The rule T-VAR states that a variable x has type T if the environment assigns T to x. For instance, we know that $\mathsf{y} : \mathsf{U}, \mathsf{x} : \mathsf{T} \vdash \mathsf{x} : \mathsf{T}$. We present more typing rules in the following chapter, but the overall principle is identical.

Evaluation Rules

Before we go into detail how type soundness is proven, we explain shortly how a program is evaluated to a value. Similar to typing rules, *evaluation rules* are inference rules that allow us to replace certain terms by other possibly simpler terms [Pie02].

The intuition is that t is the state of the abstract machine at a given moment, then the machine can make a step of computation and change its state to $\mathsf{t'}$. $\mathsf{t} \to \mathsf{t'}$ is pronounced as "t evaluates to $\mathsf{t'}$ in one step". Evaluation rules concisely define the semantics of the language and are very specific to a language. Therefore, we give a very general example that fits in most languages.

$$\frac{\mathsf{t} \to \mathsf{t'}}{\mathsf{t\ in\ some\ construct} \to \mathsf{t'\ in\ the\ same\ construct}} \qquad \text{(E-SOMETHING)}$$

The term t in some language construct can be evaluated to $\mathsf{t'}$ in the same construct, if t evaluates to $\mathsf{t'}$ in one step. Building the reflexive and transitive closure of the one-step evaluation, we get the multi-step evaluation denoted as $\mathsf{t} \to^* \mathsf{t'}$. A *value* is a term that cannot be further evaluated.

Progress and Preservation

The soundness theorem can be proven in two steps, known as the progress and the preservation theorem [Pie02]. The *progress* theorem states that a well-typed term is either a value or it can take a step according to the evaluation rules. The *preservation* theorem predicates that if a well-typed term takes a step of evaluation, then the resulting term is also well-typed. Both theorems together tell us that a well-typed term always evaluates to a value [WF94].

2.3 Proof Assistant Coq

Coq is a formal proof management system, or short proof assistant [CDT09a]. It provides the formal language Gallina to write mathematical definitions and theorems together with an environment for interactive development of machine-checked proofs. In 2004, Gonthier used Coq to proof the Four Color Theorem [Gon04]. This is probably the most famous machine-checked proof, because it is the first major theorem that was proven using a computer, for which still no manual proof exists.

A *proof assistant* is a tool for interactive theorem proving, whereas formal proofs are developed by a man-machine collaboration [BC04]. It comes with an interactive proof editor, with which a human provides mathematical definitions, theorems and proofs. The proof assistant can verify that the proofs are correct. If a proof is accepted by the proof assistant, we call it a *machine-checked proof*.

Similarly, in *automated* theorem proving, a human defines mathematical theorems in a certain logic and the machine checks their validity. Contrary to proof assistants, the human does not write a proof and theorem provers do not provide a human readable proof. Automated theorem proving usually requires more computing power and it may not terminate within a reasonable time.

In our work, the proof assistant Coq is used to verify a type soundness proof for a type system. Therefore, all definitions that build up the type system and the proof are formalized in Gallina, the proof assistant's language (see Figure 2.7). Coq verifies the definitions and the proof step-by-step. Either all statements can be verified or Coq stops earlier with an error message and we know the proof is incomplete. Given the definitions and the proof, Coq can verify whether the proof is correct or not.

In the following, we explain fundamentals of theorem proving with Coq. This background is necessary to understand definitions and proofs provided in Chapter 4 and Chapter 5. Section 2.3.1 introduces the basic language constructs of Gallina, i.e., how definitions can be expressed with it. Finally, in Section 2.3.2, we introduce theorem definitions and how tactics are used in Coq to interactively develop proofs.

2.3.1 Gallina

Gallina is the specification language of Coq and it allows to develop mathematical theories and proofs. It is out of the scope of this thesis to give a complete introduction to Coq, but we exemplify Gallina in the following. For more details, we refer to

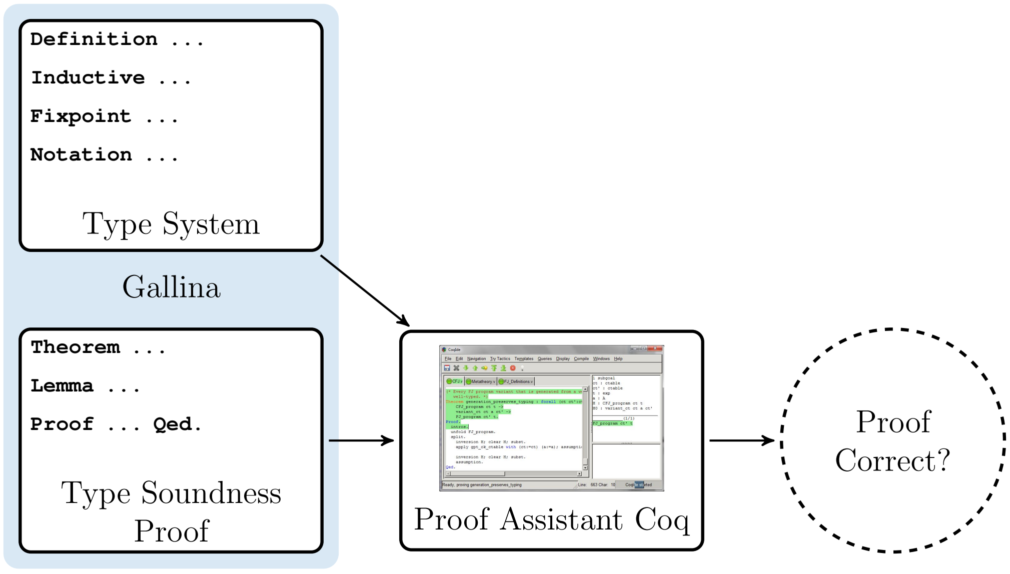


Figure 2.7: Type Soundness Proofs using Coq

the reference manual [CDT09b] as well as to Bertot and Castéran's book "Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions" [BC04].

Inductive definitions play a fundamental role in Coq. We give two examples in Listing 2.1. The first defines natural numbers inductively. We name our definition `nat` and it is of type **`Set`**. In Gallina every term is typed using the notation `term : type`. Elements in `nat` are either `O` of type `nat` or `S` which is the successor function taking one element of `nat` and returns an element of type `nat`.

```
Inductive nat : Set :=
  | O : nat
  | S : nat → nat.

Inductive even : nat → Prop :=
  | even_O : even O
  | even_SS : forall n:nat, even n → even (S (S n)).
```

Listing 2.1: Inductive Definitions in Gallina

Our second inductive definition shows how `nat` can be used. It is named `even` and is an unary relation on `nat`. The latter is indicated by the type `nat` → **`Prop`**. `O` is in the relation `even` and if `n` is in `even`, where `n : nat`, then also `(S (S n))` is in the relation `even`. The identifiers `even_O` and `even_SS` are used to refer to these options.

We now define a recursive function `add` using the keyword **`Fixpoint`** that adds two terms of type `nat` (see Listing 2.2), which is needed for a simple proof in the following section. The parameters `n` and `m` are of type `nat`. Coq verifies that the recursive function terminates for all possible parameters. We need to help Coq by specifying the decreasing parameter using `{struct n}`. We use the **`match`** operator for a distinction on the possible cases `n` with the various constructors. If `n` equals `O`, then `m` is returned. Otherwise `n` matches `S p` and we return the successor of `add p m`.

```
Fixpoint add (n m:nat) {struct n} : nat :=
  match n with
  | O ⇒ m
  | S p ⇒ S (add p m)
  end.

Definition twice (n:nat) : nat := add n n.
```

Listing 2.2: Recursive Functions and Definitions in Gallina

We can also have non-inductive definitions. Our example is the definition of `twice` as a function with a parameter of type `nat`. It returns a term of type `nat`, which is twice the given value. We explained the most used language constructs of our formalization. The next section shows how to write basic theorems and proofs in Coq.

2.3.2 Proof Tactics

In order to proof a lemma or a theorem, we first need to define it. We give two examples in Listing 2.3 based on the definitions of the previous section. The lemma `add_left` states that for the addition of two elements of type `nat`, the following equality holds $n + (1 + m) = 1 + (n + m)$. It is declared as a lemma cause we only need it to proof the theorem `even_twice`: every natural number multiplied by two is even. The theorems are rather simple, but the proofs already contain the most important language constructs for proof writing.

First, we introduce the nomenclature used in Coq. Proving a theorem means to solve a *goal*. When starting the proof, the goal is identical to the theorem. We use tactics to manipulate the (sub-)goal to get to simpler subgoals. The proof is finished, if we solved all subgoals. Therefore, a tactic might (a) solve the subgoal, (b) produce new subgoals, or (c) replaces the subgoal [BC04].

Before we explain our example proofs, we need to mention that proving in Coq has one main difference to manual proofs. Manual proofs usually base on forward reasoning: given a proof of A and B, we can deduce C. Instead, Coq uses backward reasoning: applying a tactic means to replace the proof of C by a proof of A and a proof of B. That way, a tactic reduces a goal to a number of subgoals [CDT09b].

We start explaining how our example theorem `even_twice` is proven. Usually, we begin our proofs with the `intros`-tactic that takes all quantified variables and premises and states them as our assumptions. In our example, we then have the assumption that `n` is a variable of type `nat` and our new goal is `even (twice n)`.

The tactic `unfold` replaces an identifier (in our case `twice`) by its definition. This results in the subgoal `even (add n n)`. We then start an induction on `n` using the `induction` tactic. This results in two subgoals which we explain in the following.

```
Lemma add_left : forall (n m:nat), add n (S m) = S (add n m).
Proof.
  intros.
  induction n.
    unfold add.
    trivial.

    unfold add.
    fold add.
    rewrite IHn.
    trivial.
Qed.

Theorem even_twice : forall (n:nat), even (twice n).
Proof.
  intros.
  unfold twice.
  induction n.
    unfold add.
    apply even_O.

    rewrite add_left.
    unfold add.
    fold add.
    apply even_SS.
    apply IHn.
Qed.
```

Listing 2.3: Proofs in Gallina

The first subgoal is `even (add O O)`; the induction beginning. Unfolding the definition of `add` simply means to replace it by `O`. The reason is that the first parameter matches to `O` (see definition in Listing 2.2). The `apply` tactic can be used to solve the goal since `even_O` is a proof of `even O`. We identify solved goals by a followed empty line or **`Qed`**. The indent is used to show how many subgoals are produced by a certain tactic.

The second subgoal is `even (add (S n) (S n))`; the induction step. We first use our lemma to simplify our subgoal using the `rewrite` tactic. This results in the subgoal `even (S (add (S n) n))`. Unfolding `add` and folding it right afterwards leads us to the subgoal `even (S (S (add n n)))`. Applying `even_SS` we get the subgoal `even (add n n)`, what is exactly our induction hypothesis. Hence, applying our induction hypothesis named `IHn` solves the last subgoal.

The proof of lemma `add_left` has only one different tactic named `trivial`. It solves a goal if it is a trivial equality. For instance, `S m = S m` at its first use and at the end of the proof it solves `S (S (add n m)) = S (S (add n m))`. For more details on tactics, we refer to the Coq reference manual [CDT09b].

3. Colored Featherweight Java

This chapter describes the basic concepts of FJ and CFJ. These programming languages are already known from the literature, but a clear understanding is needed for the following two chapters, where we formalize the type systems in Gallina (see Chapter 4) and proof crucial properties about them using the proof assistant Coq (see Chapter 5).

The main innovation of this chapter is our revised type system for CFJ. The type system known from literature can be simplified due to redundant premises at the some typing rules. A smaller contribution is that we give some new and adapted examples of FJ programs and CFJ product lines.

In Section 3.1, we introduce the syntax of FJ, give examples and a type system. Based on that, Section 3.2 shortly describes the extension CFJ and present a type system from the literature. Finally, we propose a simplified type system for CFJ in Section 3.3 and prove that it is equivalent to the original type system.

3.1 Featherweight Java

> "Inside every large language is a small language struggling to get out..."
> Tony Hoare [IPW01]

In 1999, Igarashi et al. presented FJ as a lightweight version of Java [IPW99]. They omit almost all language constructs of Java to ease type soundness proofs and propose it as a good starting point for proofs on language extensions. A type soundness proof for FJ and a proof sketch for an extension with generic classes are provided. A full proof is given in the revised version of the paper published in 2001 [IPW01], which describes the calculus in more detail and a slightly changed notation. Additionally, FJ is a subject of Pierce's book "Types and Programming Languages" [Pie02]. We base our work on Pierce's notation as it is also used in publications dealing with CFJ.

Notice, that for full Java type soundness proofs are not practical, since the Java language description is informal and 688 pages long [GJSB05]. Formalizations and proofs tend to be more extensive. Therefore, the community on type systems often uses elementary languages to proof certain properties of interest. We point the reader to Chapter 7 for more such elementary languages.

FJ is a programming language, while it is not intended to be used it in industrial practice. The expressiveness of FJ is closer to that of the lambda-calculus than that of a real programming language like Java. Nevertheless, it can be used to get a better understanding of the fundamental concepts in Java. Igarashi et al. propose that type soundness proofs for FJ illustrate "many of the interesting features of a safety proof for the full language" [IPW99].

FJ is designed with a special property in mind; every FJ program is also a full Java program. We refer to this property as backward compatibility and it means, that we can use tools designed for Java to write FJ programs, e.g., with syntax highlighting or code assist, and compile and run such programs. This is an important property from a tool support perspective and is also important for extensions as CFJ, as we explain in Section 3.2.

In Section 3.1.1, we formally characterize syntactically correct FJ programs. Examples for syntactically correct programs are given in Section 3.1.2 for a better comprehension. Particularly, those examples are well-typed according to the type system we state in Section 3.1.3.

3.1.1 Syntax

Before we present the syntax and type system for FJ, we introduce the notations used. Table 3.1 gives an overview on the ranges of used meta-variables. We assume the special variable this, which cannot be used as a method parameter and is replaced by an appropriate object.

Meta-Variables	Range Over
C, D, E	Class names
f, g	Field names
m	Method names
x	Variables
t	Terms
L	Class declarations
K	Constructor declarations
M	Method declarations

Table 3.1: Meta-Variables in CFJ and Their Meaning

We write $\overline{\mathsf{A}}$ for a possibly empty sequence of $\mathsf{A}_1, \ldots, \mathsf{A}_n$, where A is one of our meta-variables or a term like $\mathsf{C}\ \mathsf{f}$. The latter, i.e., $\overline{\mathsf{C}\ \mathsf{f}}$ stands for $\mathsf{C}_1\ \mathsf{f}_1, \ldots, \mathsf{C}_n\ \mathsf{f}_n$. The empty sequence is denoted by • and the concatenation of sequences by a comma. Sequences of field declarations, parameter names, and method declarations are assumed to contain no duplicate names. As in Java, we assume that casts bind less tightly than other forms of expression [IPW01].

FJ drops complex language features of Java, such as threads or reflection, but even simpler ones are neglected: interfaces, assignments, imports, abstract classes, inner classes, modifiers, static methods - to give some examples. Let us take a closer look at what language constructs FJ provides using the FJ syntax in Figure 3.1.

P	::=	($\overline{\mathsf{L}}$, t)	*FJ program*
L	::=	class C extends C {$\overline{\mathsf{C\ f}}$; K $\overline{\mathsf{M}}$ }	*class declaration*
K	::=	C($\overline{\mathsf{C\ f}}$) { super($\overline{\mathsf{f}}$); $\overline{\mathsf{this.f{=}f;}}$ }	*constructor declaration*
M	::=	C m($\overline{\mathsf{C\ x}}$) { return t; }	*method declaration*
t	::=		*terms:*
		x	*variable*
		t.f	*field access*
		t.m($\overline{\mathsf{t}}$)	*method invocation*
		new C($\overline{\mathsf{t}}$)	*object creation*
		(C)t	*cast*

Figure 3.1: The Syntax of Featherweight Java [KA08]

A FJ program consists of a list of classes (class table) and a start term that acts as the content of the programs main method. Each class has a unique name, a super class that might be Object, a number of fields, one constructor and a list of methods. The constructor provides a list of parameters, a super call and a list of assignments that save the parameters in fields.

A method has a return type, a unique name, a list of parameters and one term calculating the return value. Finally, a term can be a variable, a field access, a method invocation with a list of terms as parameters, an object creation with a list of terms as parameters or the cast of a term.

3.1.2 Examples

Since syntax descriptions are not always easy to look through, we give some examples for syntactically correct FJ programs. Additionally, these examples are well-typed regarding the type system that we provide in Section 3.1.3. They may be useful to understand the following considerations.

A Minimal Program

The first example that we give is a minimal FJ program (see Figure 3.2). The simplest start term that a program can have is given by a creation term of a class `C` without fields. Of course, we need this class `C` at the class table and it do not need to have

```
class C extends Object {
    C() { super(); }
}
```

```
new C()
```

Figure 3.2: A Minimal FJ Program

any methods. In some way, this program corresponds to a Java program with a main method that just creates an instance of a class (with an empty constructor).

The class `C` extends the class `Object`. Hence, there are no parameters that we have pass at the super call. Since `C` does not have any fields, the constructor has none either and no fields can be initialized. To get minimal program, `C` does not have any methods.

Pairs

Our second example can handle pairs of objects and we have two kinds of objects that can be passed (see Figure 3.3). This example and some of the following are based on an example from Igarashi et al. [IPW01]. The class `Pair` has two fields saving the first and the second element of the pair. The constructor gets the two elements as parameters and initializes the fields.

```
class A extends Object {
    A() { super(); }
}
class B extends Object {
    B() { super(); }
}
class Pair extends Object {
    Object fst;
    Object snd;
    Pair(Object fst, Object snd) {
        super(); this.fst=fst; this.snd=snd;
    }
    Pair setfst(Object newfst) {
        return new Pair(newfst, this.snd);
    }
}
```

```
((Pair)
    new Pair(new A(), new A()).
        setfst(new Pair(new A(), new B())).fst
).snd
```

Figure 3.3: A FJ Program that Handles Pairs

Additionally, there is a method to set the first element of the pair. But since we cannot assign a new value to the fields, we need to return a new instance of `Pair`. The start term creates a new instance of `Pair` and sets the first element to a new pair. Then, the first element of the outermost pair is returned, casted to a pair and the second element is returned. This term evaluates to `new B()`.

The presented FJ programs do not contain any type errors. But these errors do occur if we would remove the class declaration for the class `A` or if we omit the the method `setfst`. These examples would lead to dangling class or method references. To identify well-typed programs we present a type system for FJ in the following section.

3.1.3 Type System

In this section, we give a compact representation of the FJ type system. For a detailed description we refer the reader to Pierce's book "Types and Programming Languages" [Pie02]. All inference rules assume a fixed class table CT that satisfies some sanity conditions:

1. $\mathsf{class\ C}\ldots$ for every $\mathsf{C} \in \mathsf{dom}(\mathsf{CT})$
2. $\mathsf{Object} \notin \mathsf{dom}(\mathsf{CT})$
3. For every class name C (except Object) appearing anywhere in CT, we have $\mathsf{C} \in \mathsf{dom}(\mathsf{CT})$
4. There are no cycles in the subtype relation induced by CT, i.e., the $<:$ relation is antisymmetric

Figure 3.4 contains all inference rules and we explain them shortly in a top-down way. A FJ program is well-typed if all classes in the class table and the start term are well-typed. A class is well-typed if the constructor initializes the fields and super fields properly and all methods are well-typed. A method is well-typed if it correctly overrides potentially super methods and the outermost term is well-typed and the type is a subtype of the return type.

A term can have five different shapes that are typed individually. A variable has a certain type if it is specified by the context (see T-VAR). According to T-FIELD a field access is well-typed if the base term is well-typed and the field exists for that class. For a method invocation also the base term must be well-typed and the parameter terms must be well-typed and their types have to be a subtypes of the method parameter types (see T-INVK). T-NEW states the same for object creation terms except that we do not have a base term.

Casting terms is more complicated. In general, up- and down-casts are allowed. T-UCAST forces that a cast is well-typed if the base term is well-typed and its type is a subtype of the target class. Analogously, a cast is well-typed if the base type is well-typed and its type is a super type of the target class (see T-DCAST). But, a combination of up- and downcasts can result in ill-typed terms. For example, `(A)(Object)new B()` is a stupid cast, for two unrelated types `A` and `B` as defined in Figure 3.3. Therefore, the FJ type systems specifies well-typed programs except for stupid casts.

Subtyping $\boxed{\mathsf{C} <: \mathsf{D}}$

$$\frac{}{\mathsf{C} <: \mathsf{C}}$$

$$\frac{\mathsf{C} <: \mathsf{D} \quad \mathsf{D} <: \mathsf{E}}{\mathsf{C} <: \mathsf{E}}$$

$$\frac{\mathsf{class\ C\ extends\ D}\ \{\ \dots\}}{\mathsf{C} <: \mathsf{D}}$$

Field lookup $\boxed{\mathit{fields}(\mathsf{C}) = \overline{\mathsf{C}\ \mathsf{f}}}$

$$\frac{}{\mathit{fields}(\mathsf{Object}) = \bullet}$$

$$\frac{\mathit{CT}(\mathsf{C}) = \mathsf{class\ C\ extends\ D}\ \{\ \overline{\mathsf{C}\ \mathsf{f}};\ \mathsf{K}\ \overline{\mathsf{M}}\ \} \quad \mathit{fields}(\mathsf{D}) = \overline{\mathsf{D}\ \mathsf{g}}}{\mathit{fields}(\mathsf{C}) = \overline{\mathsf{D}\ \mathsf{g}}, \overline{\mathsf{C}\ \mathsf{f}}}$$

Method lookup $\boxed{\mathit{mtype}(\mathsf{m}, \mathsf{C}) = \overline{\mathsf{C}} \to \mathsf{C}}$

$$\frac{\mathit{CT}(\mathsf{C}) = \mathsf{class\ C\ extends\ D}\ \{\ \overline{\mathsf{C}\ \mathsf{f}};\ \mathsf{K}\ \overline{\mathsf{M}}\ \} \quad \mathsf{B\ m}(\overline{\mathsf{B}\ \mathsf{x}})\ \{\ \mathsf{return\ t;}\ \} \in \overline{\mathsf{M}}}{\mathit{mtype}(\mathsf{m}, \mathsf{C}) = \overline{\mathsf{B}} \to \mathsf{B}}$$

$$\frac{\mathit{CT}(\mathsf{C}) = \mathsf{class\ C\ extends\ D}\ \{\ \overline{\mathsf{C}\ \mathsf{f}};\ \mathsf{K}\ \overline{\mathsf{M}}\ \} \quad \mathsf{m} \text{ is not defined in } \overline{\mathsf{M}}}{\mathit{mtype}(\mathsf{m}, \mathsf{C}) = \mathit{mtype}(\mathsf{m}, \mathsf{D})}$$

Method overriding $\boxed{\mathit{override}(\mathsf{m}, \mathsf{C}, \overline{\mathsf{C}} \to \mathsf{C}_0)}$

$$\frac{\mathit{mtype}(\mathsf{m}, \mathsf{C}) = \overline{\mathsf{B}} \to \mathsf{B}_0 \text{ implies } \overline{\mathsf{C}} = \overline{\mathsf{B}} \text{ and } \mathsf{C}_0 = \mathsf{B}_0}{\mathit{override}(\mathsf{m}, \mathsf{C}, \overline{\mathsf{C}} \to \mathsf{C}_0)}$$

Term typing $\boxed{\Gamma \vdash \mathsf{t} : \mathsf{C}}$

$$\frac{\mathsf{x} : \mathsf{C} \in \Gamma}{\Gamma \vdash \mathsf{x} : \mathsf{C}} \quad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash \mathsf{t}_0 : \mathsf{C}_0 \quad \mathit{fields}(\mathsf{C}_0) = \overline{\mathsf{C}\ \mathsf{f}}}{\Gamma \vdash \mathsf{t}_0.\mathsf{f}_i : \mathsf{C}_i} \quad \text{(T-FIELD)}$$

$$\frac{\Gamma \vdash \mathsf{t}_0 : \mathsf{C}_0 \quad \mathit{mtype}(\mathsf{m}, \mathsf{C}_0) = \overline{\mathsf{D}} \to \mathsf{C} \quad \Gamma \vdash \overline{\mathsf{t}} : \overline{\mathsf{C}} \quad \overline{\mathsf{C}} <: \overline{\mathsf{D}}}{\Gamma \vdash \mathsf{t}_0.\mathsf{m}(\overline{\mathsf{t}}) : \mathsf{C}} \quad \text{(T-INVK)}$$

$$\frac{\mathit{fields}(\mathsf{C}) = \overline{\mathsf{D}\ \mathsf{f}} \quad \Gamma \vdash \overline{\mathsf{t}} : \overline{\mathsf{C}} \quad \overline{\mathsf{C}} <: \overline{\mathsf{D}}}{\Gamma \vdash \mathsf{new\ C}(\overline{\mathsf{t}}) : \mathsf{C}} \quad \text{(T-NEW)}$$

$$\frac{\Gamma \vdash \mathsf{t}_0 : \mathsf{D} \quad \mathsf{D} <: \mathsf{C}}{\Gamma \vdash (\mathsf{C})\mathsf{t}_0 : \mathsf{C}} \quad \text{(T-UCAST)}$$

$$\frac{\Gamma \vdash \mathsf{t}_0 : \mathsf{D} \quad \mathsf{C} <: \mathsf{D} \quad \mathsf{C} \neq \mathsf{D}}{\Gamma \vdash (\mathsf{C})\mathsf{t}_0 : \mathsf{C}} \quad \text{(T-DCAST)}$$

Method typing $\boxed{\mathsf{M\ OK\ in\ C}}$

$$\frac{\mathit{CT}(\mathsf{C}) = \mathsf{class\ C\ extends\ D}\ \{\ \dots\} \quad \mathit{override}(\mathsf{m}, \mathsf{D}, \overline{\mathsf{C}} \to \mathsf{C}_0) \quad \overline{\mathsf{x}} : \overline{\mathsf{C}}, \mathsf{this} : \mathsf{C} \vdash \mathsf{t}_0 : \mathsf{E}_0 \quad \mathsf{E}_0 <: \mathsf{C}_0}{\mathsf{C}_0\ \mathsf{m}(\overline{\mathsf{C}\ \mathsf{x}})\ \{\ \mathsf{return\ t_0;}\ \}\ \mathsf{OK\ in\ C}}$$

Class typing $\boxed{\mathsf{C\ OK}}$

$$\frac{\mathsf{K} = \mathsf{C}(\overline{\mathsf{D}\ \mathsf{g}}, \overline{\mathsf{C}\ \mathsf{f}})\ \{\ \mathsf{super}(\overline{\mathsf{g}});\ \overline{\mathsf{this.f{=}f;}}\ \} \quad \overline{\mathsf{M}}\ \mathsf{OK\ in\ C} \quad \mathit{fields}(\mathsf{D}) = \overline{\mathsf{D}\ \mathsf{g}}}{\mathsf{class\ C\ extends\ D}\ \{\ \overline{\mathsf{C}\ \mathsf{f}};\ \mathsf{K}\ \overline{\mathsf{M}}\ \}\ \mathsf{OK}}$$

Program typing $\boxed{(\overline{\mathsf{L}}, \mathsf{t})}$

$$\frac{\overline{\mathsf{L}}\ \mathsf{OK} \quad \emptyset \vdash \mathsf{t} : \mathsf{C}}{(\overline{\mathsf{L}}, \mathsf{t})}$$

Figure 3.4: Subtyping, Auxiliary, and Typing Rules for FJ

Finally, we have a look at the subtyping and auxiliary rules. To identify valid subtypes, the reflexive and transitive closure on the inheritance hierarchy from the class table is determined. Field lookup is defined recursive where all fields of a class are the own fields together with all fields of the super class; while Object has no fields at all. Method lookup gives the signature of a method in case it exists and it consist of a sequence of parameter types and the return type. If the method is defined at the current class the signature is returned, otherwise there is an recursive call on the super class. Valid method overriding simply uses the method lookup.

Another part of FJ's type system are the evaluation rules. They define the semantics of FJ programs. We omit the evaluation rules in our thesis as we do not need it. The FJ extension of interest does not need evaluation rules, because a CFJ product line is never directly evaluated. Instead, a FJ program is generated at evaluated. We would need it if we proof type soundness for FJ which is already done and out of the scope of this thesis.

3.2 Colored Featherweight Java

In 2008, Kästner and Apel presented CFJ as an extension of FJ to support software product lines [KA08]. Instead of writing multiple FJ programs separately, a CFJ product line is written, from which we generate separate FJ programs. CFJ product lines are never directly evaluated. Instead, FJ programs are generated that can be evaluated.

The principle of CFJ is identical to that of CIDE (see Section 2.1.3). Code fragments can be annotated with certain colors representing features. The tool is based on full Java, but the type system is not complete due to complexity reasons. CIDE can be used to write CFJ product lines, since (1) every CFJ product line stripped of its annotations and the feature model is a valid FJ program (see Section 3.2.1) and (2) every FJ program is also a valid full Java program (see Section 3.1).

As for FJ, we start providing the syntax description in Section 3.2.1 and some examples of syntactically valid and well-typed CFJ product lines in Section 3.2.2. Section 3.2.3 describes how variability is achieved using annotations, i.e., how FJ program variants are generated from a CFJ product line. Finally, we present the type system for CFJ in Section 3.3.4.

3.2.1 Syntax and Annotations

The overall idea in CIDE is that languages are extended to develop product lines, but the syntax is identical to that of the host language. This way, editors and views in IDEs can be used for the extended language. As CFJ product lines can be written in CIDE, the syntax (of classes) is identical to that of FJ. The only difference is that a CFJ program additionally consists of a feature model FM describing the available variants (highlighted in Figure 3.5).

CFJ is a product-line–aware language and the variability is achieved by annotations. Code fragments can be annotated with a certain color representing a feature of the software. Given a CFJ product line, variants can be generated by removing code annotated

P	::=	($\overline{\mathbf{L}}$, t, FM)	*CFJ product line*
L	::=	class C extends C {$\overline{\mathbf{C\ f}}$; K $\overline{\mathbf{M}}$ }	*class declaration*
K	::=	C($\overline{\mathbf{C\ f}}$) { super($\overline{\mathbf{f}}$); $\overline{\mathbf{this.f{=}f}}$; }	*constructor declaration*
M	::=	C m($\overline{\mathbf{C\ x}}$) { return t; }	*method declaration*
t	::=		*terms:*
		x	*variable*
		t.f	*field access*
		t.m($\overline{\mathbf{t}}$)	*method invocation*
		new C($\overline{\mathbf{t}}$)	*object creation*
		(C)t	*cast*

Figure 3.5: The Syntax of Colored Featherweight Java [KA08]

with particular colors. The idea of colors as annotations in Java or fragments of Java is similar to that of #ifdef statements for the C preprocessor, but more disciplined. There are more restrictions on the kind of annotations and which code fragments can be annotated as we show in the following.

All optional code fragments in the syntax can be annotated. This makes sure that removing annotated code fragments we get a syntactically correct FJ program. Optional code fragments do only occur as elements of lists (**bold** in Figure 3.5), such as classes, fields, methods, constructor parameters, super constructor call parameters, field initializations, method parameters, method call parameters, and creation parameters. An element can be annotated with more than one color.

As there are no syntactical constructs to store annotations, the annotations are stored externally in an annotation table. The annotation table maps code fragments to their annotated colors. Since a code fragment might occur multiple times, additionally the line number is used to reference each element and that, e.g., different occurrences of `new C()` can have different annotations.

Given a type system for FJ and an appropriate type system for CFJ, the backward compatibility can be shown. A proof is given in [KA08]. Together with the backward compatibility of FJ to full Java, we get that every CFJ stripped of its feature model and all annotations is a valid Java program. Again, this is useful as we can use Java tools to develop CFJ product lines.

3.2.2 Examples

Syntactically correct CFJ product lines were formalized in the previous section. Before we present the type system in Section 3.2.4, we want to give two examples of well-typed CFJ product lines for a better understanding. We visualize how code fragments are annotated with colors and explain how the according FJ program variants look like.

Swapping Pairs

In Figure 3.6, a CFJ product line of programs to handle pairs is presented. There is a method to swap elements, i.e., to flip the left and the right element of a pair. The method `swap()` in the class `Pair` is annotated with the feature Swapping (highlighted in Figure 3.6). We can remove this method since it overrides a method in `AbstractPair` and all calls to this method are still possible.

```
//definition of A and B
class AbstractPair extends Object {
    Object fst;
    Object snd;
    AbstractPair(Object fst, Object snd) {
        super(); this.fst=fst; this.snd=snd;
    }
    Pair swap() {
        return new Pair(this.fst, this.snd);
    }
class Pair extends AbstractPair {
    Pair(Object fst, Object snd) {
        super(fst, snd);
    }
    Pair swap() {
        return new Pair(this.snd, this.fst);
    }
}
```

```
new Pair(new A(), new B()).swap().fst
```

Figure 3.6: A CFJ Product Line that Handles Pairs

The super method does not swap the elements. Thus, we get a different behavior depending on whether the highlighted code is present or not. Suppose the feature model allows it, we are able to generate two FJ program variants. One with and one without the Swapping feature. If we select the Swapping feature the start term evaluates to **`new`** `B()` and otherwise to **`new`** `A()`.

Pairs with an Optional Element

This example also handles pairs, whereas the second element is optional. According to Figure 3.7, we have pairs that consists of a first and a second element and pairs that only have the first element.

Instances of the class `Pair` store two objects `fst` and `snd`. The objects are initialized in the constructor and no further methods or fields exist. The field `snd` is annotated with the feature Second (highlighted in Figure 3.7). Additionally, the according constructor parameter and the field assignment are annotated with that feature, too.

Furthermore, we have a library to create pairs. The class `Library` has no further fields, but two methods that create pairs, where either the left or the right element is always an instance of `A`. In both method the second parameter is annotated with the feature SecondCall (highlighted in Figure 3.7). We annotated the code in our example with

```
//definition of A and B
class Pair extends Object {
    Object fst;
    Object snd;
    Pair(Object fst, Object snd) {
        super(); this.fst=fst; this.snd=snd;
    }
}
class Library extends Object {
    Library() { super(); }
    Pair pairAX(Object second) {
        return new Pair(new A(), second);
    }
    Pair pairXA(Object first) {
        return new Pair(first, new A());
    }
}
```

```
new Library().pairAX(new B()).fst
```

Figure 3.7: A CFJ Product Line with Pairs and Single Elements

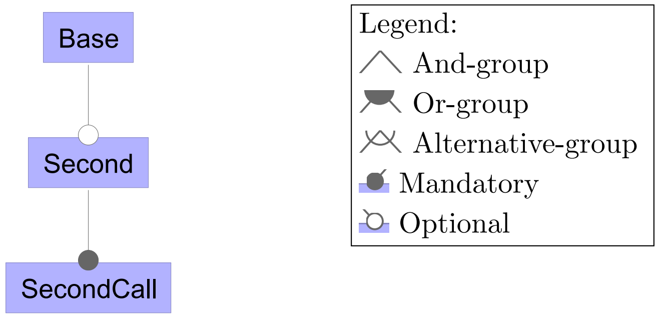


Figure 3.8: The Feature Model for our Pair Product Line

two features to clarify the impact of the feature model. We present the feature model for our example in Figure 3.8.

The feature Base represents all code that not annotated with other features. The feature Second is optional and the feature SecondCall is selected, if and only if Second is selected. This is important, because otherwise the number of parameters of `Pair` creation terms and the constructor of `Pair` would be different. The propositional formula representing this feature model is as follows.

$$Base \wedge (Second \vee SecondCall \Rightarrow Base) \wedge (Second \Leftrightarrow SecondCall)$$

Hence, our product line contains two FJ programs, the program with pairs (all features selected) and the program with single elements (only Base selected). The start term calls the method from the library, that creates a pair with **new** `A()` as the first element and the parameter as the second. We return the first element of this pair. Hence, the start term evaluates for both programs to **`new`** `A()`.

3.2.3 Variant Generation

Given a CFJ product line, FJ programs are generated by removing source code annotated with certain colors. Therefore, a feature selection specifies which features we want to be present in our variant, where a color is annotated to all the code that represents a feature. We use the same notation as Kästner and Apel to formalize the variant generation [KA08].

FJ programs are generated using the function *variant* that takes a CFJ product line or a CFJ code fragment X and a configuration C as input and returns a FJ program. The function *variant* is defined recursively decreasing in X and applies the function *remove* to all code fragments that can be annotated. The latter removes code fragments, for which the annotation evaluates to false. All other code fragments remain in the code and are stripped of their annotations. For brevity, we write *variant*(X,C) as $[\![\mathsf{X}]\!]$ and *remove*$(\overline{\mathsf{X}},\mathsf{C})$ as $\langle\!\langle\overline{\mathsf{X}}\rangle\!\rangle$. The *variant* function is defined in Figure 3.9.

$$[\![\mathsf{x}]\!] = \mathsf{x} \tag{G.1}$$
$$[\![\mathsf{t.f}]\!] = [\![\mathsf{t}]\!].\mathsf{f} \tag{G.2}$$
$$[\![\mathsf{t.m(\bar{t})}]\!] = [\![\mathsf{t}]\!].\mathsf{m}([\![\langle\!\langle\bar{\mathsf{t}}\rangle\!\rangle]\!]) \tag{G.3}$$
$$[\![\mathsf{new\ C(\bar{t})}]\!] = \mathsf{new\ C}([\![\langle\!\langle\bar{\mathsf{t}}\rangle\!\rangle]\!]) \tag{G.4}$$
$$[\![\mathsf{(C)t}]\!] = \mathsf{(C)}[\![\mathsf{t}]\!] \tag{G.5}$$
$$[\![\mathsf{C\ m(\overline{C\ x})\ \{return\ t;\}}]\!] = \mathsf{C\ m}(\langle\!\langle\overline{\mathsf{C\ x}}\rangle\!\rangle)\ \{\mathsf{return}\ [\![\mathsf{t}]\!];\} \tag{G.6}$$
$$[\![\mathsf{C(\overline{C\ f})\ \{super(\bar{f});\ \overline{this.f{=}f;}\}}]\!] = \mathsf{C}(\langle\!\langle\overline{\mathsf{C\ f}}\rangle\!\rangle)\ \{\mathsf{super}(\langle\!\langle\bar{\mathsf{f}}\rangle\!\rangle);\ \langle\!\langle\overline{\mathsf{this.f{=}f;}}\rangle\!\rangle\} \tag{G.7}$$
$$[\![\mathsf{class\ C\ extends\ D\ \{\ \overline{C\ f};\ K\ \overline{M}\ \}}]\!] = \mathsf{class\ C\ extends\ D}\ \{\ \langle\!\langle\overline{\mathsf{C\ f}}\rangle\!\rangle;\ [\![\mathsf{K}]\!]\ [\![\langle\!\langle\overline{\mathsf{M}}\rangle\!\rangle]\!]\ \} \tag{G.8}$$
$$[\![(\overline{\mathsf{L}},\mathsf{t},\mathsf{FM})]\!] = ([\![\langle\!\langle\overline{\mathsf{L}}\rangle\!\rangle]\!],[\![\mathsf{t}]\!]) \tag{G.9}$$

Figure 3.9: The Variant Generation for CFJ [KA08]

Such program variants generated from a CFJ product line are always syntactically correct FJ programs. The reason is, that CFJ uses the same syntax for classes as FJ and only optional elements can be annotated and thus removed. Additionally, it can be shown that every variant generated from a well-typed CFJ product line with a valid configuration is a well-typed FJ program. Given type systems for FJ and CFJ we present a machine-checked proof in Chapter 5.

3.2.4 Type System

Type errors can especially occur in CFJ. For instance, if we annotate a method declaration but not the method invocation or if we annotate a parameter of a method but do not annotate the method invocation accordingly. The problem might only occur in some program variants. As the number of program variants is potentially high, it is not feasible to generate and type-check all FJ programs.

In 2008, Kästner and Apel presented a product-line–aware type system for CFJ which supersedes generating and type-checking each variant [KA08]. They extended the FJ type system with reachability checks. Reachability checks make sure that if a certain code fragment is present in a variant, also other code fragments are present, e.g., the invocation of a method always implies an existing declaration. Since this type system is similar to the FJ type system we highlighted the extensions, which are basically reachability checks. We only explain the extensions and start with the auxiliary rules in Figure 3.10.

As this is part of ongoing work, there are some differences between our definitions and those in [KA08]. For the type system, we use unpublished auxiliary and typing rules, since they are revised and will appear in upcoming work [KAS].

Subtyping $\boxed{\mathsf{C} <: \mathsf{D}}$

$$\frac{}{\mathsf{C} <: \mathsf{C}}$$

$$\frac{\mathsf{C} <: \mathsf{D} \quad \mathsf{D} <: \mathsf{E}}{\mathsf{C} <: \mathsf{E}}$$

$$\frac{\mathsf{class\ C\ extends\ D\ \{\ \ldots\}}}{\mathsf{C} <: \mathsf{D}}$$

Field lookup $\boxed{\mathit{fields}(\mathsf{C}) = \overline{\mathsf{C}\ \mathsf{f}}}$

$$\frac{}{\mathit{fields}(\mathsf{Object}) = \bullet}$$

$$\frac{CT(\mathsf{C}) = \mathsf{class\ C\ extends\ D}\ \{\ \overline{\mathsf{C}\ \mathsf{f}};\ \mathsf{K}\ \overline{\mathsf{M}}\ \} \qquad \mathit{fields}(\mathsf{D}) = \overline{\mathsf{D}\ \mathsf{g}}}{\mathit{fields}(\mathsf{C}) = \overline{\mathsf{D}\ \mathsf{g}}, \overline{\mathsf{C}\ \mathsf{f}}}$$

Method lookup $\boxed{\mathit{mtype}(\mathsf{m}, \mathsf{C}, \mathcal{A}) = \overline{\mathsf{C}\ \mathsf{x}} \to \mathsf{C}}$

$$\frac{CT(\mathsf{C}) = \mathsf{class\ C\ extends\ D}\ \{\ \overline{\mathsf{C}\ \mathsf{f}};\ \mathsf{K}\ \overline{\mathsf{M}}\ \} \qquad \mathsf{M} = \mathsf{B\ m}(\overline{\mathsf{B}\ \mathsf{x}})\ \{\ \mathsf{return\ t};\ \} \quad \mathsf{M} \in \overline{\mathsf{M}} \qquad \mathcal{A} \to AT(\mathsf{M})}{\mathit{mtype}(\mathsf{m}, \mathsf{C}, \mathcal{A}) = \overline{\mathsf{B}\ \mathsf{x}} \to \mathsf{B}}$$

$$\frac{CT(\mathsf{C}) = \mathsf{class\ C\ extends\ D}\ \{\ \overline{\mathsf{C}\ \mathsf{f}};\ \mathsf{K}\ \overline{\mathsf{M}}\ \} \qquad \mathsf{M} = \mathsf{B\ m}(\overline{\mathsf{B}\ \mathsf{x}})\ \{\ \mathsf{return\ t};\ \} \quad \mathsf{M} \in \overline{\mathsf{M}} \qquad \neg(\mathcal{A} \to AT(\mathsf{M}))}{\mathit{mtype}(\mathsf{m}, \mathsf{C}, \mathcal{A}) = \mathit{mtype}(\mathsf{m}, \mathsf{D}, \mathcal{A})}$$

$$\frac{CT(\mathsf{C}) = \mathsf{class\ C\ extends\ D}\ \{\ \overline{\mathsf{C}\ \mathsf{f}};\ \mathsf{K}\ \overline{\mathsf{M}}\ \} \qquad \mathsf{m}\text{ is not defined in }\overline{\mathsf{M}}}{\mathit{mtype}(\mathsf{m}, \mathsf{C}, \mathcal{A}) = \mathit{mtype}(\mathsf{m}, \mathsf{D}, \mathcal{A})}$$

Method overr. $\boxed{\mathit{override}(\mathsf{m}, \mathsf{C}, \overline{\mathsf{C}\ \mathsf{x}} \to \mathsf{C}, \mathcal{A})}$

$$\frac{}{\mathit{override}(\mathsf{m}, \mathsf{Object}, \overline{\mathsf{C}\ \mathsf{x}} \to \mathsf{C}_0, \mathcal{A})}$$

$$\frac{\begin{array}{c}CT(\mathsf{C}) = \mathsf{class\ C\ extends\ D}\ \{\ \overline{\mathsf{D}\ \mathsf{f}};\ \mathsf{K}\ \overline{\mathsf{M}}\ \} \\ \mathsf{M} = \mathsf{B}_0\ \mathsf{m}(\overline{\mathsf{B}\ \mathsf{g}})\ \{\ \mathsf{return\ t};\ \} \\ \mathit{override}(\mathsf{m}, \mathsf{D}, \overline{\mathsf{C}\ \mathsf{x}} \to \mathsf{C}_0, \mathcal{A}) \\ \mathsf{M} \in \overline{\mathsf{M}}\text{ implies }\overline{\mathsf{C}} = \overline{\mathsf{B}}\text{ and }\mathsf{C}_0 = \mathsf{B}_0\text{ and} \\ (\mathcal{A} \wedge AT(\mathsf{M})) \to (AT(\overline{\mathsf{C}\ \mathsf{x}}) \leftrightarrow AT(\overline{\mathsf{B}\ \mathsf{g}}))\end{array}}{\mathit{override}(\mathsf{m}, \mathsf{C}, \overline{\mathsf{C}\ \mathsf{x}} \to \mathsf{C}_0, \mathcal{A})}$$

Figure 3.10: Subtyping and Auxiliary Rules for CFJ [KAS]

The *method lookup* is enriched by a check that the method is reachable from the current code. A rule is added to make sure that otherwise the search is continued at the super class. Valid *method overriding* additionally checks that the annotations of all method parameters for the method and the super method are equivalent.

The extended typing rules can be found in Figure 3.11. A product line additionally consists of a feature model specifying valid variants. It is referred indirectly by the results of the *annotation table*. Assume a is a code fragment, then $AT(\mathsf{a}) = \mathcal{A} \wedge \mathcal{FM}$, where $\mathcal{A}$ is the annotation and $\mathcal{FM}$ is a propositional formula representing the feature model (see Section 2.1.1). The annotation $\mathcal{A}$ of a code fragment is the conjunction of all annotated features F_i, i.e., $\mathcal{A} = F_1 \wedge \ldots \wedge F_n$.

A *product line* is is well-typed, if all classes in the class table are well-typed and if the start term is well-typed. The start term is typed under the empty annotation and the empty context, both denoted as $\emptyset$. The empty annotation is used since since the start term has no surrounding code fragments that could be annotated.

Class typing needs a couple of checks. If a class is present, (a) the super class must be present, (b) the fields, the field assignments and the according constructor parameters must have equivalent annotations, and (c) the fields of the super class, the super call parameters and the according constructor parameters must have equivalent annotations.

That three annotations a, b, and c are equivalent is denoted by Kästner and Apel as $\mathsf{a} \leftrightarrow \mathsf{b} \leftrightarrow \mathsf{c}$ [KA08]. Note, that biconditional is not associative, i.e., $(\mathsf{a} \leftrightarrow \mathsf{b}) \leftrightarrow \mathsf{c}$ is not equal to $\mathsf{a} \leftrightarrow (\mathsf{b} \leftrightarrow \mathsf{c})$. Therefore, we have to specify the semantics, which is $(\mathsf{a} \leftrightarrow \mathsf{b}) \wedge (\mathsf{b} \leftrightarrow \mathsf{c})$.

Furthermore, if (a) a method, (b) a field, (c) a constructor parameter, (d) a super call parameter, or (e) a field assignment is present, the surrounding class must also be present. For all fields and also for the fields of the super class, the type must be present if the field is present.

In *method typing* it is forced that the return type is present, whenever the method is present. The parameter type and the method itself is present, if one of its parameters is present. For typing the methods outermost term, each element in the context is enriched with an annotation stating in which variants the type exists. Moreover, our rules for term typing get a new element—the annotation of the current context, which is the methods annotation.

The rules for *term typing* are extended as follows. A variable is well-typed if it is present in all variants of the current context. For a field access, the field and its type must be present in all variants where the access occurs. If the parameters of a method invocation are present also the invocation needs to be present. Whenever the method invocation is present, the parameter terms and the parameters at the methods declaration must have equivalent annotations. Both checks also occur for object creation terms and additionally the type of the object must be present, whenever the object creation is present. For casting, an additional premise makes sure that the target type is present in the current context.

Term typing $\boxed{\mathcal{A}; \Gamma \vdash \mathsf{t} : \mathsf{C}}$

$$\frac{\mathsf{x} : \mathsf{C}\ \mathit{with}\ \mathcal{A}' \in \Gamma \quad \mathcal{A} \to \mathcal{A}'}{\mathcal{A}; \Gamma \vdash \mathsf{x} : \mathsf{C}} \quad \textsc{(T-Var)}$$

$$\frac{\begin{array}{c}\mathcal{A}; \Gamma \vdash \mathsf{t}_0 : \mathsf{C}_0 \quad \mathit{fields}(\mathsf{C}_0) = \overline{\mathsf{C}\ \mathsf{f}} \\ \mathcal{A} \to \mathit{AT}(\mathsf{C}_i\ \mathsf{f}_i) \quad \mathcal{A} \to \mathit{AT}(\mathsf{C}_i)\end{array}}{\mathcal{A}; \Gamma \vdash \mathsf{t}_0.\mathsf{f}_i : \mathsf{C}_i} \quad \textsc{(T-Field)}$$

$$\frac{\begin{array}{c}\mathcal{A}; \Gamma \vdash \mathsf{t}_0 : \mathsf{C}_0 \\ \mathit{mtype}(\mathsf{m}, \mathsf{C}_0, \mathcal{A}) = \overline{\mathsf{D}}\ \overline{\mathsf{y}} \to \mathsf{C} \\ \mathit{AT}(\bar{\mathsf{t}}); \Gamma \vdash \bar{\mathsf{t}} : \overline{\mathsf{C}} \quad \overline{\mathsf{C}} <: \overline{\mathsf{D}} \\ \mathcal{A} \to \left(\mathit{AT}(\bar{\mathsf{t}}) \leftrightarrow \mathit{AT}(\overline{\mathsf{D}\ \mathsf{y}})\right) \\ \mathit{AT}(\bar{\mathsf{t}}) \to \mathcal{A}\end{array}}{\mathcal{A}; \Gamma \vdash \mathsf{t}_0.\mathsf{m}(\bar{\mathsf{t}}) : \mathsf{C}} \quad \textsc{(T-Invk)}$$

$$\frac{\begin{array}{c}\mathcal{A} \to \mathit{AT}(\mathsf{C}) \\ \mathit{fields}(\mathsf{C}) = \overline{\mathsf{D}\ \mathsf{f}} \\ \mathit{AT}(\bar{\mathsf{t}}); \Gamma \vdash \bar{\mathsf{t}} : \overline{\mathsf{C}} \quad \overline{\mathsf{C}} <: \overline{\mathsf{D}} \\ \mathcal{A} \to \left(\mathit{AT}(\bar{\mathsf{t}}) \leftrightarrow \mathit{AT}(\overline{\mathsf{D}\ \mathsf{f}})\right) \\ \mathit{AT}(\bar{\mathsf{t}}) \to \mathcal{A}\end{array}}{\mathcal{A}; \Gamma \vdash \mathsf{new}\ \mathsf{C}(\bar{\mathsf{t}}) : \mathsf{C}} \quad \textsc{(T-New)}$$

$$\frac{\begin{array}{c}\mathcal{A}; \Gamma \vdash \mathsf{t}_0 : \mathsf{D} \quad \mathsf{D} <: \mathsf{C} \\ \mathcal{A} \to \mathit{AT}(\mathsf{C})\end{array}}{\mathcal{A}; \Gamma \vdash (\mathsf{C})\mathsf{t}_0 : \mathsf{C}} \quad \textsc{(T-UCast)}$$

$$\frac{\begin{array}{c}\mathcal{A}; \Gamma \vdash \mathsf{t}_0 : \mathsf{D} \quad \mathsf{C} <: \mathsf{D} \\ \mathsf{C} \neq \mathsf{D} \quad \mathcal{A} \to \mathit{AT}(\mathsf{C})\end{array}}{\mathcal{A}; \Gamma \vdash (\mathsf{C})\mathsf{t}_0 : \mathsf{C}} \quad \textsc{(T-DCast)}$$

Method typing $\boxed{\mathsf{M\ OK\ in\ C}}$

$$\frac{\begin{array}{c}\mathsf{M} = \mathsf{C}_0\ \mathsf{m}(\overline{\mathsf{C}\ \mathsf{x}})\ \{\ \mathsf{return}\ \mathsf{t}_0;\ \} \\ \mathit{AT}(\mathsf{M}) = \mathcal{A} \quad \mathcal{A} \to \mathit{AT}(\mathsf{C}_0) \\ \mathit{AT}(\overline{\mathsf{C}\ \mathsf{x}}) \to \mathit{AT}(\overline{\mathsf{C}}) \quad \mathit{AT}(\overline{\mathsf{C}\ \mathsf{x}}) \to \mathcal{A} \\ \mathit{CT}(\mathsf{C}) = \mathsf{class}\ \mathsf{C}\ \mathsf{extends}\ \mathsf{D}\ \{\ \ldots\} \\ \mathit{override}(\mathsf{m}, \mathsf{D}, \overline{\mathsf{C}\ \mathsf{x}} \to \mathsf{C}_0, \mathcal{A}) \\ \Gamma = \bar{\mathsf{x}} : \overline{\mathsf{C}}\ \mathit{with}\ \mathit{AT}(\overline{\mathsf{C}\ \mathsf{x}}), \\ \mathsf{this} : \mathsf{C}\ \mathit{with}\ \mathit{AT}(\mathsf{C}) \\ \mathcal{A}; \Gamma \vdash \mathsf{t}_0 : \mathsf{E}_0 \quad \mathsf{E}_0 <: \mathsf{C}_0\end{array}}{\mathsf{M\ OK\ in\ C}}$$

Class typing $\boxed{\mathsf{C\ OK}}$

$$\frac{\begin{array}{c}\mathsf{K} = \mathsf{C}(\overline{\mathsf{D}\ \mathsf{g}}, \overline{\mathsf{C}\ \mathsf{f'}})\ \{\ \mathsf{super}(\overline{\mathsf{g'}});\ \overline{\mathsf{this.f{=}f}};\ \} \\ \overline{\mathsf{M}}\ \mathsf{OK\ in\ C} \quad \mathit{fields}(\mathsf{D}) = \overline{\mathsf{D}\ \mathsf{g''}} \\ \mathit{AT}(\mathsf{C}) = \mathcal{A} \quad \mathcal{A} \to \mathit{AT}(\mathsf{D}) \\ \mathit{AT}(\overline{\mathsf{M}}) \to \mathcal{A} \quad \mathit{AT}(\overline{\mathsf{C}\ \mathsf{f}}) \to \mathit{AT}(\overline{\mathsf{C}}) \quad \mathcal{A} \to \\ \left(\mathit{AT}(\overline{\mathsf{C}\ \mathsf{f}}) \leftrightarrow \mathit{AT}(\overline{\mathsf{this.f{=}f}}) \leftrightarrow \mathit{AT}(\overline{\mathsf{C}\ \mathsf{f'}})\right) \\ \overline{\mathsf{C}\ \mathsf{f}} = \overline{\mathsf{C}\ \mathsf{f'}} \quad \mathit{AT}(\overline{\mathsf{C}\ \mathsf{f}}) \to \mathcal{A} \\ \mathcal{A} \to \left(\mathit{AT}(\overline{\mathsf{D}\ \mathsf{g}}) \leftrightarrow \mathit{AT}(\overline{\mathsf{g'}}) \leftrightarrow \mathit{AT}(\overline{\mathsf{D}\ \mathsf{g''}})\right) \\ \overline{\mathsf{D}\ \mathsf{g}} = \overline{\mathsf{D}\ \mathsf{g''}} \quad \bar{\mathsf{g}} = \overline{\mathsf{g'}} \quad \mathit{AT}(\overline{\mathsf{D}\ \mathsf{g}}) \to \mathcal{A} \\ \mathit{AT}(\overline{\mathsf{D}\ \mathsf{g}}) \to \mathit{AT}(\overline{\mathsf{D}}) \quad \mathit{AT}(\overline{\mathsf{C}\ \mathsf{f'}}) \to \mathcal{A} \\ \mathit{AT}(\overline{\mathsf{g'}}) \to \mathcal{A} \quad \mathit{AT}(\overline{\mathsf{this.f{=}f}}) \to \mathcal{A}\end{array}}{\mathsf{class}\ \mathsf{C}\ \mathsf{extends}\ \mathsf{D}\ \{\ \overline{\mathsf{C}\ \mathsf{f}};\ \mathsf{K}\ \overline{\mathsf{M}}\ \}\ \mathsf{OK}}$$

Product line typing $\boxed{(\overline{\mathsf{L}}, \mathsf{t}, \mathsf{FM})}$

$$\frac{\overline{\mathsf{L}}\ \mathsf{OK} \quad \emptyset; \emptyset \vdash \mathsf{t} : \mathsf{C}}{(\overline{\mathsf{L}}, \mathsf{t}, \mathsf{FM})}$$

Figure 3.11: Typing Rules for CFJ [KAS]

The reachability check in T-UCast is redundant and can be removed without changing the typing of CFJ product lines. We found some simplifications like this, for which we give a proof in the following section. Based on that we present revised typing rules for CFJ.

3.3 Simplifications of the Type System

The type system for CFJ given in Section 3.2.4 can be simplified. While working with the type system, we encountered redundant premises in some inference rules. For other rules we can provide simplifications for reachability checks based on propositional formulas.

We speak of simplifications as all the intended changes result in a more compact representation of the inference rules. Hence, the representation of the type system itself is more compact and a bit more readable. Although our scope is not on implementations of type checkers, we also expect a small speedup for type-checking a CFJ product line as a type checker needs to verify all (redundant) premises for each inference rule that is applied.

We mentioned earlier that FJ is not intended to be used in industrial practice and so CFJ is not either. Therefore a speedup of an implementation seems not useful. But (a) the type system might be extended to check product lines and programs with additional language constructs, e.g., interfaces and (b) some of our rules can also be applied without further adaptions when CFJ is extended, e.g., adding a base type (like Java's **`int`**) has no influence on checking method parameter annotations of declarations and invocations.

We have the choice to prove that our changes lead to an equivalent type system (a) informally or (b) formally using Coq. Using Coq we can (i) formalize both type system at once and proof the equivalence or (ii) just formalize the parts where our proofs rely on. (ii) is not feasible for all our proofs as some rely on almost all inference rules. In the following chapter, we formalize the revised type system, but (i) would require that we additionally formalize the original type system and that we formally proof the equivalence even in parts that are unchanged. As our changes are rather simple, we decided to give informal proofs.

First, we manually proof that our changes lead to an equivalent type system. Section 3.3.1 presents changes to casting and field access rules. In Section 3.3.2 and Section 3.3.3, rewrites for premises in method and class typing are given and proved. Second, we summarize our changes by proposing a revised type system in Section 3.3.4.

3.3.1 Casting and Field Access

Let us take a closer look at the typing rule T-UCast checking the type soundness for up-casts. Contrary to the rules of FJ, there is a premise added to make sure that the type to cast to is reachable in all variants. In the following, we prove that this premise is redundant. But first we prove a lemma stating that a premise implicitly holds for all typed terms.

Lemma 3.1. *The following implication holds.*

$$(\mathcal{A}; \Gamma \vdash t : C) \rightarrow \big(\mathcal{A} \rightarrow AT(C)\big)$$

Proof. We proof by induction on the term t.

Our induction hypothesis is

$$(\mathcal{A}; \Gamma \vdash \mathsf{t} : \mathsf{C}) \rightarrow \big(\mathcal{A} \rightarrow AT(\mathsf{C})\big) \tag{3.10}$$

We show the induction beginning using a case analysis.

(B1) $\mathcal{A}; \Gamma \vdash \mathsf{x} : \mathsf{C}$. Using rule T-Var we get $\mathsf{x} : \mathsf{C}$ *with* $\mathcal{A}' \in \Gamma$ and $\mathcal{A} \rightarrow \mathcal{A}'$. Context is only created in method typing. We analyze the cases that occur.

(a) $\mathsf{this} : \mathsf{C}$ *with* $AT(\mathsf{C})$. Hence $\mathcal{A}' = AT(\mathsf{C})$ we have $\mathcal{A} \rightarrow AT(\mathsf{C})$.

(b) $\overline{\mathsf{x}} : \overline{\mathsf{C}}$ *with* $AT(\overline{\mathsf{C}\ \mathsf{x}})$. Hence $\mathcal{A}' = AT(\overline{\mathsf{C}\ \mathsf{x}})$ we have $\mathcal{A} \rightarrow AT(\overline{\mathsf{C}\ \mathsf{x}})$. Furthermore, in method typing we have $AT(\overline{\mathsf{C}\ \mathsf{x}}) \rightarrow AT(\overline{\mathsf{C}})$. With transitivity of propositional formulas we obtain $\mathcal{A} \rightarrow AT(\overline{\mathsf{C}})$. Rewriting our list notation we get $\mathsf{x}_i : \mathsf{C}_i$ *with* $AT(\mathsf{C}_i\ \mathsf{x}_i)$ and $\mathcal{A} \rightarrow AT(\mathsf{C}_i)$ for all i, what is exactly what we wanted to show.

(B2) $\mathcal{A}; \Gamma \vdash \mathsf{t.f} : \mathsf{C}$. T-Field directly leads us to $\mathcal{A} \rightarrow AT(\mathsf{C})$.

(B3) $\mathcal{A}; \Gamma \vdash \mathsf{new}\ \mathsf{C}(\ldots) : \mathsf{C}$. T-New directly leads us to $\mathcal{A} \rightarrow AT(\mathsf{C})$.

(B4) $\mathcal{A}; \Gamma \vdash (\mathsf{D})\mathsf{t} : \mathsf{D}$. We give a proof for up- and down-casts separately. Note, that contrary to the inferences rules of the type system we have switched C and D.

(a) $\mathsf{C} <: \mathsf{D}$. T-UCast directly leads us to $\mathcal{A} \rightarrow AT(\mathsf{D})$.

(b) $\mathsf{D} <: \mathsf{C} \wedge \mathsf{C} \neq \mathsf{D}$. T-DCast directly leads us to $\mathcal{A} \rightarrow AT(\mathsf{D})$.

For the induction step there is just one case left to proof: $\mathcal{A}; \Gamma \vdash \mathsf{t.m}(\ldots) : \mathsf{C}$. T-Invk forces that $\mathcal{A}; \Gamma \vdash \mathsf{t} : \mathsf{C}_0$. Applying our induction hypothesis in Equation 3.10, we get $\mathcal{A} \rightarrow AT(\mathsf{C}_0)$. □

Theorem 3.2. *Assume $\mathcal{A}; \Gamma \vdash t : D$ and $D <: C$, the following formula holds*

$$\mathcal{A} \rightarrow AT(C).$$

Proof. Class typing makes sure that for every class F the super class G is always reachable, i.e., $AT(\mathsf{F}) \rightarrow AT(\mathsf{G})$. Because of transitivity, this also holds if G is not a direkt super class, as long as $\mathsf{F} <: \mathsf{G}$. Thus, $\mathsf{D} <: \mathsf{C}$ leads to

$$AT(\mathsf{D}) \rightarrow AT(\mathsf{C}). \tag{3.11}$$

Applying Lemma 3.1 to $\mathcal{A}; \Gamma \vdash \mathsf{t} : \mathsf{D}$, we omit $\mathcal{A} \rightarrow AT(\mathsf{D})$. Together with Equation 3.11 we can conclude $\mathcal{A} \rightarrow AT(\mathsf{C})$. □

The check that the type of a field must be present in T-FIELD is redundant as it is already checked for classes. In the following we give a proof.

Theorem 3.3. *Given a well-typed CFJ product line $(\overline{L}, t, FM)$, a class $C \in \overline{L}$ and a field access $t_0.f_i$ anywhere at the class table $\overline{L}$ or in the start term t, then the following implication holds.*

$$\left[(\mathcal{A};\Gamma \vdash t_0 : C) \wedge (\mathit{fields}(C) = \overline{C\ f}) \wedge \left(\mathcal{A} \rightarrow AT(C_i\ f_i)\right)\right] \rightarrow \left(\mathcal{A} \rightarrow AT(C_i)\right)$$

Proof. Since $(\overline{\mathsf{L}}, \mathsf{t}, \mathsf{FM})$ is well-typed and $\mathsf{C} \in \overline{\mathsf{L}}$, we know that

$$\mathsf{class\ C\ extends\ D\ \{\ \overline{C\ f};\ K\ \overline{M}\ \}\ OK.}$$

Therefore, $AT(\overline{\mathsf{C\ f}}) \rightarrow AT(\overline{\mathsf{C}})$ and especially, $AT(\mathsf{C}_i\ \mathsf{f}_i) \rightarrow AT(\mathsf{C}_i)$. Together with $\mathcal{A} \rightarrow AT(\mathsf{C}_i\ \mathsf{f}_i)$ we get $\mathcal{A} \rightarrow AT(\mathsf{C}_i)$. □

Note, that removing the redundant premise does not affect the proof of Theorem 3.2 as the premise can be expressed as shown in the proof of Theorem 3.3.

3.3.2 Method Typing

Method typing has two redundant premises. We give a proof for both premises separately.

Theorem 3.4. *The premise $\mathcal{A} \wedge AT(M)$ in method overriding can be simplified to $AT(M)$ without changing the type systems behaviour.*

Proof. Valid method overriding is formalized in Figure 3.10, where a reachability check contains $\mathcal{A} \wedge AT(\mathsf{M})$. Method overriding is only called from method typing in Figure 3.11 and always with $\mathcal{A} = AT(\mathsf{M})$. Therefore, we get $AT(\mathsf{M}) \wedge AT(\mathsf{M})$, which can be simplified to $AT(\mathsf{M})$. □

Theorem 3.5. *The premise $\mathcal{A} \rightarrow AT(C_0)$ in method typing is redundant.*

Proof. On of the premises in method typing is $\mathcal{A};\Gamma \vdash \mathsf{t}_0 : \mathsf{C}_0$. Lemma 3.1 states that $\mathcal{A} \rightarrow AT(\mathsf{C}_0)$ is fulfilled. Therefore, the premise is redundant. □

3.3.3 Class Typing

In class typing we found premises that can be removed or simplified. We proof that one premise is redundant and that two sets of premises can be rewritten.

Theorem 3.6. *The following premise in class typing is redundant.*

$$AT(\overline{D\ g}) \rightarrow AT(\overline{D}) \tag{3.12}$$

Proof. Transitivity of
$$AT(\overline{\mathsf{D\ g}}) \rightarrow \mathcal{A}$$
and
$$\mathcal{A} \rightarrow \big(AT(\overline{\mathsf{D\ g}}) \leftrightarrow AT(\overline{\mathsf{g'}}) \leftrightarrow AT(\overline{\mathsf{D\ g''}})\big)$$
leads to
$$AT(\overline{\mathsf{D\ g}}) \rightarrow \big(AT(\overline{\mathsf{D\ g}}) \leftrightarrow AT(\overline{\mathsf{g'}}) \leftrightarrow AT(\overline{\mathsf{D\ g''}})\big).$$
Hence we can conclude that
$$AT(\overline{\mathsf{D\ g}}) \rightarrow AT(\overline{\mathsf{D\ g''}}). \tag{3.13}$$
Additionally, the rule
$$AT(\overline{\mathsf{C\ f}}) \rightarrow AT(\overline{\mathsf{C}}) \text{ in } \mathsf{C\ OK}$$
corresponds to the rule
$$AT(\overline{\mathsf{D\ f''}}) \rightarrow AT(\overline{\mathsf{D}}) \text{ in } \mathsf{D\ OK}. \tag{3.14}$$
With Equation 3.13 and Equation 3.14 we shown that Equation 3.12 is already contained in other premises and thus redundant. □

Theorem 3.7. *The equality $P \equiv Q$ holds, where*
$$P = \big(\mathcal{A} \rightarrow (\mathcal{B} \leftrightarrow \mathcal{C} \leftrightarrow \mathcal{D})\big) \wedge (\mathcal{B} \rightarrow \mathcal{A}) \wedge (\mathcal{C} \rightarrow \mathcal{A}) \wedge (\mathcal{D} \rightarrow \mathcal{A})$$
and
$$Q = (\mathcal{B} \leftrightarrow \mathcal{C}) \wedge (\mathcal{B} \leftrightarrow \mathcal{D}) \wedge (\mathcal{B} \rightarrow \mathcal{A}).$$
Especially with $\mathcal{B} = AT(\overline{\mathsf{C\ f}})$, $\mathcal{C} = AT(\overline{\mathsf{this.f{=}f}})$, and $\mathcal{D} = AT(\overline{\mathsf{C\ f'}})$.

Proof. We give a proof using the truth value table.

$\mathcal{A}$	$\mathcal{B}$	$\mathcal{C}$	$\mathcal{D}$	P	Q
0	0	0	0	1	1
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	0
0	1	0	0	0	0
0	1	0	1	0	0
0	1	1	0	0	0
0	1	1	1	0	0

$\mathcal{A}$	$\mathcal{B}$	$\mathcal{C}$	$\mathcal{D}$	P	Q
1	0	0	0	1	1
1	0	0	1	0	0
1	0	1	0	0	0
1	0	1	1	0	0
1	1	0	0	0	0
1	1	0	1	0	0
1	1	1	0	0	0
1	1	1	1	1	1

□

Theorem 3.8. *The equality $P \equiv Q$ holds, where*
$$P = \big(\mathcal{A} \rightarrow (\mathcal{B} \leftrightarrow \mathcal{C} \leftrightarrow \mathcal{D})\big) \wedge (\mathcal{B} \rightarrow \mathcal{A}) \wedge (\mathcal{C} \rightarrow \mathcal{A})$$
and
$$Q = (\mathcal{B} \leftrightarrow \mathcal{C}) \wedge \big(\mathcal{A} \rightarrow (\mathcal{B} \leftrightarrow \mathcal{D})\big) \wedge (\mathcal{B} \rightarrow \mathcal{A}).$$
Especially with $\mathcal{B} = AT(\overline{\mathsf{D\ g}})$, $\mathcal{C} = AT(\overline{\mathsf{g'}})$, and $\mathcal{D} = AT(\overline{\mathsf{D\ g''}})$.

Proof. Again, we give a proof using the truth value table.

$\mathcal{A}$	$\mathcal{B}$	$\mathcal{C}$	$\mathcal{D}$	P	Q
0	0	0	0	1	1
0	0	0	1	1	1
0	0	1	0	0	0
0	0	1	1	0	0
0	1	0	0	0	0
0	1	0	1	0	0
0	1	1	0	0	0
0	1	1	1	0	0

$\mathcal{A}$	$\mathcal{B}$	$\mathcal{C}$	$\mathcal{D}$	P	Q
1	0	0	0	1	1
1	0	0	1	0	0
1	0	1	0	0	0
1	0	1	1	0	0
1	1	0	0	0	0
1	1	0	1	0	0
1	1	1	0	0	0
1	1	1	1	1	1

□

3.3.4 Revised Type System

In this section, we propose our revised type system, which is largely the same as the type system by Kästner and Apel. But as we use this revised type system for our formalization in Coq (see Chapter 4), we want to present it as one piece. This might also be appreciated by future work that relies on our type system.

In Figure 3.12, we present the revised subtyping and auxiliary rules for CFJ. We only applied Theorem 3.4, i.e., we replace $\mathcal{A} \wedge AT(\mathsf{M})$ with $AT(\mathsf{M})$. Additionally, the annotation as a parameter is no longer needed and removed.

The revised typing rules for CFJ can be found in Figure 3.13. At the rules T-UCast and T-Field we omit the redundant reachability check according to Theorem 3.2 and Theorem 3.3. According to Theorem 3.5, we remove the redundant premise from method typing. For class typing we apply the equalities from Theorem 3.7 and Theorem 3.8 as well as we omit the redundant premise known from Theorem 3.6.

Furthermore, we added the annotation table to the tuple that represents a CFJ product line, because it has an influence on the FJ programs we can generate and also whether the product line is well-typed or not. It also needs to be inserted at the syntax of CFJ, but we omit a repetition of the syntax.

Subtyping $\boxed{\mathsf{C} <: \mathsf{D}}$

$$\frac{}{\mathsf{C} <: \mathsf{C}}$$

$$\frac{\mathsf{C} <: \mathsf{D} \quad \mathsf{D} <: \mathsf{E}}{\mathsf{C} <: \mathsf{E}}$$

$$\frac{\mathsf{class\ C\ extends\ D}\ \{\ \ldots\}}{\mathsf{C} <: \mathsf{D}}$$

Field lookup $\boxed{\mathit{fields}(\mathsf{C}) = \overline{\mathsf{C}\ \mathsf{f}}}$

$$\frac{}{\mathit{fields}(\mathsf{Object}) = \bullet}$$

$$\frac{\begin{array}{c} CT(\mathsf{C}) = \mathsf{class\ C\ extends\ D}\ \{\ \overline{\mathsf{C\ f}};\ \mathsf{K}\ \overline{\mathsf{M}}\ \} \\ \mathit{fields}(\mathsf{D}) = \overline{\mathsf{D\ g}} \end{array}}{\mathit{fields}(\mathsf{C}) = \overline{\mathsf{D\ g}}, \overline{\mathsf{C\ f}}}$$

Method lookup $\boxed{\mathit{mtype}(\mathsf{m}, \mathsf{C}, \mathcal{A}) = \overline{\mathsf{C\ x}} \to \mathsf{C}}$

$$\frac{\begin{array}{c} CT(\mathsf{C}) = \mathsf{class\ C\ extends\ D}\ \{\ \overline{\mathsf{C\ f}};\ \mathsf{K}\ \overline{\mathsf{M}}\ \} \\ \mathsf{M} = \mathsf{B\ m}(\overline{\mathsf{B\ x}})\ \{\ \mathsf{return\ t};\ \} \quad \mathsf{M} \in \overline{\mathsf{M}} \\ \mathcal{A} \to AT(\mathsf{M}) \end{array}}{\mathit{mtype}(\mathsf{m}, \mathsf{C}, \mathcal{A}) = \overline{\mathsf{B\ x}} \to \mathsf{B}}$$

$$\frac{\begin{array}{c} CT(\mathsf{C}) = \mathsf{class\ C\ extends\ D}\ \{\ \overline{\mathsf{C\ f}};\ \mathsf{K}\ \overline{\mathsf{M}}\ \} \\ \mathsf{M} = \mathsf{B\ m}(\overline{\mathsf{B\ x}})\ \{\ \mathsf{return\ t};\ \} \quad \mathsf{M} \in \overline{\mathsf{M}} \\ \neg(\mathcal{A} \to AT(\mathsf{M})) \end{array}}{\mathit{mtype}(\mathsf{m}, \mathsf{C}, \mathcal{A}) = \mathit{mtype}(\mathsf{m}, \mathsf{D}, \mathcal{A})}$$

$$\frac{\begin{array}{c} CT(\mathsf{C}) = \mathsf{class\ C\ extends\ D}\ \{\ \overline{\mathsf{C\ f}};\ \mathsf{K}\ \overline{\mathsf{M}}\ \} \\ \mathsf{m}\ \text{is not defined in}\ \overline{\mathsf{M}} \end{array}}{\mathit{mtype}(\mathsf{m}, \mathsf{C}, \mathcal{A}) = \mathit{mtype}(\mathsf{m}, \mathsf{D}, \mathcal{A})}$$

Method overriding $\boxed{\mathit{override}(\mathsf{m}, \mathsf{C}, \overline{\mathsf{C\ x}} \to \mathsf{C})}$

$$\frac{}{\mathit{override}(\mathsf{m}, \mathsf{Object}, \overline{\mathsf{C\ x}} \to \mathsf{C}_0)}$$

$$\frac{\begin{array}{c} CT(\mathsf{C}) = \mathsf{class\ C\ extends\ D}\ \{\ \overline{\mathsf{D\ f}};\ \mathsf{K}\ \overline{\mathsf{M}}\ \} \\ \mathsf{M} = \mathsf{B}_0\ \mathsf{m}(\overline{\mathsf{B\ g}})\ \{\ \mathsf{return\ t};\ \} \\ \mathit{override}(\mathsf{m}, \mathsf{D}, \overline{\mathsf{C\ x}} \to \mathsf{C}_0) \\ \mathsf{M} \in \overline{\mathsf{M}}\ \text{implies}\ \overline{\mathsf{C}} = \overline{\mathsf{B}}\ \text{and}\ \mathsf{C}_0 = \mathsf{B}_0\ \text{and} \\ AT(\mathsf{M}) \to (AT(\overline{\mathsf{C\ x}}) \leftrightarrow AT(\overline{\mathsf{B\ g}})) \end{array}}{\mathit{override}(\mathsf{m}, \mathsf{C}, \overline{\mathsf{C\ x}} \to \mathsf{C}_0)}$$

Figure 3.12: Our Revised Subtyping and Auxiliary Rules for CFJ

Term typing $\boxed{\mathcal{A}; \Gamma \vdash \mathsf{t} : \mathsf{C}}$

$$\frac{\mathsf{x} : \mathsf{C}\ \textit{with}\ \mathcal{A}' \in \Gamma \quad \mathcal{A} \rightarrow \mathcal{A}'}{\mathcal{A}; \Gamma \vdash \mathsf{x} : \mathsf{C}} \quad \text{(T-Var)}$$

$$\frac{\begin{array}{c}\mathcal{A}; \Gamma \vdash \mathsf{t}_0 : \mathsf{C}_0 \quad \textit{fields}(\mathsf{C}_0) = \overline{\mathsf{C}\ \mathsf{f}} \\ \mathcal{A} \rightarrow AT(\mathsf{C}_i\ \mathsf{f}_i)\end{array}}{\mathcal{A}; \Gamma \vdash \mathsf{t}_0.\mathsf{f}_i : \mathsf{C}_i} \quad \text{(T-Field)}$$

$$\frac{\begin{array}{c}\mathcal{A}; \Gamma \vdash \mathsf{t}_0 : \mathsf{C}_0 \\ \textit{mtype}(\mathsf{m}, \mathsf{C}_0, \mathcal{A}) = \overline{\mathsf{D}\ \mathsf{y}} \rightarrow \mathsf{C} \\ AT(\bar{\mathsf{t}}); \Gamma \vdash \bar{\mathsf{t}} : \bar{\mathsf{C}} \quad \bar{\mathsf{C}} <: \bar{\mathsf{D}} \\ \mathcal{A} \rightarrow \left(AT(\bar{\mathsf{t}}) \leftrightarrow AT(\overline{\mathsf{D}\ \mathsf{y}})\right) \\ AT(\bar{\mathsf{t}}) \rightarrow \mathcal{A}\end{array}}{\mathcal{A}; \Gamma \vdash \mathsf{t}_0.\mathsf{m}(\bar{\mathsf{t}}) : \mathsf{C}} \quad \text{(T-Invk)}$$

$$\frac{\begin{array}{c}\mathcal{A} \rightarrow AT(\mathsf{C}) \\ \textit{fields}(\mathsf{C}) = \overline{\mathsf{D}\ \mathsf{f}} \\ AT(\bar{\mathsf{t}}); \Gamma \vdash \bar{\mathsf{t}} : \bar{\mathsf{C}} \quad \bar{\mathsf{C}} <: \bar{\mathsf{D}} \\ \mathcal{A} \rightarrow \left(AT(\bar{\mathsf{t}}) \leftrightarrow AT(\overline{\mathsf{D}\ \mathsf{f}})\right) \\ AT(\bar{\mathsf{t}}) \rightarrow \mathcal{A}\end{array}}{\mathcal{A}; \Gamma \vdash \mathsf{new}\ \mathsf{C}(\bar{\mathsf{t}}) : \mathsf{C}} \quad \text{(T-New)}$$

$$\frac{\mathcal{A}; \Gamma \vdash \mathsf{t}_0 : \mathsf{D} \quad \mathsf{D} <: \mathsf{C}}{\mathcal{A}; \Gamma \vdash (\mathsf{C})\mathsf{t}_0 : \mathsf{C}} \quad \text{(T-UCast)}$$

$$\frac{\begin{array}{c}\mathcal{A}; \Gamma \vdash \mathsf{t}_0 : \mathsf{D} \quad \mathsf{C} <: \mathsf{D} \\ \mathsf{C} \neq \mathsf{D} \quad \mathcal{A} \rightarrow AT(\mathsf{C})\end{array}}{\mathcal{A}; \Gamma \vdash (\mathsf{C})\mathsf{t}_0 : \mathsf{C}} \quad \text{(T-DCast)}$$

Method typing $\boxed{\mathsf{M}\ \mathsf{OK}\ \mathsf{in}\ \mathsf{C}}$

$$\frac{\begin{array}{c}\mathsf{M} = \mathsf{C}_0\ \mathsf{m}(\overline{\mathsf{C}\ \mathsf{x}})\ \{\ \mathsf{return}\ \mathsf{t}_0;\ \} \\ AT(\mathsf{M}) = \mathcal{A} \\ AT(\overline{\mathsf{C}\ \mathsf{x}}) \rightarrow AT(\bar{\mathsf{C}}) \quad AT(\overline{\mathsf{C}\ \mathsf{x}}) \rightarrow \mathcal{A} \\ CT(\mathsf{C}) = \mathsf{class}\ \mathsf{C}\ \mathsf{extends}\ \mathsf{D}\ \{\ \ldots \} \\ \textit{override}(\mathsf{m}, \mathsf{D}, \overline{\mathsf{C}\ \mathsf{x}} \rightarrow \mathsf{C}_0) \\ \Gamma = \bar{\mathsf{x}} : \bar{\mathsf{C}}\ \textit{with}\ AT(\overline{\mathsf{C}\ \mathsf{x}}), \\ \mathsf{this} : \mathsf{C}\ \textit{with}\ AT(\mathsf{C}) \\ \mathcal{A}; \Gamma \vdash \mathsf{t}_0 : \mathsf{E}_0 \quad \mathsf{E}_0 <: \mathsf{C}_0\end{array}}{\mathsf{M}\ \mathsf{OK}\ \mathsf{in}\ \mathsf{C}}$$

Class typing $\boxed{\mathsf{C}\ \mathsf{OK}}$

$$\frac{\begin{array}{c}\mathsf{K} = \mathsf{C}(\overline{\mathsf{D}\ \mathsf{g}}, \overline{\mathsf{C}\ \mathsf{f'}})\ \{\ \mathsf{super}(\overline{\mathsf{g'}});\ \overline{\mathsf{this.f{=}f}};\ \} \\ \bar{\mathsf{M}}\ \mathsf{OK}\ \mathsf{in}\ \mathsf{C} \quad \textit{fields}(\mathsf{D}) = \overline{\mathsf{D}\ \mathsf{g''}} \\ \overline{\mathsf{C}\ \mathsf{f}} = \overline{\mathsf{C}\ \mathsf{f'}} \quad \overline{\mathsf{D}\ \mathsf{g}} = \overline{\mathsf{D}\ \mathsf{g''}} \quad \bar{\mathsf{g}} = \overline{\mathsf{g'}} \\ AT(\mathsf{C}) = \mathcal{A} \quad \mathcal{A} \rightarrow AT(\mathsf{D}) \\ AT(\bar{\mathsf{M}}) \rightarrow \mathcal{A} \quad AT(\overline{\mathsf{C}\ \mathsf{f}}) \rightarrow \mathcal{A} \wedge AT(\bar{\mathsf{C}}) \\ AT(\overline{\mathsf{C}\ \mathsf{f}}) \leftrightarrow AT(\overline{\mathsf{this.f{=}f}}) \\ AT(\overline{\mathsf{C}\ \mathsf{f}}) \leftrightarrow AT(\overline{\mathsf{C}\ \mathsf{f'}}) \\ \mathcal{A} \rightarrow \left(AT(\overline{\mathsf{D}\ \mathsf{g}}) \leftrightarrow AT(\overline{\mathsf{D}\ \mathsf{g''}})\right) \\ AT(\overline{\mathsf{D}\ \mathsf{g}}) \leftrightarrow AT(\overline{\mathsf{g'}}) \quad AT(\overline{\mathsf{D}\ \mathsf{g}}) \rightarrow \mathcal{A}\end{array}}{\mathsf{class}\ \mathsf{C}\ \mathsf{extends}\ \mathsf{D}\ \{\ \overline{\mathsf{C}\ \mathsf{f}};\ \mathsf{K}\ \bar{\mathsf{M}}\ \}\ \mathsf{OK}}$$

Product line typing $\boxed{(\bar{\mathsf{L}}, \mathsf{t}, \mathsf{FM}, \mathsf{AT})}$

$$\frac{\bar{\mathsf{L}}\ \mathsf{OK} \quad \emptyset; \emptyset \vdash \mathsf{t} : \mathsf{C}}{(\bar{\mathsf{L}}, \mathsf{t}, \mathsf{FM}, \mathsf{AT})}$$

Figure 3.13: Our Revised Typing Rules for CFJ

4. Formalization of Colored Featherweight Java

This chapter presents our formalization of CFJ in Coq, that is used in the following chapter to provide a machine-checked proof of type soundness for CFJ. Our formalization is based on a formalization of FJ in Coq by de Fraine [Fra09b]. We present some changes to this existing formalization and its proofs, making it consistent with the informal type system presented in the previous chapter. To formalize CFJ, we need to realize the typing rules, the variant generation and the fundamental concept of the annotation table.

Our contribution is the formalization of CFJ in Coq. This is the first time CFJ is formalized using a proof assistant. The main innovation is the formalization of the annotation table, for which only informal descriptions exist so far. While the realization of the variant generation is straightforward, the formalization of the typing rules is not, caused by the concept to formalize the annotation table. A smaller contribution is our adapted formalization for FJ.

First, we prepare a formalization of FJ in Section 4.1. The idea is to reuse an existing formalization of FJ, where we only need to create new definitions if necessary for CFJ and otherwise reuse existing ones. In Section 4.2, we present new definitions for the CFJ type system. The CFJ variant generation mechanism is formalized in Section 3.2.3. Finally, we summarize our formalization in Section 4.4.

4.1 Building on a FJ Formalization

Fortunately, formalizations of FJ in Coq already exist and two of them are publicly available. In Section 4.1.1, we justify our choice for one formalization. In Section 4.1.2, we disclose inconsistencies in this formalization according to our definitions in Section 3.1.3. We adjust the formalization and the proofs accordingly.

4.1.1 Choosing a FJ Formalization

We want to reuse one of the existing formalizations of Featherweight Java, because they are peer reviewed and accepted in the community. We found one formalization of FJ by Weirich and contacted the Coq mailing list for further formalizations, where we got a second formalization by de Fraine.[1] Both formalizations come with type soundness proofs.

The first formalization by Weirich is from 2005 and compiles with Coq 7.3.1 (October 2002) [Wei05]. It is not compatible with newer versions of Coq, i.e., the verification in Coq stops with an error. The second formalization by de Fraine is from 2009 and intended to run with Coq 8.2 (May 2009), but is also compatible with the most recent version Coq 8.2pl1 (July 2009) [Fra09a].

We decided to base our formalization on de Fraine's FJ formalization due to two reasons. First, this formalization was used as a basis for type soundness proofs of particular aspect-oriented concepts, i.e., they also wanted to prove properties for an extension of Featherweight Java [Fra09b]. Therefore, this formalization is build with an extension in mind and our extension might be easier.

Second, we want to use the most recent version of Coq, namely Coq 8.2pl1 (July 2009), because newer versions come with improvements and many other researches work with the current version. If we would use the version from 2002, we might have a problem to prove a certain theorem that is possible in newer versions. Additionally, for the recent version we have better tool support.

In principle, it is possible to port a formalization from an older version to a newer one. But in the particular case of Weirich's formalization, we were not able to. This is mostly due to missing documentations on older versions of Coq, since we need to understand the old version before we can migrate it. For the stated reasons, we base our work on de Fraine's formalization.

4.1.2 Corrections to the FJ Formalization

This section uncovers four inconsistencies of de Fraine's formalization according to the type system presented in Section 3.1.3:

1. The class table may contain references to classes that are not in the class table.
2. A method can override another method if the return type is a subtype of return type in the original method; the types do not need to by identical.
3. Casting is not formalized.
4. There is no representation of the constructor.

[1]http://logical.saclay.inria.fr/coq-puma/messages/f4840f7689b9905a

We found an example for a well-typed FJ class table according to this formalization that references classes not in the class table. We give a machine-checked proof in Appendix A.1 and propose a change to class typing that incomplete class tables are no longer classified as well-typed.

Valid method overriding in de Fraine's formalization is realized as we know it from full Java. A method overriding is valid, if the return type is a subtype of the return type of the overridden method. For FJ, the type has to be exactly the same as of the super method [Pie02, IPW01]. In Appendix A.2, we present our adapted method overriding.

Both other limitations are not so easy to adopt. They require large rewritings of proof scripts and are out of the scope of this thesis. The missing representation of the constructor involves that there is no representation of the constructor parameters, the super call parameters, or the field assignments.

Note, that for FJ the constructor is well-defined giving the fields of the class and the super class. Therefore, the annotations of constructor parameters, super call parameters and field assignments have to be equivalent to those of the particular fields. That this is fulfilled, can easily be seen at the typing rules for CFJ.

4.2 Type System

The formalization of the type system for CFJ that we present in this section, is similar to the type system for FJ formalized by de Fraine. Since we want to prove properties on both formalizations it is useful to stay as close as possible. Both type systems are formalized step-wise and we often have one definition for every typing rule.

We base our formalization on the revised typing rules in Figure 3.13 and the auxiliary rules in Figure 3.12. We present our definitions bottom-up as defined in Coq. In Section 4.2.1, we explain how we realized the fundamental concept of the annotation table in Coq. The subtyping and auxiliary rules are presented in Section 4.2.2, while Section 4.2.3 shows the formalized typing rules.

4.2.1 Realization of Annotations

This section explains how we formalized annotations in Coq. We discuss realizations of the annotation table and come up with storing annotations directly at the class table. Hence, a new definition of the class table is required as well as functions to lookup annotations at the class table, e.g., given a class or method name.

Annotation Table

Annotations are modeled as propositional formulas. The type of propositional formulas in Coq is **`Prop`**. An annotation can either be `True`, `False`, or a function returning one of these two values, while the function can represent every conceivable propositional formula based on variables of type **`Prop`**. For example, `A ∧ B` is such a function, where `A : `**`Prop`** and `B : `**`Prop`**, and the computed value is `True`, if and only if `A` and `B` are `True`.

We defined `ann` as a notation for **`Prop`** to make clear where we expect annotations (see Listing 4.1). In listings of this chapter, we highlight all parts that are of interest, i.e., parts added in CFJ to similar definitions in the FJ formalization. Many of the definitions we present in this chapter are largely the same as for FJ, which is intended as we want to prove properties between CFJ and FJ.

```
Notation ann := Prop.
```

Listing 4.1: Formalization of Annotations

In the following, we discover alternatives to realize the annotation table, i.e., how do we map code fragments to annotations. Since Kästner et al. only propose informal descriptions for the annotation table [KA08, KAS], it is a challenge to formalize this construct into a proof assistants language. For this purpose, we show how terms are formalized at the FJ formalization (see Listing 4.2).

```
Inductive exp : Set :=
| e_var : var → exp
| e_field : exp → fname → exp
| e_meth : exp → mname → list exp → exp
| e_new : cname → list exp → exp.
```

Listing 4.2: Terms in the FJ Formalization [Fra09a]

A FJ term (expression) is (a) a variable, (b) a field access, (c) a method invocation, or (d) an object creation. Field access and method invocation terms consist of an arbitrary base term, while method invocation and object creation terms handle a list of arbitrary terms, the parameter list. Each parameter term can be annotated. We need a strategy to map these terms to annotations.

In CFJ, annotations are stored in the annotation table. It maps code fragments to annotations by their node in an abstract syntax tree with the offset and length [KA08]. We simply have no abstract syntax tree to determine the absolute position of of a particular term at the class table. The problem is, that identical terms may occur at different positions with different annotations. Our example in Figure 3.7 on Page 26 contains the term `new A()` with two different annotations.

Identifying the position of a particular term would require (a) the class and method where it occurs if it is not part of the start term, (b) the position of the term in the outermost parameter list, and (c) the positions of all further parameter lists if applicable. There is a simpler solution, which we use to realize the annotation table. We store the annotations directly at each parameter list.

Terms

Terms can be annotated if they occur in a list. Since a term can itself contain a list of terms, we need to extend the inductive definition of terms. Term lists are stored for

```
Inductive a_exp : Type :=
| ae_var : var → a_exp
| ae_field : a_exp → fname → a_exp
| ae_meth : a_exp → mname → list (ann * a_exp) → a_exp
| ae_new : cname → list (ann * a_exp) → a_exp.
```

Listing 4.3: Formalization of Terms

method invocations and object creation terms. For both we simply store a list of pairs of an annotation and a term.

Whenever we need to adapt a FJ definition, we give it a new name with the prefix `a_` to state that this definition handles annotations and is part of the CFJ formalization. We do not use the same name as for FJ, because we need to distinguish between FJ and CFJ definitions in our proofs.

Class Table

We must provide a new definition for CFJ class tables for two reasons. First, the FJ class table uses the definition of terms. Second, other code fragments than terms can be annotated and we want to store those annotations accordingly. In Listing 4.4, we present our adapted definition of the class table.

```
Notation a_env := (list (var * (ann * typ))).

Notation a_flds := (list (fname * (ann * typ))).
Notation a_mths := (list (mname * (ann * (typ * a_env * a_exp)))).

Notation a_ctable := (list (cname * (ann * cname * a_flds * a_flds *
    a_flds * a_flds * a_mths))).

Parameter aCT : a_ctable.
```

Listing 4.4: Formalization of the Class Table

First, in environments mainly used for method parameters, we can annotate every parameter. The variable names are mapped to an annotation and a type. Second, field names are mapped not only to a type, they are mapped to an annotation and a type. Third, method names are mapped to an annotation and a method declaration. The declaration consists of a return type, a parameter list and a term.

The class table maps class names to a class declaration. The class declaration is a six-tuple `(a,D,gs,ts,fs,ms)`, where `a` is the annotation of the class, `D` is the super class, `gs` are the constructor parameters, `hs` are the super call parameters, `ts` are the field assignments at the constructor, `fs` are the fields, and `ms` are the methods. To be able to save all possible annotations, we added `a`, `gs`, `hs`, and `ts` at the CFJ class table. Note, that de Fraine do not model constructor parameters, super call parameters and assignments as they are uniquely described by the fields of each class.

The FJ formalization by de Fraine makes advantage of a fixed class table. It means, that we assume to have one class table for FJ that remains unchanged while we check certain properties. For instance, we check that each class is well-typed, but therefore we rely on the whole class table. We check classes, methods, and fields iteratively. For instance, a list of classes is well-typed if the first class is well-typed and the list without the first class is well-typed—or if the list is empty. We remove classes from this local list, while checks, e.g., whether the super class exists, are made on the fixed class table. This way, we can check class tables where two classes reference each other, what is explicitly allowed in FJ.

Since we want to reuse the FJ formalization, we also need to assume that the FJ class table is fixed. Additionally, we assume that the CFJ class table is fixed, because our formalization should be as close as possible to the FJ formalization. Hence, whenever we want to prove properties between the CFJ and the FJ class table, we also need to assume a fixed configuration that is used to generate the FJ class table from the CFJ class table. We give more details on the variant generation in Section 4.3. As in the FJ formalization, we assume the CFJ class table `aCT` as a global parameter that we do not have to pass it to every definition (see Listing 4.4).

Annotation Lookup

In our typing rules we need to lookup annotations at the class table. For instance, for typing a class we need the annotation of the super class. The function given in Listing 4.5 returns the annotation of a class and expects the class name as a parameter. We use a similar naming to that of the informal description that the reachability checks look similar and the correct realization can easily be verified by the reader.

```
Definition AT (C:cname) : ann :=
    match (get C aCT) with
    | None ⇒ False
    | Some (a, _, _, _, _, _, _) ⇒ a
    end.
```

Listing 4.5: Formalization of Annotation Lookup

To lookup annotations of other code elements than classes, we need different functions, because they appear at other locations at the class table and we need more parameters to identify the code fragments. In Table 4.1, we summarize the annotation lookup functions and their meaning. We omit the definitions in Coq here, since they are highly repetitive. They can be found in Appendix B.2.

4.2.2 Subtyping and Auxiliary Rules

We present our formalization of the CFJ subtyping and auxiliary rules. Subtyping and field lookup is identically to that of FJ except that the lookup is defined on the CFJ class table. For method lookup we additionally needed to add reachability checks and a new sub case. The formalization of CFJ method overriding also includes some additionally reachability checks.

Function	Annotation Lookup for
`AT`	Classes
`ATp`	Constructor Parameters
`ATs`	Super Call Parameters
`ATa`	Assignments `this.f:=f;`
`ATf`	Fields
`ATm`	Methods

Table 4.1: Annotation Lookup at the Class Table

Subtyping

The subtyping rules of CFJ and FJ are identical. Nonetheless, we need a new definition in Coq that lookups the super type relation at the CFJ class table. In Listing 4.6, we present the formalization consisting of two definitions as for FJ. The definition `a_extends C D` is valid, if `C` is a direct subclass of `D`. The reflexive and transitive closure of `a_extends` is determined by `a_sub`.

```
Definition a_extends (C D : cname) : Prop :=
  exists gs, exists hs, exists ts, exists fs, exists ms,
    binds C (AT C,D,gs,hs,ts,fs,ms) aCT.
Inductive a_sub : typ → typ → Prop :=
| a_sub_refl : forall t, a_sub t t
| a_sub_trans : forall t1 t2 t3,
    a_sub t1 t2 → a_sub t2 t3 → a_sub t1 t3
| a_sub_extends : forall C D, a_extends C D → a_sub C D.
```

Listing 4.6: Formalization of Subtyping

As in the FJ formalization, we use implication whenever possible, as it is common in Coq and the proofs are easier. One needs to know that the implication in Coq is right-associative [BC04], meaning that `A → B → C` is equivalent to `A → (B → C)`. We know that `B → C` is true, whenever `A` ist true. Assuming that `A` is true, we get that `C` is true, whenever `B` ist true. Therefore, `A → (B → C)` is equivalent to `(A ∧ B) → C`, what corresponds to the semantics of inference rules. This also holds for more than two premises.

Field Lookup

The FJ field lookup determines the fields of a class inductively. The class `Object` has no fields. Every other class has the own fields concatenated with the fields of the super class. The CFJ field lookup works absolutely identical, but on the CFJ class table (see Listing 4.7). The definition `a_field` is used to check if a class contains a field mapped to a particular type and annotation.

```
Inductive a_fields : cname → a_flds → Prop :=
| a_fields_obj : a_fields Object nil
| a_fields_other : forall C D gs hs ts fs fs' ms,
    binds C (AT C,D,gs,hs,ts,fs,ms) aCT →
    a_fields D fs' →
    a_fields C (fs'++fs).
Definition a_field (C:cname) (f:fname) (a:ann) (t:typ) : Prop :=
    exists2 fs, a_fields C fs & binds f (a,t) fs.
```

Listing 4.7: Formalization of Field Lookup

Method Lookup

Method lookup needs to be redefined, because *mtype* in Figure 3.12 on Page 36 has a new case, that returns the method declaration of the super class if the method is not present in all variants, where it is used. In Listing 4.8, we realized this new case.

```
Inductive a_method : cname → mname → ann → typ ∗ a_env ∗ a_exp →
  Prop :=
| a_method_this : forall (C D:cname) (fs gs hs ts:a_flds) (m:mname)
    (ms:a_mths) (mdecl:typ ∗ a_env ∗ a_exp) (a:ann),
    binds C (AT C,D,gs,hs,ts,fs,ms) aCT →
    binds m (ATm C m,mdecl) ms →
    (a → ATm C m) →
    a_method C m a mdecl
| a_method_notthis : forall (C D:cname) (fs gs hs ts:a_flds) (m:mname)
    (ms:a_mths) (mdecl mdecl':typ ∗ a_env ∗ a_exp) (a:ann),
    binds C (AT C,D,gs,hs,ts,fs,ms) aCT →
    binds m (ATm C m,mdecl') ms →
    ~(a → ATm C m) →
    a_method D m a mdecl →
    a_method C m a mdecl
| a_method_super : forall (C D:cname) (fs gs hs ts:a_flds) (m:mname)
    (ms:a_mths) (mdecl:typ ∗ a_env ∗ a_exp) (a:ann),
    binds C (AT C,D,gs,hs,ts,fs,ms) aCT →
    no_binds m ms →
    a_method D m a mdecl →
    a_method C m a mdecl.
```

Listing 4.8: Formalization of Method Lookup

Furthermore, the formalization of reachability checks is straightforward, as annotations are modeled of type **`Prop`**. Implication can be implemented like it is used for the type system in Section 3.3.4. Negation of propositional formulas is denoted by ∼ and we use the annotation lookup function for methods as described above.

Valid Method Overriding

Valid method overriding is realized as follows. For every method declaration that exists with the given method name at any super class, we force that the return type is equal

```
Definition a_can_override (C D:cname) (m:mname) (C0:typ) (E:a_env)
  (a:ann) : Prop :=
    forall C0' E' t,
    a_method D m a (C0',E',t) →
    C0 = C0' ∧ a_can_override_check a m C E D E'.
```

Listing 4.9: Formalization of Valid Method Overriding

to that of to the given method (see Listing 4.9). Additionally, some reachability checks need to be satisfied.

Listing 4.10 shows the reachability checks for method overriding. It makes sure that both parameter lists have identical types and that the annotations are equivalent whenever the method is present.

```
Inductive a_can_override_check : ann → mname → cname → a_env →
  cname → a_env → Prop :=
| a_coc_nil : forall (a:ann) (m:mname) (C D:cname),
    a_can_override_check a m C nil D nil
| a_coc_cons : forall (a AT_Cixi AT_Diyi:ann) (m:mname) (C D:cname)
    (Cxs Dxs:a_env) (xi yi:var) (Ci:cname),
    a_can_override_check a m C Cxs D Dxs →
    (a → (AT_Cixi ↔ AT_Diyi)) →
    a_can_override_check a m C ((xi,(AT_Cixi,Ci))::Cxs)
                              D ((yi,(AT_Diyi,Ci))::Dxs).
```

Listing 4.10: Reachability Checks for Valid Method Overriding

4.2.3 Typing Rules

In this section, we present the formalization of the CFJ typing rules. The new definitions are enriched with respective reachability checks. We describe term typing, method typing, class typing, and finally product line typing.

Term Typing

The formalization of term typing is different than the definitions we already presented. First, it is defined inductively, because a term might contain further terms. Second, we need to type terms and list of terms. Third, for lists we usually need wide typing, i.e., a term has a type that is a sub type of a particular type. For instance, we need wide typing to classify well-typed methods.

Listing 4.11 shows the formalization and again it is very close to that of term typing for FJ by de Fraine. Term typing is realized using mutually inductive definitions. Mutually inductive means that the definitions for `a_typing`, `a_wide_typing` and `a_wide_typings` can reference each other. This is only possible due to a special language construct named **`with`**. Otherwise, Coq would bring an error if we try to use a definition that is not specified before.

```
Inductive a_typing : ann → a_env → a_exp → typ → Prop :=
| a_t_var : forall (a a’:ann) (E:a_env) (x:var) (C:typ),
    ok E →
    binds x (a’,C) E →
    (a → a’) →
    (a’ → AT C) →
    a_typing a E (ae_var x) C
| a_t_field : forall (a:ann) (E:a_env) (t0:a_exp) (C0 Ci:typ)
    (fi:fname),
    a_typing a E t0 C0 →
    a_field C0 fi (ATf C0 fi) Ci →
    (a → ATf C0 fi) →
    (ATf C0 fi → AT Ci) →
    a_typing a E (ae_field t0 fi) Ci
| a_t_meth : forall (a:ann) (E:a_env) (Dys:a_env) (t0 t:a_exp)
    (C0 C:typ) (m:mname) (ts:list (ann ∗ a_exp)),
    a_typing a E t0 C0 →
    a_method C0 m a (C,Dys,t) →
    a_wide_typings a E ts (imgs Dys) →
    (a → AT C) →
    a_typing a E (ae_meth t0 m ts) C
| a_t_new : forall (a:ann) (E:a_env) (Dfs:a_flds) (C:typ)
    (ts:list (ann ∗ a_exp)),
    a_fields C Dfs →
    a_wide_typings a E ts (imgs Dfs) →
    (a → AT C) →
    a_typing a E (ae_new C ts) C

with a_wide_typing : ann → a_env → a_exp → typ → Prop :=
| a_wt_sub : forall (a:ann) (E:a_env) (e:a_exp) (t t’:typ),
    a_typing a E e t → sub t t’ → a_wide_typing a E e t’

with a_wide_typings : ann → a_env → list (ann ∗ a_exp) →
  list (ann ∗ typ) → Prop :=
| a_wts_nil : forall (a:ann) (E:a_env),
    ok E →
    a_wide_typings a E nil nil
| a_wts_cons : forall (a AT_t AT_Dy:ann) (E:a_env)
    (Ds:list (ann ∗ typ)) (ts:list (ann ∗ a_exp)) (t:a_exp) (C:typ),
    a_wide_typings a E ts Ds →
    a_wide_typing AT_t E t C →
    (a → (AT_t ↔ AT_Dy)) →
    (AT_t → a) →
    a_wide_typings a E ((AT_t,t)::ts) ((AT_Dy,C)::Ds).
```

Listing 4.11: Formalization of Term Typing

The inductive definition `a_typing` is used to describe well-typed terms. We have the same cases as in Figure 3.13 on Page 37, except for casting, since we decided to formalize cast-free CFJ. The reachability checks are straightforward to implement. We strongly highlighted three additional checks in typing of variables, field accesses, and method invocations, which we explain in Section 5.2.4. For short, they are used to reduce the complexity of the type soundness proof.

In the cases `a_t_meth` and `a_t_new`, the typing of parameters is realized using `a_wide_typings`. Fortunately, the parameter checks for method invocation and object creation are identical, so that we only need to implement them once at `a_wts_cons`. The definition `a_wide_typing` is used to type every parameter and it takes advantage of `sub` to identify valid sub classes, as it is defined for FJ.

Method Typing

Method typing for CFJ is straightforward given the formalization of FJ. Listing 4.12 presents our formalized definition. The let-in construct is used to define a local variable `a`, representing the annotation of the current method.

```
Definition a_ok_meth (C D:cname) (m:mname) (C0:typ) (Cxs:a_env)
   (t0:a_exp) : Prop :=
     let a := ATm C m in
     let E := (this,(AT C,C))::Cxs in
     (a → AT C0) ∧
     a_ok_meth_check a Cxs ∧
     a_can_override C D m C0 Cxs a ∧
     a_wide_typing a E t0 C0.

Definition a_ok_meth' (C D: cname) (m : mname)
   (v:ann ∗ (typ ∗ a_env ∗ a_exp)) : Prop :=
     match v with (_,(C0,E,t)) ⇒ a_ok_meth C D m C0 E t end.
```

Listing 4.12: Formalization of Method Typing

A method is well-typed, if the following four conditions are fulfilled [Pie02].

1. The return type `C0` is present, whenever the method is present.
2. The reachability checks for the method parameters are satisfied.
3. The method overrides potentially super methods properly.
4. The term is well-typed according to the return type and under the annotation of the method.

The variable `Cxs` represents the method parameters. In FJ, it is a list of pairs mapping a variable name to a class name. Since we store annotations directly at the class table, our parameter list is a list of pairs mapping variable names to a pair of an annotation and a class name. Accordingly, the variable name `this` is mapped to the annotation of the class and the class name.

Definitions ending with `'` are only needed for technical reasons. Whenever a function need to be applied to all elements of a list, de Fraine uses `forall_env`. It gets the function and a list as parameters. It yields `True` if the function evaluates to `True` for all elements in the list. We reuse `forall_env`.

In Listing 4.13, the reachability checks for the method parameters are formalized. As for class typing, they are defined inductively. The realization of the reachability checks is straightforward.

```
Inductive a_ok_meth_check : ann → a_env → Prop :=
| a_omc_nil : forall (a:ann),
    a_ok_meth_check a nil
| a_omc_cons : forall (a AT_Cixi:ann) (Cxs:a_env) (Ci:typ) (xi:var),
    a_ok_meth_check a Cxs →
    (AT_Cixi → AT Ci) →
    (AT_Cixi → a) →
    a_ok_meth_check a ((xi,(AT_Cixi,Ci))::Cxs).
```

Listing 4.13: Reachability Checks for Method Typing

Class Typing

In Listing 4.14, we present how we formalized well-typed CFJ classes in Coq.

```
Definition a_ok_class (C:cname) (D:cname) (fs:a_flds) (ms:a_mths) :
  Prop := let a := AT C in
    (exists gs:a_flds,
      a_fields D gs ∧ a_ok_class_check a C D fs gs ms ∧
            ok (gs ++ fs) ) ∧
    (a → AT D) ∧ ok ms ∧ forall_env (a_ok_meth' C D) ms.

Definition a_ok_class' (C:cname) (v:ann * cname * a_flds * a_flds *
  a_flds * a_flds * a_mths) : Prop :=
    match v with (_,D,_,_,_,fs,ms) ⇒ a_ok_class C D fs ms end.
```

Listing 4.14: Formalization of Class Typing

A class is well-typed if five conditions are fulfilled. We give informal descriptions while using the same order as they appear in the formalization.

1. The class check is fulfilled.
2. The field names of this and all super classes are distinct.
3. The annotation of the class implies the annotation of the superclass.
4. All methods have distinct names.

5. All methods are well-typed.

The first and the third condition are added to realize the reachability checks. The other conditions are identical to that of FJ, while we need to use our adapted method typing.

The reachability checks we realize at the class check iterate on the own fields, the overall fields of the super class, and the methods of a particular class. Listing 4.15 shows that we defined the check inductively. If the three lists of interest are empty (denoted by `nil`), the check is fulfilled. If the check is fulfilled for some particular lists and certain conditions are satisfied, then the check is also fulfilled for the lists, where a field or method is added.

```
Inductive a_ok_class_check :
  ann → cname → cname → a_flds → a_flds → a_mths → Prop :=
| a_occ_nil : forall (a:ann) (C:cname) (D:cname),
    a_ok_class_check a C D nil nil nil
| a_occ_cons_flds1 : forall (a:ann) (C:cname) (D:cname) (fs:a_flds)
    (fi:fname) (Ci:typ),
    a_ok_class_check a C D fs nil nil →
    (ATf C fi → a ∧ AT Ci) →
    ((ATf C fi ↔ ATa C fi) ∧ (ATa C fi ↔ ATp C fi)) →
    a_ok_class_check a C D ((fi,(ATf C fi,Ci))::fs) nil nil
| a_occ_cons_flds2 : forall (a AT_Digi:ann) (C:cname) (D:cname)
    (fs gs:a_flds) (gi:fname) (Di:typ),
    a_ok_class_check a C D fs gs nil →
    (a → (ATp C gi ↔ AT_Digi)) →
    (ATp C gi ↔ ATs C gi) →
    (ATp C gi → a) →
    a_ok_class_check a C D fs ((gi,(AT_Digi,Di))::gs) nil
| a_occ_cons_mths : forall (a:ann) (C: cname) (D: cname)
    (fs gs:a_flds) (ms:a_mths) (m:mname) (m':typ * a_env * a_exp),
    a_ok_class_check a C D fs gs ms →
    (ATm C m → a) →
    a_ok_class_check a C D fs gs ((m,(ATm C m,m'))::ms).
```

Listing 4.15: Reachability Checks for Class Typing

Using the lookup functions for annotations, the reachability checks are straightforward to implement. Due to our similar naming, they even appear very close to the checks we presented in Figure 3.13 on Page 37.

Product Line Typing

Before we formalize product line typing, we give a definition that identifies well-typed FJ programs (see Listing 4.16). This definition was assumed implicitly for the FJ type system formalized by de Fraine. A FJ program is well-typed if the class table is well-typed and the start term is well-typed. Since the type of the start term is not part of a FJ program, we state that there exists such a type, for that the start term is well-typed. The start term is typed under the empty context, because there are no surrounding variables.

```
Definition FJ_program (t:exp) : Prop :=
    ok_ctable CT ∧ exists C:cname, typing nil t C.
```

Listing 4.16: Formalization of Well-Typed FJ Programs

A CFJ product line is well-typed, if both, the class table and the start term are well-typed pursuant to the CFJ type system (see Listing 4.17). The definition `a_typing` needs the current annotation as an additional parameter, which is in our case the empty annotation represented by `True`.

```
Definition CFJ_product_line (t:a_exp) : Prop :=
    a_ok_ctable aCT ∧ exists C:cname, a_typing True nil t C.
```

Listing 4.17: Formalization of Well-Typed CFJ Product Lines

A CFJ class table is well-typed, if all classes have unique names and all classes are well-typed (see Listing 4.18). This definition is identical to that of FJ, except for the new definition that is used to identify well-typed classes. Fortunately, we can reuse `ok` as is, since the CFJ class tables also map class names to their declaration and the definition only uses the class names to make sure that no two classes with the same name occur.

```
Definition a_ok_ctable (ct:a_ctable) :=
    ok ct ∧ forall_env a_ok_class' ct.
```

Listing 4.18: Formalization of Well-Typed CFJ Class Tables

4.3 Variant Generation

This section presents our formalization of the variant generation. It maps CFJ class tables and the CFJ start term to a FJ class table and a FJ start term. Existentially for the variant generation is a valid configuration describing which annotated code to remove.

Valid configurations are specified at the feature model, so that we start with the formalization of the feature model. Afterwards, we formalize the variant generation for parameter lists, terms, methods, and whole class tables. Finally, we give an overview on the presented definitions.

Feature Model and Valid Configuration

Before we can formalize the variant generation, we need a representation of the feature model in Coq. The reason is, that we need to distinguish between valid and invalid configurations. We present three ideas to formalize the feature model that arise from our introduction on feature models in Section 2.1.1.

1. Enumeration. We enumerate all valid variants, i.e., we define a set in Coq including all valid configurations. A function could lookup, whether a given configuration is in the specified set or not. The configurations itself could be represented as sets containing all selected features.

2. Feature diagram. We emulate the structure of a feature diagram and possible cross-tree constraint in Coq. This includes all group types—*And*-, *Or*-, and *Alternative*-groups—and arbitrary propositional formulas for cross-tree constraints. Additionally, a function is needed to determine if a given configuration is valid according to the feature model, which is not trivial.

3. Propositional formula. The feature model is converted into a propositional formula by human or a feature modeling tool. Then, this propositional formula is formalized in Coq. As for annotations, we can use the type **`Prop`** and logical connector given for this type. A configuration is an allocation of `True` or `False` to each variable representing a feature. A configuration is valid, if the propositional formula representing the feature model evaluates to `True` for this allocation.

The first idea is unpractical for large feature models which can have millions of valid configurations. The second idea is more complicated than the third one, as the cross-tree constraints may be arbitrary propositional formulas and we would need to realize the third idea as well as formalizing the feature model. Therefore, we decided to formalize the feature model based on propositional formulas.

Given a valid configuration a code fragment is removed, if and only if the annotation evaluates to false under the configuration. To apply variant generation to the class table, we need an inductively defined generation function that is defined on classes, methods, fields, terms and parameters, i.e., we need multiple functions that have a similar structure as our definitions for well-typed CFJ class tables.

Parameter Lists

Whenever a FJ method is generated, we might need to remove certain parameters whose annotations evaluate to `False`. We defined `variant_env` for this issue and it gets a CFJ parameter list and a FJ parameter list and returns `True` if both describe valid variant generation according to an assumed configuration (see Listing 4.19).

In fact, `variant_env` is a relation and two parameter lists are in that relation if both are empty. According to `ve_cons_y`, given two lists that are in relation and an annotation `a` that evaluates to `True`, we know that adding this variable and type to both lists is in that relation too. If `a` does evaluate to `False`, i.e., ∼`a` evaluates to `True`, then we do not add this parameter to the FJ parameter list.

The reader might wonder why we do not formalize variant generation as a recursive function, e.g., using **`Fixpoint`**. The reason is, that the result depends on the annotation, i.e., whether it evaluates to `True` or `False`. Recursive functions that rely on values of propositional formulas are not allowed in Coq.

```
Inductive variant_env : a_env → env → Prop :=
| ve_nil :
    variant_env nil nil
| ve_cons_y : forall (E:a_env) (E':env) (x:var) (a:ann) (C:typ),
    variant_env E E' →
    a →
    variant_env ((x,(a,C))::E) ((x,C)::E')
| ve_cons_n : forall (E:a_env) (E':env) (x:var) (a:ann) (C:typ),
    variant_env E E' →
    ~a →
    variant_env ((x,(a,C))::E) E'.
```

Listing 4.19: Formalization of Parameter List Generation

Terms

FJ terms need to be generated, since they occur in methods. The outermost term of a method cannot be annotated, but we might need to remove certain terms given as parameters. Parameters occur in method invocation terms and object creation terms. Listing 4.20 presents our formalization of term generation.

The mapping of variables and fields is straightforward, since they cannot be annotated. However, for fields we need to force that the term, on which the field is applied, is generated correctly. The same is true for methods. Additionally, for method invocations and object creations we have to generate the parameter term lists.

The latter is done by a mutual definition `variant_terms`. Contrary to `variant_term`, it directly removes terms occurring in a list based on their annotation. It is very similar to `variant_env`. Note, the recursive call to `variant_term` which is needed as, e.g., a parameter list might contain a further parameter list.

Methods

Our formalization of FJ method generation is based on the three previous definitions. Listing 4.21 presents the definition `variant_methods`, which gets a list of CFJ methods and a list of FJ methods as parameters. Additionally, it requires the name of the class in which the methods are defined.

The class name is needed for a special reason; we use it as a parameter for the function `ATm`, which returns the annotation of the method. We do not need to lookup the annotation at the class table at this point, since we get the method declaration including the annotations as a parameter. The usage of the lookup function has technical reasons: it simplifies some of the proofs we present in the next chapter.

Beside this consideration, the formalization of `variant_methods` is straightforward using induction on lists. Again, we have three cases: (a) both lists are empty, (b) the annotation of a method evaluates to `True`, or (c) to `False`. If it evaluates to `True`, we add this method to the list of FJ methods with the generated term and the generated parameter list.

```
Inductive variant_term : a_exp → exp → Prop :=
| vt_var : forall (x:var),
    variant_term (ae_var x) (e_var x)
| vt_field : forall (t:a_exp) (t':exp) (f:fname),
    variant_term t t' →
    variant_term (ae_field t f) (e_field t' f)
| vt_meth : forall (t:a_exp) (t':exp) (m:mname) (ts:list (ann∗a_exp))
    (ts':list exp),
    variant_term t t' →
    variant_terms ts ts' →
    variant_term (ae_meth t m ts) (e_meth t' m ts')
| vt_new : forall (ts:list (ann∗a_exp)) (ts':list exp) (C:typ),
    variant_terms ts ts' →
    variant_term (ae_new C ts) (e_new C ts')

with variant_terms : list (ann∗a_exp) → list exp → Prop :=
| vts_nil :
    variant_terms nil nil
| vts_cons_y : forall (ts:list(ann∗a_exp)) (ts':list exp) (a:ann)
    (t:a_exp) (t':exp),
    variant_terms ts ts' →
    a →
    variant_term t t' →
    variant_terms ((a,t)::ts) (t'::ts')
| vts_cons_n : forall (ts:list(ann∗a_exp)) (ts':list exp) (a:ann)
    (t:a_exp),
    variant_terms ts ts' →
    ∼a →
    variant_terms ((a,t)::ts) ts'.
```

Listing 4.20: Formalization of Term Generation

```
Inductive variant_methods : cname → a_mths → mths → Prop :=
| vms_nil : forall (C:cname),
    variant_methods C nil nil
| vms_cons_y : forall (C:cname) (ms:a_mths) (ms':mths) (m:mname)
    (C0:typ) (Cxs:a_env) (Cxs':env) (t:a_exp) (t':exp),
    variant_methods C ms ms' →
    variant_term t t' →
    variant_env Cxs Cxs' →
    ATm C m →
    variant_methods C ((m,(ATm C m,(C0,Cxs,t)))::ms)
                      ((m,(C0,Cxs',t'))::ms')
| vms_cons_n : forall (C:cname) (ms:a_mths) (ms':mths) (m:mname)
    (C0:typ) (Cxs:a_env) (t:a_exp),
    variant_methods C ms ms' →
    ∼(ATm C m) →
    variant_methods C ((m,(ATm C m,(C0,Cxs,t)))::ms) ms'.
```

Listing 4.21: Formalization of Method Generation

Class Table

Finally, we present how a FJ class table is generated from a CFJ class table. The definition we present in Listing 4.22 is very similar to that for methods.

```
Inductive variant_ct : a_ctable → ctable → Prop :=
| vct_nil :
    variant_ct nil nil
| vct_cons_y : forall (ct:a_ctable) (ct':ctable) (C D:cname)
    (fs gs hs ts:a_flds) (fs':flds) (ms:a_mths)(ms':mths),
    variant_ct ct ct' →
    variant_env fs fs' →
    variant_methods C ms ms' →
    AT C →
    variant_ct ((C,(AT C,D,gs,hs,ts,fs,ms))::ct)
               ((C,(D,fs',ms'))::ct')
| vct_cons_n : forall (ct:a_ctable) (ct':ctable) (C D:cname)
    (fs gs hs ts:a_flds) (fs':flds) (ms:a_mths) (ms':mths),
    variant_ct ct ct' →
    ~(AT C) →
    variant_ct ((C,(AT C,D,gs,hs,ts,fs,ms))::ct) ct'.
```

Listing 4.22: Formalization of Class Table Generation

The cases `vct_nil` and `vct_cons_n` are straightforward to implement using our previous considerations. The interesting case is `vct_cons_y`, which generates a FJ class from a CFJ class. The superclass remains the same and we only have to generate fields and methods.

Methods are generated using `variant_methods` we just defined. Fortunately, the fields of the FJ class can be generated using the parameter list generation `variant_env`. The reason is, that in the FJ formalization and also in our CFJ formalization parameter lists and fields are both lists of pairs mapping a variable or field name to a type and in case of CFJ to a pair of annotation and type. Variable and fields names are convertible, i.e., `fname` and `var` are both notations for the same type `atom` that is used for all names at the FJ and CFJ class tables.

4.4 Summary

We presented several definitions that identify well-typed CFJ product lines. Figure 4.1 summarizes the definitions and shows how they are related. An arrow that begins in at a certain definition, indicates the definitions that is uses.

Additionally, `CFJ_product_line` uses `a_ok_ctable` and `a_typing`. Again, we see the special position of the term typing that we realized using mutual induction. Variant generation for CFJ is formalized using five definitions. The dependencies between these definitions are visualized in Figure 4.2. Again, an arrow starting at a definition means, that this definition uses another definition.

We appear some similarities to typing for CFJ. The overall structure beginning at the class table and iterating through fields, methods and finally terms is similar. For terms

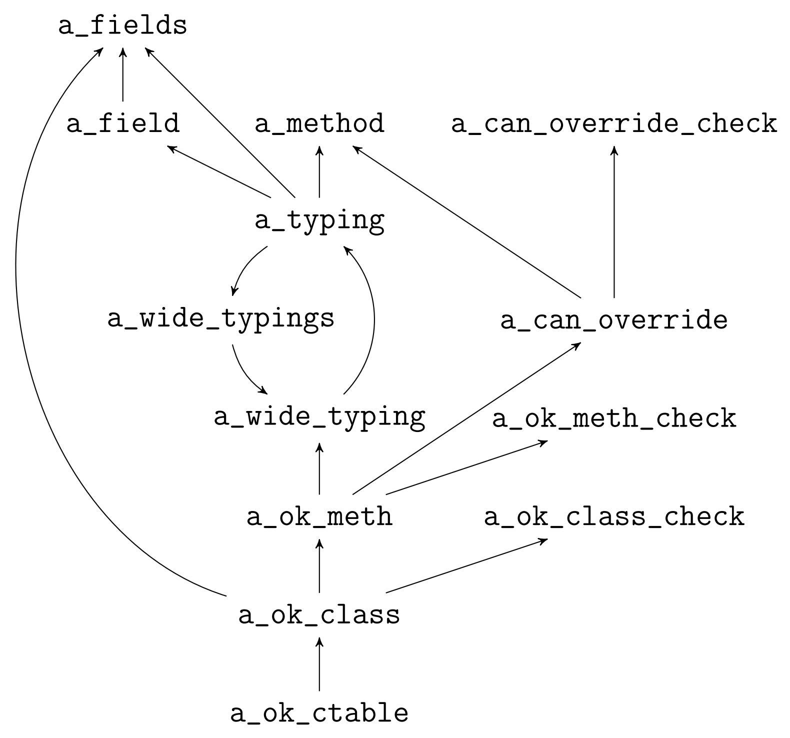


Figure 4.1: Overview on Definitions for CFJ

we also have mutual definitions, in this case only two ones, since wide typing is not needed for the variant generation.

Given the formalization of the type system and the variant generation, we can prove type soundness of CFJ in the following chapter.

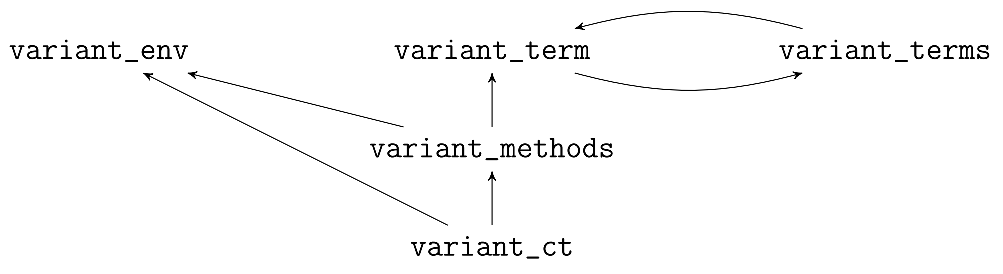


Figure 4.2: Overview on the Variant Generation for CFJ

5. Type Soundness of Colored Featherweight Java

This chapter presents our machine-checked type soundness proof for CFJ. A type soundness theorem states that a given type system is correct, e.g., there is no dangling method reference in any FJ program variant. The proof is based on our formalization of the CFJ type system and variant generation proposed in the previous chapter. Our contribution is the first machine-checked proof of type soundness for CFJ. Recently, type soundness was proven by Kästner et al. in informal math [KAS, KA08], which is the only type soundness proof for CFJ so far.

Our proof of the type soundness theorem is based on 29 further theorems, lemmas, and facts for that we also provide formal proofs checked by the proof assistant Coq. We do not want to examine every proof in detail. The reason is, that the proofs can be verified by Coq and the main attention should be on the definitions and the theorems, since Coq can only verify proofs, but not that our definitions and theorems are reasonable. Instead, we give an overview on the theorems and discuss problems solved in proof writing.

First, we formalize the type soundness theorem for CFJ in Section 5.1. Additionally, we describe all theorems, lemmas, and facts used. Second, Section 5.2 describes our proof strategy, discusses problems in proving and presents our solutions used to complete all proofs. Finally, we summarize this chapter in Section 5.3.

5.1 Type Soundness Theorem

Section 5.1.1 describes what type soundness means for CFJ and proposes our formalization of the theorem. In Section 5.1.2, we give an overview on the facts, lemmas, and theorems used to prove the type soundness theorem.

5.1.1 Formalization of the Theorem

Type soundness of FJ is proven using the progress theorem and the preservation theorem [IPW99]. The formalization by de Fraine already contains the proofs of both. We introduced progress and preservation in Section 2.2.2. Progress means that a well-typed term is either a value or it can take a step of evaluation. Preservation states that if a well-typed term takes a step of evaluation, then the resulting term is well-typed as well.

Type soundness of CFJ cannot be proven using progress and preservation, because CFJ product lines are never directly evaluated. Kästner and Apel argue that the crucial property for CFJ is that variant generation preserves typing [KA08]. That is, *every FJ program generated from a well-typed CFJ product line using a valid configuration is well-typed according to the FJ typing rules.* Kästner and Apel provided a proof sketch for the generation preserves typing theorem in informal math [KA08]. In recent work, Kästner et al. provided a full proof in informal math [KAS].

The theorem concludes that every generated FJ program is well-typed, given (a) the CFJ product line is well-typed, (b) the program is generated from the product line and (c) using a valid configuration. We present our formalization of type soundness for CFJ in Listing 5.1, which is straightforward given the definitions of well-typed CFJ product lines and the variant generation defined in Chapter 4.

```
1 Theorem generation_preserves_typing : forall (t:a_exp) (t':exp),
2     CFJ_product_line t →
3     variant_ct aCT CT →
4     variant_term t t' →
5     FM →
6     FJ_program t'.
```

Listing 5.1: The Type Soundness Theorem—Generation Preserves Typing

The first premise states that the CFJ product line is well-typed (see Line 2). We defined `CFJ_product_line` in Section 4.2. The product lines consists of the CFJ class table `aCT` and the start term `t`. The second and the third premise describes the connection between the CFJ product line and the FJ program, i.e., the variant generation (see Line 3 and Line 4). The FJ class table `aCT` is generated from the CFJ class table and the FJ start term `t'` is generated from the CFJ start term `t`.

The conclusion is that the FJ class table `CT` and the start term `t'` together build up a well-typed FJ program (see Line 6). We use our adapted FJ formalization at this point, because it is consistent with the FJ type system presented in Chapter 3. Note, that our adoptions are so simple that we reuse the most parts of de Fraine's FJ formalization as-is.

The last premise `FM` corresponds to the requirement that the configuration is valid (see Line 5). It is a propositional formula representing the feature model, which is `True` for all configurations that are valid. This premise is not needed for the proof itself, but we need it to be able to build CFJ product lines that are classified as well-typed. The problem is, that for invalid configurations an empty FJ class table is generated.

Remember, that we decided to insert the feature model `FM` into every annotation, by what all annotations evaluate to `False` and so all classes are removed. But this implies, that whenever we use a feature model, where such an invalid configuration exists, we can only use `new Object()` as a start term, since other start terms would need existing classes at the class table.

5.1.2 Splitting the Theorem

The proof of the type soundness theorem is not trivial. In fact, it is so complex that we proved it based on two other theorems stating generation preserves typing on terms and class tables (`gpt_typing` and `gpt_ok_ctable`), i.e., every FJ term or class table generated from a well-typed CFJ term or class table is well-typed as well.

In Figure 5.1, we visualize the dependencies between the theorems, i.e., which theorems were used to prove a certain theorem. The definitions in Coq are omitted here, since many of them are very similar. All definitions can be found in Appendix B.3, while the number in the right upper corner of each theorem is the line number.

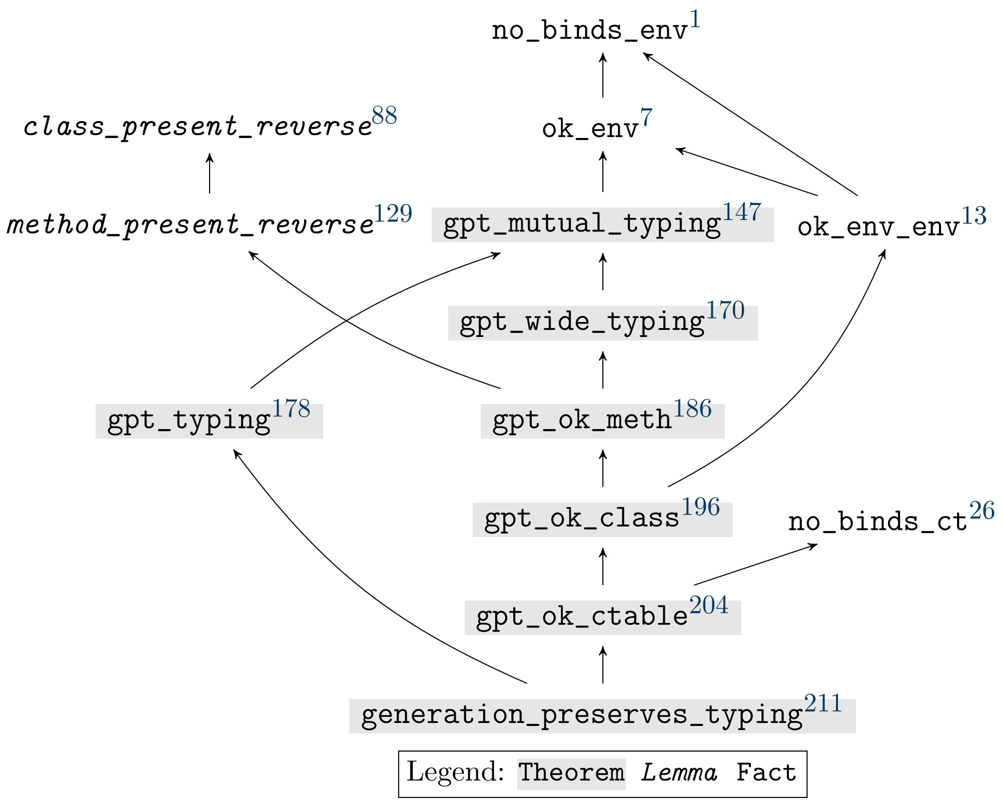


Figure 5.1: Theorems to Prove Generation Preserves Typing

We distinguish between theorems (highlighted), lemmas (italic), and facts. The theorems state generation preserves typing on different levels, e.g., classes, methods, or terms. Facts can either be proven without splitting or just rely on other facts. Lemmas indicate properties concerning reachability checks.

Generation preserves typing for class tables is proven using generation preserves typing on classes (`gpt_ok_class`). In turn, the latter is shown with generation preserves typing on methods (`gpt_ok_meth`). This breakdown is basically caused by the structure of CFJ product lines and FJ programs. Both consist of a class table and a start term, while the class table contains classes containing methods.

We need two theorems stating generation preserves typing for terms. While for the start term the type is established (`gpt_typing`), for the term of a method, we only know it has to be well-typed with a sub type of the method's return type (`gpt_wide_typing`). In Section 5.2.3, we explain the need for a further theorem to prove these two theorems concerning term typing (`gpt_mutual_typing`).

We split theorems into smaller ones, largely for two reasons, because the proof is too complex to prove at once or because the proof is needed at multiple positions in one or more proofs. For clarity, we avoided splitting theorems if not necessary. Still, all dependencies do not fit into one single figure, so that we present further dependencies in Figure 5.2.

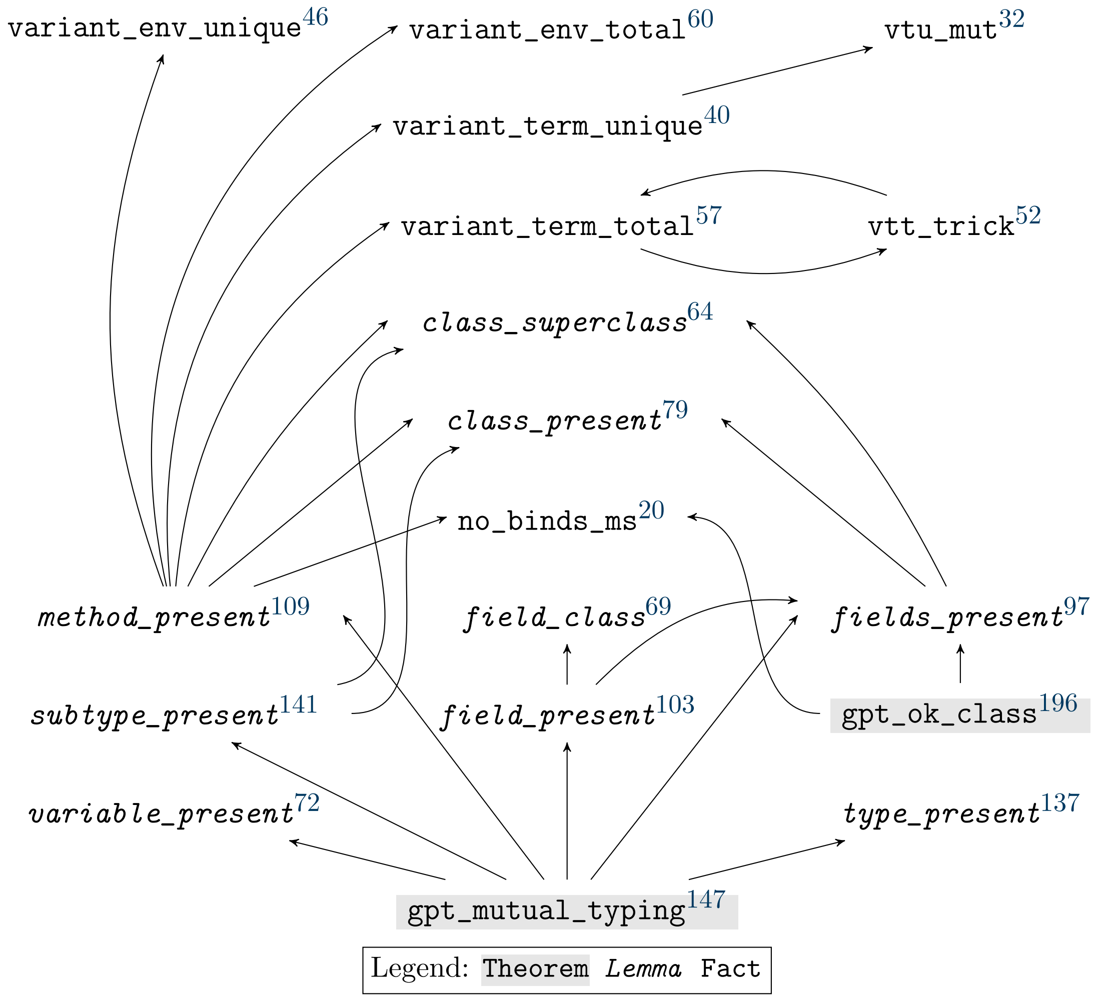


Figure 5.2: Further Theorems to Prove Generation Preserves Typing

We classify facts into three groups. First, if a variable, method, or class is not contained in a CFJ list, then it is not in the generated FJ list (`no_binds_env`, `no_binds_ms`, and `no_binds_ct`). Second, if a list contains no duplicate names in CFJ, then the generated FJ list does not either (`ok_env` and `ok_env_env`). Third, variant generation for terms and parameter lists is a total function, i.e., for every CFJ code fragment there is an unique FJ code fragment generated (remaining facts).

We also have three groups of lemmas. First, if a given annotation evaluates to `True`, then also another annotation evaluates to `True` as well. We proved the two lemmas that whenever a field is present, the class is present too (`field_class`) and that whenever a class is present, its super class is as well (`class_superclass`).

Second, whenever an annotation for a particular code fragment evaluates to `True`, then the code fragment is at the FJ class table, while we need this property for variables, methods, and fields (`variable_present`, `method_present`, `field_present`, and `fields_present`). All these lemmas are used for term typing. Remember, a term might be a parameter, a field access, a method invocation or an object creation. In all these cases we can ensure that the annotation evaluates to `True` and need to prove that a particular code fragment is at the FJ class table.

Third, a group of lemmas states the reverse of the second group. Whenever a code fragment occurs at the FJ class table, then we know that its annotation at the CFJ class table evaluates to `True`. We need to prove it for code fragments as methods (`method_present_reverse` in Figure 5.1) and classes (`class_present_reverse`).

5.2 Type Soundness Proof

Type soundness of CFJ has already been proven in informal math by Kästner and Apel [KAS]. In this section, we present our formal proof of type soundness, that is very different from the informal proof. We argue why we proved it another way in Section 5.2.1 and explain some assumptions we used for many proofs.

We do not present all proofs in detail in this thesis, since they are verified by the proof assistant Coq. Instead, we present problems arisen in proving type soundness and how we solved them. This way, we give details of those parts of our proofs, which were not straightforward. In Appendix B, we give instructions how to verify our type soundness proof using Coq.

In Section 5.2.2 and Section 5.2.3, we discuss problems dealing with induction or mutual induction. Section 5.2.4 describes problems and our solutions of further problems not dealing with induction. These sections are intended to make clear which problems we solved for our proofs and possibly helps further work with similar problems. Note, that we describe technical details that are probably hard to read for people not familiar with Coq. The reader might want to skip Section 5.2.2 and Section 5.2.3.

5.2.1 The Proof Strategy

Kästner and Apel proved type soundness in two steps. First, they proved that a well-typed CFJ product line stripped of its annotations without removing any code fragments is always a well-typed FJ program. Second, they proved for every code fragment that is removed, that it is not referenced anywhere else at the class table [KAS].

The formalization of the soundness theorem in Coq is straightforward given the formalization of both type systems and the variant generation. But in Coq the theorem can only be proven by reduction on other theorems. It is not obvious how to reduce the type soundness theorem in Coq according to the two steps of the informal proof presented by Kästner and Apel.

Therefore, we decided to use a different proof strategy. We prove that the generated FJ program is well-typed, directly on its structure. It consists of a class table and a start term. In turn, the class table consists of classes with fields and methods and we prove for every part that it is well-typed, i.e., that all referenced classes, methods, fields and variables do exist at the class table. We prove that everything referenced in the FJ program exists at the class table, while Kästner and Apel proved, that everything we remove from the class table is not referenced in the FJ program.

In the following, we denote assumptions used for the proof. The proof of the theorem generation preserves typing is not trivial. We created several theorems, lemmas, and facts to prove it. There are assumptions that we need at several proofs, for which we do not want to give the assumptions explicitly over and over again. Coq provides the ability to specify parameters, which were also used for the FJ and the CFJ class table. We defined three parameters that are listed in Listing 5.2.

```
Parameter aCT_well_typed : a_ok_ctable aCT.

Parameter aCT_correlates_with_CT : variant_ct aCT CT.

Parameter FM : Prop.
```

Listing 5.2: Assumptions for Our Proofs

First, we force that the CFJ class table given as the parameter `aCT` is well-typed. Second, the FJ class table is generated from the CFJ class table. Both assumptions are rather intuitive and premises of generation preserves typing. Third, we assume that the feature model `FM` is given as a parameter. Again, it is used to specify valid configurations and is one premise in our theorem.

5.2.2 Problems with Induction

This section describes problems dealing with induction and how we solve them. First, a general problem with variable quantification in Coq is presented. Second, we discuss a problem with induction and tuple destruction, already known in the Coq community. Third, a problem arisen by the use of existential quantification is shown and two solutions are presented.

Quantification

Almost all of our theorems, lemmas, and facts are proven using induction. But in one case the induction could not be applied as usual. It concerns the proof, that the generated FJ parameter list from a given CFJ parameter list assuming a fixed configuration is unique. This fact is formalized in Listing 5.3.

```
Fact variant_env_unique : forall (fs:a_flds) (fs' fs'':flds),
    variant_env fs fs' →
    variant_env fs fs'' →
    fs' = fs''.
Proof.
  intros fs fs' fs'' H; revert fs''.
  induction H; inversion 1; intuition (f_equal; eauto).
Qed.
```

Listing 5.3: Variant Generation for Parameter Lists is Unique

This fact can be proven using induction on `variant_env fs fs'`, but we tried it unsuccessfully. We created a minimal example where our problem still occurs, i.e., without references to FJ or CFJ definitions. We sent this example to the Coq mailing list and received an answer with a solution.[1]

The problem was, that the variable `fs''` is defined before the premise used for induction. This way, the variable is quantified inappropriate and the proof is impossible. The solution by Strub was to generalize `fs''` before applying the induction, which is done using `revert fs''`. Another possible solution would be, to rewrite the fact. The variable `fs''` can simply be introduced between the first and the second premise. In Coq, one should introduce only what is necessary to apply induction, possibly using `revert`.

Tuple Destruction

We use tuples at several parts of our CFJ formalization, e.g., for class declarations and method declarations. A know problem in Coq is, that induction generalizes all variables, dropping any information on the structure. This is problematically, if we do induction on an expression containing a tuple and one of the tuples elements is used also outside of the tuple. The proof cannot be completed since Coq does replace the tuple with a new variable, but not the elements accordingly.

The problem was discussed at the Coq mailing list by others.[2] Overall, they presented two solutions if induction cannot be avoided. First, prove the theorem without tuples first and use this prove. Second, rewrite all occurrences of elements using the whole tuple, i.e., given a tuple `(a, b)` we can rewrite `a` by `fst (a, b)`.

This problem occurred in our proof of `method_present` and `method_present_reverse`. Since, the proof could not easily be done without tuples, we decided to rewrite all occurrences of tuple elements. Additionally, the generalized tuple need to be destructed into their elements and the elements we rewrote using `fst` and `snd` need to be simplified to elements again. These proofs would be a easier without tuples, therefore we recommend to use tuples only if necessary.

[1]http://logical.saclay.inria.fr/coq-puma/messages/696999107b55d8a6
[2]http://logical.saclay.inria.fr/coq-puma/messages/4db1bb6e78519ef9

Existential Quantification

The proofs of the lemmas `method_present` and `method_present_reverse` are the two most complicated proofs, mainly for two reasons. First, these are the only proofs where we have used three nested inductions; (a) induction over the CFJ method lookup, (b) induction over the class table, and (c) induction over the variant generation for methods. Second, these two proofs could not be solved as usual, since they contain existential quantifications. For example, `method_present` states that if a CFJ method's annotation evaluates to `True`, then the method exists at the FJ class table, while the parameter list and the term are generated using the variant generations (see Listing 5.4).

```
Lemma method_present' : forall (a:ann) (C C0:cname) (m:mname)
  (Dys:a_env) (t:a_exp),
    a →
    AT C →
    a_method C m a (C0,Dys,t) →
    exists Dys':env, exists t':exp,
      variant_env Dys Dys' ∧
      variant_term t t' ∧
      method C m (C0,Dys',t').
```

Listing 5.4: Method Present with Existential Quantification

The problem with the existential quantification of the parameter list and the method's term strongly relies on how we prove this theorem. As we do not want to get further into details, we give a brief explanation. For the proof, we need to provide instances for the parameter list and the term, for which the induction on the variant generation for methods is required. But to prove that the method is part of the FJ class table, the induction on the class table needs to be applied before doing induction on the variant generation—and we have to provide instances for the parameter list and the term before. We solved this contradiction by proving a lemma first, where we eliminated the existential quantification (see Listing 5.5)

```
Lemma method_present : forall (a:ann) (C C0:cname) (m:mname)
  (Dys:a_env) (t:a_exp) (Dys':env) (t':exp),
    a →
    AT C →
    a_method C m a (C0,Dys,t) →
    variant_env Dys Dys' →
    variant_term t t' →
    method C m (C0,Dys',t').
```

Listing 5.5: Method Present without Existential Quantification

The proofs of `method_present'` and `method_present` using `method_present'` require further facts due to this rewrite. We need to prove that the variant generation for parameter lists and terms is a total function, i.e., that for every input CFJ parameter list or CFJ term there is an output and the output is unique (see these facts in Figure 5.2 on Page 62).

In proving `method_present_reverse` it occurred the same problem with existential quantification. Unfortunately, we cannot apply the same strategy as above presented for `method_present`, because the CFJ term (parameter list) from that a given FJ term (parameter list) is generated, is not unique, i.e., variant generation is not a bijective function. For example, a CFJ object creation term with one parameter whose annotation evaluate to false and the same term without a parameter lead to the same FJ term.

Our solution uses the command `cut X`. By this command, we can prove that our theorem holds given that `X` is valid and second, we need to prove that `X` is valid. We needed to use different `X` depending on the case of our induction. For further details, we refer to the full proof (see Appendix B).

5.2.3 Problems with Mutual Induction

This section describes problems dealing with mutual induction. We already introduced mutual definitions in Section 4.2 and Section 4.3. Mutual definitions often require mutual induction, even if we only need to prove a fact on one of two or three mutual definitions. Mutual induction means, that we prove a fact for all (two or more) mutual definitions, assuming that it is proven for all other definitions.

Mutual Induction on Terms

The fact `variant_term_unique` states that, given a CFJ term, the generated FJ is unique. This cannot be proven using straightforward induction on terms, because `variant_term` is defined mutually with `variant_terms`. A proof of a property for `variant_term` always relies on a proof of an according property for `variant_terms`. Listing 5.6 shows how to generate a useful induction principle in Coq.

```
Scheme variant_term_ind2 := Minimality for variant_term Sort Prop
with variant_terms_ind2 := Minimality for variant_terms Sort Prop.

Combined Scheme variant_term_mutind from variant_term_ind2,
  variant_terms_ind2.
```

Listing 5.6: Combined Scheme for Mutual Induction

The command **`Scheme`** generates induction principles for each mutual definition and takes into account the mutual structure. The command **`Combined Scheme`** is used create an induction principle, which can be used to prove a property for all mutual definitions. In Listing 5.7, we present the fact `variant_term_unique_mut` proving the uniqueness for terms and lists of terms at the same time.

Given the induction principle and the combined fact, the proof is straightforward. In `variant_term_unique` we can simply use the proof for terms. The literature on mutual induction in Coq is sparsely. The book by Bertot and Castéran dedicates only four pages on mutual induction [BC04]. We contacted the Coq mailing list for that issue and got support.[3]

[3]http://logical.saclay.inria.fr/coq-puma/messages/786f85b60432ce86

```
Fact variant_term_unique_mut :
  ( forall (t:a_exp) (t':exp),
    variant_term t t' →
    forall (t'':exp), variant_term t t'' → t' = t'' ) ∧
  ( forall (ts:list (ann*a_exp)) (ts':list exp),
    variant_terms ts ts' →
    forall (ts'':list exp), variant_terms ts ts'' → ts' = ts'' ).
Proof.
  apply variant_term_mutind; try inversion 6; try inversion 5;
    try inversion 4; try inversion 3; try inversion 1;
    intuition (f_equal; eauto).
Qed.

Fact variant_term_unique :
  forall (t:a_exp) (t' t'':exp),
    variant_term t t' →
    variant_term t t'' →
    t' = t''.
Proof.
  destruct variant_term_unique_mut; eauto.
Qed.
```

Listing 5.7: Variant Generation for Terms is Unique using Mutual Induction

Mutual Proofs

A very similar fact is `variant_term_total`, whereas our proof is totally different. The reason is that `variant_term_mutind` cannot be applied, because there is simply no premise containing `variant_term`. This relation is only contained at the conclusion (see Listing 5.8).

We could create a fact similar to `variant_term_unique_mut` in Listing 5.7. But we cannot apply `variant_term_mutind`, since this would imply a fact with a structure like `(variant_term x y → P x y) ∧ (variant_terms a b → Ps a b)` and our fact has no premise.

Again, we created a minimal example and asked on the Coq mailing list.[4] The first solutions posted only worked for the example and not for our real fact. The reason is, that we needed to strongly simplify our example. Finally, Auger presented two solutions.

The first solution requires to rewrite the definition of FJ terms. For method invocation terms and object creation terms a new type represents the list of parameter terms instead of a simple list of terms. This way, Coq perceives the mutual definition and we can generate a useful induction principle. Applying this solution would mean to totally overwork the FJ and the CFJ formalization, because terms occur in almost all definitions and theorems.

The second solution by Auger is a trick with a special notation. It uses our idea how we would proof this in informal math; first, we assume `variant_term_total` to prove

[4] http://logical.saclay.inria.fr/coq-puma/messages/b5c4bad3158b6616

```
Definition vtt_trick
  (REC : forall (t:a_exp), exists t':exp, variant_term t t') :
  forall (ts:list (ann*a_exp)),
    exists ts':list exp, variant_terms ts ts'.
Proof.
  intro H.
  induction ts.
    exists nil; auto.

    destruct a as [a t].
    destruct H with (t:=t) as [t' Hvt].
    destruct IHts as [ts' Hts].
    assert (Ha:a ∨ ~a) by apply classic.
    destruct Ha.
      exists (t'::ts'); auto.

      exists ts'; auto.
Defined.

Fact variant_term_total :
  forall (t:a_exp), exists t':exp, variant_term t t'.
Proof.
  refine (fix variant_term_total (t : a_exp) := _); destruct t;
    try destruct (variant_term_total t) as [t' Hvt];
    try destruct (vtt_trick variant_term_total l) as [l' Hvts].
    exists (e_var v); auto.

    exists (e_field t' f); auto.

    exists (e_meth t' m l'); auto.

    exists (e_new c l'); auto.
Qed.
```

Listing 5.8: Variant Generation for Terms is Total

`variant_terms_total`, which is the same fact for lists of terms, i.e., there is a generated list of FJ terms for every list of CFJ terms. Second, we prove `variant_term_total` using the above fact. The idea is a proof on a special assumption, followed by a proof of the assumption.

The proof is realized with the second solution. The definition `vtt_trick` is used at the proof of the fact `variant_term_total`. Using this special notation, the proof is straightforward. We conclude that Coq's support for lists is not satisfying and the usage of lists might be avoided in further formalizations.

Advanced Mutual Induction

The facts `gpt_typing` and `gpt_wide_typing` are also proved using mutual induction. They use the definitions of well-typed terms in FJ and CFJ (see Listing 5.9). Well-typed FJ terms are specified using three mutual definitions `typing`, `wide_typing`, and `wide_typings`, to cover typing of terms, method terms, and terms as parameters. Accordingly, well-typed CFJ terms are defined by `a_typing`, `a_wide_typing`, and `a_wide_typings`.

```
Theorem gpt_wide_typing :
  forall (a:ann) (E:a_env) (t:a_exp) (C:cname) (E':env) (t':exp),
    a_wide_typing a E t C →
    a →
    variant_env E E' →
    variant_term t t' →
    wide_typing E' t' C.

Theorem gpt_typing :
  forall (a:ann) (E:a_env) (t:a_exp) (C:cname) (E':env) (t':exp),
    a_typing a E t C →
    a →
    variant_env E E' →
    variant_term t t' →
    typing E' t' C.
```

Listing 5.9: Generation Preserves Typing for Terms

Furthermore, as both facts formulate generation preserves typing for terms, they reference the variant generation for terms, i.e., `variant_term`. Hence, both theorems are using three independent definitions, where each is mutually defined. Clearly, we need mutual induction to prove this, but it is not trivial to see on what to apply induction.

With our experience from other proofs, we conclude that this can be proven using mutual induction on the variant generation. To apply the combined scheme created in Listing 5.6, we need to create a theorem for proving generation preserves typing at the same time for `a_typing`, `a_wide_typing`, and `a_wide_typings`.

Again, we could not apply this combined scheme without further help from the Coq mailing list.[5] The problem is, that we have to prove three facts with a combined scheme designed to prove two facts at the same time. Creating a minimal example was time-consuming, but an idea by de Fraine lead us to a proof of our facts. We present the combined fact `gpt_mutual_typing` in Listing 5.10.

We merged the fact `gpt_typing` with the fact `gpt_wide_typing`, since for both the FJ term is generated using `variant_term`. We omit the proofs here, since they are long and redundant. The redundancy comes from the induction on the variant generation. The induction principle does not take advantage of the correlation between the definitions `a_typing` and `a_wide_typing`. This results in many similar cases, because a goal including `a_wide_typing` needs to be reduced to a goal using `a_typing`. Therefore, we need to prove the same fact twice under very similar assumptions.

5.2.4 Further Problems

In this section, we describe two problems not dealing with induction and our solutions. First, we present how we realized case analysis on annotations. Second, a problem

[5]http://logical.saclay.inria.fr/coq-puma/messages/786f85b60432ce86

```
Theorem gpt_mutual_typing :
    ( forall (t:a_exp) (t':exp),
      variant_term t t' →
      ( forall (a:ann) (E:a_env) (C:cname) (E':env),
        a_typing a E t C →
        a →
        variant_env E E' →
        typing E' t' C ) ∧
      ( forall (a:ann) (E:a_env) (C:cname) (E':env),
        a_wide_typing a E t C →
        a →
        variant_env E E' →
        wide_typing E' t' C ) ) ∧
    ( forall (ts:list (ann*a_exp)) (ts':list exp),
      variant_terms ts ts' →
      forall (a:ann) (E:a_env) (Cxs:a_env) (E':env) (Cxs':env),
        a_wide_typings a E ts (imgs Cxs) →
        a →
        variant_env E E' →
        variant_terms ts ts' →
        variant_env Cxs Cxs' →
        wide_typings E' ts' (imgs Cxs') ).
```

Listing 5.10: Generation Preserves Typing for Terms using Mutual Induction

with reachability checks is discussed and we make a decision for one of two presented solutions.

Case Analysis on Annotations

A simple proof we want to present in detail is the proof of the fact `variant_env_total`. Given a parameter list in CFJ, it exists a parameter list in FJ that is generated from the former list. The formalization of this fact is straightforward (see Listing 5.11).

We proof this fact using induction on the list of CFJ parameters `fs`. The induction beginning: if the CFJ list is empty, then the FJ list is empty, too. Otherwise, we need a case analysis on the annotation of the first parameter to prove the induction step. The parameter is included, if and only if the annotation evaluates to `True`.

The intended case analysis can be done using `destruct` on an assumption stating that `a ∨ ~a`, whereas `a` is the annotation of the parameter. We get two new subgoals, where the first has an assumption stating `a` holds, and the second has an assumption stating `~a` holds.

What remains unclear is how we prove `a ∨ ~a` in Coq. For this proof we need to add a further library named `Coq.Logic.Classical_Prop` to our imports. It contains an axiom stating exactly this formula. It is called the excluded middle and cannot be proven using Coq.

Type Present in Term Typing

This section describes another issue with the proof of `gpt_mutual_typing`. Several cases of the proof rely on the fact, that for every well-typed term in CFJ, the type is

```
Fact variant_env_total :
  forall (fs:a_flds),
    exists fs':flds, variant_env fs fs'.
Proof.
  intros fs.
  induction fs.
    exists nil; auto.

    destruct a as [f [a C]].
    destruct IHfs as [fs' Hve].
    assert (Ha:a ∨ ~a) by apply classic.
    destruct Ha.
      exists ((f, C)::fs'); auto.

      exists fs'; auto.
Qed.
```

Listing 5.11: Variant Generation for Parameter Lists is a Total Function

always reachable regarding the current annotation. We already proved this fact for CFJ in informal math (see Lemma 3.1 on Page 32).

We give an example to illustrate why we cannot avoid to prove this in order to prove `gpt_mutual_typing`. Using the induction principle explained in the previous section, we prove that the FJ term is well-typed if it is generated from well-typed CFJ term. A sub case is, that the term is a variable and we prove the property for wide typing, i.e., the term has a subtype of a particular type.

Therefore, we need to prove the following: the subtype exists at the FJ class table. The only way to prove this, is to evaluate the annotation of that class, because a class is present at the FJ class table, if and only if its annotation evaluates to `True`. We need a proof of this, for all shapes of terms, i.e., the term is a variable, a field access, a method invocation, or an object creation term. The reason is that all these terms can occur in a method, where we need this wide typing. Hence, we need to prove Lemma 3.1 in Coq.

Unfortunately, the proof is not so easy like in informal math. The following sentence from our informal proof is problematically: “Context is only created in method typing.” In Coq, we cannot access all positions from where a definition is called. In the following we present how to prove this.

A term we type is either the start term, a method’s term, or part of it. The start term has an empty context and the problematically case of variable typing cannot occur in well-typed start terms. For typing of all other terms we can use the class and method of the term instead of the environment for typing. First, the environment can be retrieved using class and method and second, we can inductively prove, that the annotations at the context always imply that the type is present.

The proof of Lemma 3.1 would need large rewritings of the definitions and adaption of many proofs. Therefore, we decided to add further checks in term typing. We highlighted these additional premises in Listing 4.11 on Page 48. For object creation terms we do not need to add this check, since it is already part of the CFJ type system.

Notice, all three premises are redundant and do not change the type system, since they are already fulfilled according to Lemma 3.1. Nevertheless, this thesis is about proving type soundness of CFJ formally and this is a potential weakness of our formalization. Assuming our proof of Lemma 3.1 is wrong, than our proof would still be valid, but the type system we formalized in Coq might be more restrictive than the CFJ type system, for which we want to prove type soundness.

5.3 Summary

We proposed a formalization of the type soundness theorem for CFJ in Coq. It states that generation preserves typing, i.e., every FJ program generated from a well-typed CFJ product line is well-typed. Since the theorem is too heavy to prove at once, we presented six further theorems, eleven lemmas, and eleven facts reducing the complexity.

While our formalization of the CFJ type system and the variant generation has a total of 467 lines, all proof scripts sum up to 1081 lines. The proof strategy we used is totally different from the existing proof in informal math. The idea is to prove that everything referenced in the FJ program exists at the class table.

Proving type soundness for CFJ in formal math was *not* straightforward. We discussed eight problems and how we solved them. Three times, we asked for help on the Coq mailing list and one solution was already at the archive of the mailing list. The proofs would probably not be completed within the time restrictions of this thesis without the help by others on the mailing list.

For one problem we decided to reasonably change the type system to reduce the complexity of the proof scripts. This change does not affect the proof of type soundness, but might make the CFJ type system more restrictive. We proved in informal math that these changes have no influence on the behavior of the type system.

Coq has the ability to print the axioms used to prove a certain theorem. The axioms are important, as a wrong axiom would make the proof incorrect. Listing 5.12 prints all axioms used to prove generation preserves typing. We highlighted the assumptions due to our formalization of CFJ. All other assumptions are caused by the underlying formalization of FJ by de Fraine.

The axiom `aCT` is similarly to that of FJ and needed to make the CFJ class table available to all theorems and definitions. For the same reason the assumptions `FM`, `aCT_correlates_with_CT`, and `aCT_well_typed` are modeled as parameters. They are part of the assumptions at the type soundness theorem. A further axiom needed for case analysis on annotation is that of classical logic, which seams reasonable in this context.

```
Axioms:
FM : ann
aCT : a_ctable
aCT_correlates_with_CT : variant_ct aCT CT
aCT_well_typed : a_ok_ctable aCT
Classical_Prop.classic : forall P : ann, P ∨ ~ P
CT : ctable
Object : cname
this : var
atom : Set
eq_atom_dec : forall x y : atom, {x = y} + {x <> y}
```

Listing 5.12: Axioms of Our Type Soundness Proof

6. Experiences

Whenever we want to prove properties about certain definitions, we can chose between an informal proof and a formal proof, which can be verified by a proof assistant. We want to share our experiences writing formal proofs in Coq to help others making a decision to write formal or informal proofs.

We start with providing some statistics on the effort of this thesis and the parts of our formalization in Section 6.1. Then, we explain some challenges we encountered while working with the proof assistant Coq in Section 6.2. Advantages of formal proofs that we experienced are described in Section 6.3. Finally in Section 6.4, we summarize our experiences and give an advice when to use formal instead of informal proofs.

6.1 Estimated Effort of Our Proof

In this thesis, we developed a formalization of CFJ in Coq and provided a formal proof of type soundness. We were already familiar with software product lines and annotations, but not with type systems or formal proofs. In Figure 6.1, we estimated the time needed for getting familiar with basic theories, formalizing CFJ, and proving type soundness. This consideration might help to predict the effort for similar theses.

Getting a handle on formalizing definitions and proving properties in Coq took more than six weeks. Practicing proof writing on small examples was essential to complete the large proofs for CFJ. We completed some tutorials before we could start with the formalization presented in this thesis.

The thesis forced us to get familiar with the theory of type systems. More than three weeks were necessary to learn the basic concepts. Additionally, we spend about three more weeks to understand all details of FJ and CFJ. This includes the syntax, the type systems, and the informal type soundness proof of CFJ.

The introduction to CFJ and to Coq was not completely done before we started with our actual challenge. But in total, we needed about three months to have the knowledge and practice to formalize CFJ in Coq, which is about 55 percent of the total time.

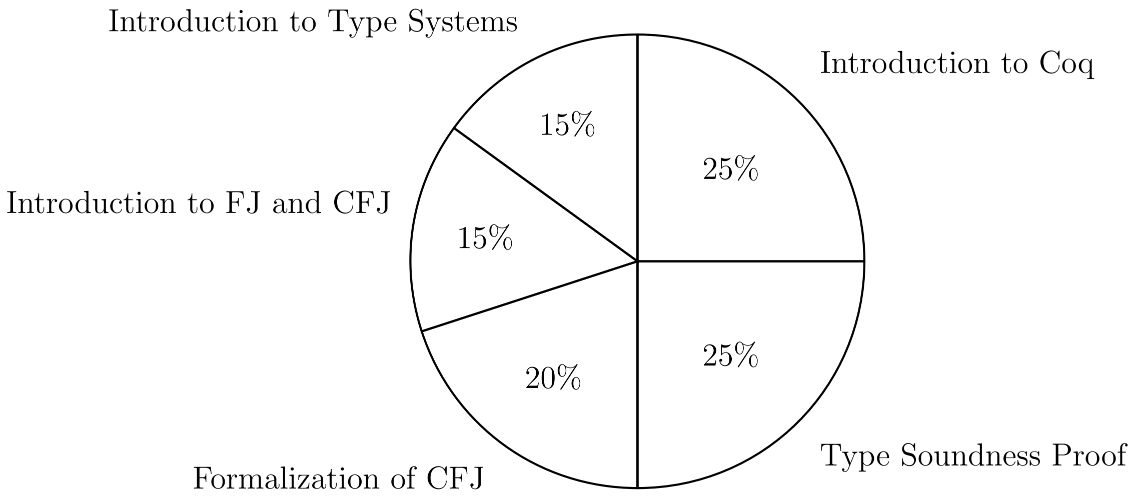


Figure 6.1: Estimated Effort in Formally Proving Type Soundness for CFJ

Roughly, we worked five weeks on the formalization and approximately seven weeks on the type soundness proof. Again, the formalization was not entirely completed before we started working on the proof. However, while proving type soundness we found several errors in our formalization. For some errors large rewriting on the definitions and proofs were indispensable.

Getting familiar with interactive theorem proving in Coq was similar to learning the first programming language, e.g., Java. First, one learns how interactive theorem proving works in general—we have goals and tactics can manipulate, split or solve goals (similar to learn about control flow statements in Java). Second, we needed to understand how to write proofs in Coq (analog to programs in Java), i.e., what are the basic tactics and how do we formalize our definitions and theorems. This analogy points up the effort for getting familiar with theorem proving in Coq.

We want to give a detailed view on the effort of the formalization of CFJ and the type soundness proof. Our formalization consists of five files, containing (a) the definitions of type soundness and variant generation, (b) facts and their proofs, (c) lemmas and their proofs, (d) theorems and their proofs, and (e) an example CFJ product line and a proof that it is well-typed. In Figure 6.2, we visualize the total number of lines and the estimated lines written and removed, e.g., lines containing definitions that were replaced or unsuccessful proof scripts.

We want to explain the order these files were written, because it largely influences the overhead in writing. Of course, we started writing `CFJ_Definitions.v` containing all definitions for the type system and the variant generation. It was followed by `CFJ_Properties.v` containing the proofs of all theorems, since we decided to prove type soundness top-down.

While proving the theorems, we found mistakes in our definitions. The changes in `CFJ_Definitions.v` lead to overhead for the definitions and for already proved theorems, since they strongly rely on the definitions. We continued proving several facts

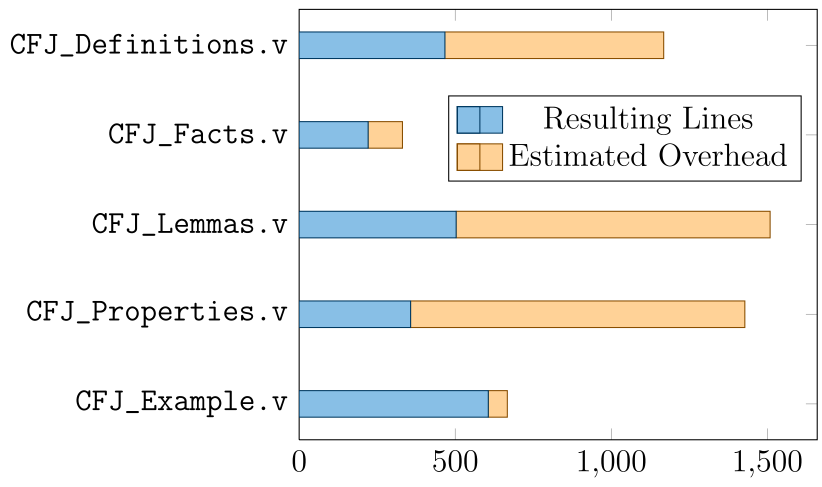


Figure 6.2: Estimated Effort for Each Part of the Formalization

and lemmas in `CFJ_Facts.v` and `CFJ_Lemmas.v`. The proofs again made it necessary to change definitions and all referencing parts in the proof scripts.

The file `CFJ_Example.v` could be created with little overhead, because it was written at last. The definitions were almost correct, we only found smaller bugs concerning a too restrictive type system. The large overhead for `CFJ_Lemmas.v` is caused by very complex proofs which needed several tries combining tactics in different ways.

The time Coq needs to verify our proof scripts is not negligible. We used a standard notebook with a Windows Experience Index of 3.2 and Coq consumes up to three minutes to check our formalization. In Figure 6.3, we present the time in seconds needed for each file. We measure the time to check all commands and the time to compile. Compilation is necessary to use the definitions and theorems in other files.

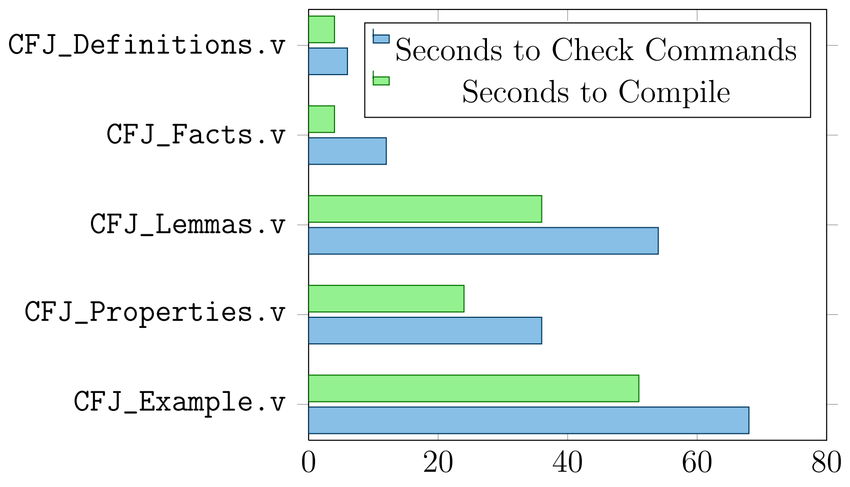


Figure 6.3: Time Coq Needs to Check Our Formalization

Compilation is always a little bit faster. The reason might be that no graphical user interface needs to be refreshed after every command. Checking our definitions takes 6 seconds, while our large proof scripts of our lemmas need up to 54 seconds. There is no obvious reason for us, why it takes so long to verify the proof, that our example is a well-typed CFJ product line.

Note, that whenever we change the some small part of the definitions and want to make sure, that the proof scripts remain valid, it takes up to three minutes to compile all files. Whenever we change one of the files, Coq forces us to recompile this file. We spent a lot of time in waiting for Coq's response. An incremental check as known from modern IDEs would be valuable.

6.2 Challenges with Proof Assistants

The first challenge is the high effort to *get familiar* with the proof assistant's language and interactive theorem proving in general. When starting this work, we could hardly read existing definitions and proof scripts in Coq. We give an example of a hard-to-read mutual induction principle generated by Coq in Listing 6.1, e.g., we used this principle to prove `gpt_mutual_typing` and `variant_term_unique_mut`.

Everything needs to be *formalized*, e.g., we could not easily omit formalizing the annotation table in our formalization of CFJ as by Kästner and Apel [KA08]. Moreover, we cannot use all proof ideas available for informal proofs. For instance, it is not possible to reason on all definitions that use a particular definition, since proofs are checked command-wise and a further command can come up with such a new definition. We explained this problem in Section 5.2.4.

Proofs scripts in Coq are hard to be verified by human. They are intended to be verified by the proof assistant and we need to *trust* its algorithm. Furthermore, we need to trust in the compiler and the used hardware, which is more hypothetically. Basically, we need to verify by ourselves that the definitions are formalized correctly.

Proof assistants evolve over time, e.g., their languages are extended or changed. Therefore, a formalization including the proofs might only be accepted by specific *versions of the proof assistant*. This especially becomes a problem if one wants to use formalizations together which do not compile with the same version of the proof assistant. For example, we had the problem to port a FJ formalization to the most current version of Coq (see Section 4.1.1).

A proof usually relies on definitions. When definitions and proofs get more and more complex, a change of a definition is often very expensive. The reason is that proof script rely on names of variables, on their order, on the order of premises and so on. If we need to change a definition, we usually need to *adapt proof scripts* that rely on the particular definition. For example, we had to change how annotations are realized. First, we stored annotations in an annotation table, but it was not obvious how to map different annotations to identical terms occurring at different positions. Hence, we decided to store annotations directly at the class table. This caused rewrites of almost all definitions and proofs, which took several days. But, there are also changes

```
variant_term_mutind
     : forall (P : a_exp → exp → ann)
         (P0 : list (ann * a_exp) → list exp → ann),
       (forall x : var, P (ae_var x) (e_var x)) →
       (forall (t : a_exp) (t' : exp) (f0 : fname),
        variant_term t t' → P t t' →
                P (ae_field t f0) (e_field t' f0)) →
       (forall (t : a_exp) (t' : exp) (m : mname)
                (ts : list (ann * a_exp)) (ts' : list exp),
        variant_term t t' →
        P t t' →
        variant_terms ts ts' →
        P0 ts ts' → P (ae_meth t m ts) (e_meth t' m ts')) →
       (forall (ts : list (ann * a_exp)) (ts' : list exp) (C : typ),
        variant_terms ts ts' → P0 ts ts' →
                P (ae_new C ts) (e_new C ts')) →
       P0 nil nil →
       (forall (ts : list (ann * a_exp)) (ts' : list exp) (a : ann)
          (t : a_exp) (t' : exp),
        variant_terms ts ts' →
        P0 ts ts' →
        a → variant_term t t' → P t t' →
                P0 ((a, t) :: ts) (t' :: ts')) →
       (forall (ts : list (ann * a_exp)) (ts' : list exp) (a : ann)
          (t : a_exp),
        variant_terms ts ts' → P0 ts ts' → ~ a →
                P0 ((a, t) :: ts) ts') →
       (forall (a : a_exp) (e : exp), variant_term a e → P a e) ∧
       (forall (l : list (ann * a_exp)) (l0 : list exp),
        variant_terms l l0 → P0 l l0)
```

Listing 6.1: Example for a Mutual Induction Principle in Coq

to definitions we avoided, because of the overhead, e.g., according tuples in Section 5.2.2 and according the location of terms at the class table in Section 5.2.4.

Before a proof is written in formal as well as in informal math, we need to have a proof idea, i.e., a clear understanding how to prove a theorem. Given this idea it is challenging task to find the proof assistant's *tactics* needed to realize the proof. Coq's manual lists all tactics with a short description each and we often searched through all these tactics to find a suited one. We still needed help by the Coq mailing list, as the documentation of tactics is insufficient, e.g., syntactical elements used to perform mutual proofs are not documented (see Section 5.2.3).

6.3 Advantages of Formal Proofs

One of the most obvious advantages is that our proofs scripts are *verified*. In particular, this gives non-mathematician proof writers a good feeling that they are done and did a good job. A proof reader can concentrate on the definitions and axioms to find out if the formalization is useful in the context. For example, we found inconsistencies at the formalization of FJ in Coq (see Section 4.1.2). We adapted some definitions and proof scripts if necessary, for what we needed to understand only particular proofs, i.e., those where the verification failed.

When writing a proof in a proof assistant the proof is *checked stepwise*. A mistake in the proof can be found very early. This prevents from working on wrong assumptions which might be time consuming. Every time a further step is accepted by the proof assistant we know that it is correct, while it does not need to be a promising direction. For instance, we made a typical human mistake and tried to deduce a false conclusion (a post-hoc fallacy), e.g., given `A → B` and `B`, we wanted to show `A`. Fortunately, Coq did not except that and we recognized our mistake.

A proof assistant provides at each step a complete *overview* about the assumptions and the things left to show. In this manner, we cannot forget to proof a subgoal and it is impossible to use an assumption not available at this step. Additionally, it gives an idea how far we are from finishing the proof. A higher number of subgoals to show usually results in more steps to do. Especially, when continuing a particular proof after an intermission, we found the complete overview on assumptions and sub goals very helpful to get back into the proof.

Writing proofs, we might occur a case that cannot be proven easily. Sometimes, this *indicates an error* at the definitions and also the concrete position of that error. A proof assistant does not only verify proof scripts, it also points out locations of possible errors in definitions or theorems. For example, we found an error in our definitions concerning the CFJ method lookup. For a method that is not always reachable in all variants were it is called, the lookup continues at the super class. Our error was, that the method declaration has be identical to that of the super class. Hence, the proof of `method_present_reverse` was impossible and we uncovered this error. But we also found an error in the informal definition of the CFJ type system, where `A ↔ B ↔ C` was used to express that the value of these three variables is identical, which is wrong independent of how we put it into parenthesis (see Section 3.2.4).

6.4 Summary

We presented some statistics on this thesis, showing that for our machine-checked proof we needed six weeks to get familiar with interactive theorem proving in Coq. Furthermore, we shown that writing our machine-checked proof consists of some effort that not directly leads to the formalization. More than 3000 lines were written useless. We argued that the time Coq needs for verification—up to three minutes for our formalization—is an issue.

Proof assistants come with certain new challenges. Definitions and theorems are often hard to read by humans and proof scripts are almost impossible to verify by human that we have to trust in the proof assistant. We explained problems with both evolving definitions, proof scripts need to be adapted, and evolving proof assistants, whereas formalizations only compile with particular versions of the proof assistant. Finally, finding appropriate tactics is not always easy.

On the other hand, formal proofs have strong advantages. Our proof scripts are verified by the proof assistant and humans can concentrate more one the definitions itself. Already incomplete proofs can be checked and errors can often be detected very early. Proof assistants can assist the proof in giving an overview on the assumptions and the goals remaining to prove. A rejected proof step sometimes points out errors at the formalization.

Considering all these advantages and disadvantages of formal proofs, we can neither recommend formal proofs for all applications nor advise against them. Our suggestion is that machine-checked proofs are used, whenever it is easier for a human to verify the formal definitions and theorems in a proof assistants language than to verify the informal proof.

Furthermore, this advantage should be so significant that the additional challenges of interactive theorem proving are worth. We might get better results by analyzing the humans who want to verify our proof. For a community with a good knowledge in interactive theorem proving or even in the particular proof assistant, it will be easier to verify the formal definitions than for others without this knowledge.

7. Related Work

We first discuss related work on extensions of Colored Featherweight Java (CFJ). Kästner et al. present an extension of CFJ with alternative features [KAS]. In 2009, Rosenthal worked on particular problems dealing with alternative features in CFJ [Ros09]. Alternative features do not occur in the same variant, as such variants are prohibited by the feature model. Their work focuses on a type system that allows alternative features and an implementation in CIDE. Notice, that the type system of our consideration forbids alternative features to conserve backward compatibility, e.g., alternative implementations of one method. Furthermore, our focus is more on proving the correctness of a given type system.

Beside CFJ, there are two other software product line approaches using annotation to achieve variability. Czarnecki and Pietroszek present an automated verification procedure for feature-based model templates guarantying that all template instances generated from using a valid configuration are well-typed [CP06]. A feature-based model template consists of a feature model and a model template. They use annotated UML class models as model templates and argue that also other model templates are possible. At the UML class model, classes and connections can be annotated with features or propositional formulas on features of the feature model. Given a valid configuration, the template instance is generated, while all classes and connections whose annotation evaluates to false are removed as in CFJ. Contrary to CFJ, they have not presented a type system for that type soundness could be proven.

Huang et al. proposed a similar approach based on a programming language named cJ [HZS07]. cJ adds to generic Java classes the ability to annotate super types, fields, and methods with type conditionals. Given a generic class with one or more parameter types, a type condition checks whether the parameter types are sub types of a particular given type. Only if the type condition is fulfilled, the following super class, method or field is present at the generated variant. Contrary, CFJ additionally supports annotations for parameters. cJ is more interesting from a practical point of view as it is based on full Java 5, but a type soundness proof for cJ is impractical for that reason. They informally prove type soundness for a subset of cJ named Featherweight cJ (FCJ).

In the following, we discuss three approaches to type check feature-oriented software product lines. Thaker et el. present safe composition to guarantee the absence of references to undefined elements, e.g., classes [TBKC07]. Their approach uses a satisfiability solver to check that all implementation constraints are implied by the feature model and is based on the feature-oriented language Jak. It is a programming language similar to Java enriched with new keywords to express variability. Their type checks are not complete, e.g., a parameter list could contain a reference to a type that is undefined.

Ott is a tool to semantically define full-scale programming languages [SNO$^+$07]. A meta language is introduced to simplify the precise definition of large programming languages. Once defined in Ott, a formal representation in Coq, Isabelle or HOL and an informal representation in LaTeX can be generated. This is especially useful to make sure that informal and formal representations are synchronized correctly. Lightweight Java (LJ) is a fragment of Java proposed by Strniša et al. [SSP07]. They used Ott to formally describe LJ in the proof assistant Isabelle/HOL. It was originally used to introduce the Java Module System and is imperative, whereas FJ is only functional, since variable assignments are not supported. As FJ, LJ is a proper subset of Java, i.e. every LJ program is a Java program. Delaware et al. present a type system in Coq for Lightweight Feature Java (LFJ), an extension of LJ with support for feature modules [DCB09]. The type system is proven to be sound using Coq. Furthermore, a constrained-based type system has been derived that a satisfiability solver can be used to check whether all products specified by a feature model are type-safe Lightweight Java programs.

Apel et al. present Feature Featherweight Java (FFJ) as an extension of FJ to support feature-oriented programming [AKGL09]. Furthermore, they provide Feature Featherweight Java Product Line (FFJ_{PL})—a language for feature-oriented product lines. Type systems were presented for both languages. They prove the type soundness of the FFJ type system and give a proof sketch for the correctness of the FFJ_{PL} type system. They state that FFJ_{PL} has a property called completeness, meaning that if all program variants that can be generated are well-typed, then also the product line is well-typed. Note, that this property does not hold for CFJ, because of backward compatibility, e.g., two methods with the same name in one class are not allowed, even if they do not occur in the same program variant.

There is also work on type checking aspect-oriented software product lines. Kammüller and Vösgen present their ongoing work towards type soundness for aspect-oriented languages [KV06]. Similar to our work, they work with an extension of FJ and formalize the type systems in Coq. The main difference is that their extension is aspect-oriented. Additionally, they are not yet able to give proofs as they first have to come up with a solution to the runtime weaving problem. This problem is specific to aspect-oriented languages.

8. Conclusion

Software product lines and type system, both intend to build software more efficiently. When applying type systems to software product lines, it is not feasible to generate each program variant and type-check it separately. Hence, product-line–aware type systems have been proposed to check the software product line at once. Type soundness proofs for these type system are long and hard to verify by human, what makes it worth to prove them formally using a proof assistant.

We present the first machine-checked type soundness proof for CFJ, i.e., we show that the CFJ type system we formalized is correct. Given a software product line that is well-typed according to our CFJ type system, all program variants that can be generated using a valid configuration are well-typed FJ programs, e.g., they cannot lead to errors as dangling method references.

We propose a simplified type system for CFJ based on the type system by Kästner et al. [KAS] and prove its semantically equivalence in informal math. Furthermore, by proving type soundness formally, we found an error in the informal type system, which remained uncovered at the informal proof. We presented the revised type system informally and in Coq. Both representations may be useful for future work.

Our machine-checked type soundness proof verifies that our formal type system for CFJ is correct. From this follows that our informal type system is correct, if (a) all product lines well-typed in our informal type system are also well-typed according to the formalized type system, (b) type soundness can also be proven for casting, which we omitted in our formalization, (c) type soundness can be proven for the formal type system without the three premises we added to simplify the proof, and (d) we trust into machine-checked proofs using Coq. But it might also be correct, if not all these conditions are fulfilled.

We reported our experiences in writing machine-checked proofs in Coq, to help others to decide for a formal or informal proof. We discussed several advantages and even more challenges with proof assistants. Our recommendation is that machine-checked proofs should be used, whenever a human can better verify the formal definitions and theorems than an informal proof. In the case of type systems, we conclude that formally proving type soundness using a proof assistant is worth, mainly because of three reasons. First, type soundness proofs get used to be proven with interactive theorem proving and so the community is familiar with verifying formal definitions. Second, the proofs tend to have many cases that are probably harder to verify than the formalized definitions with a manageable amount of cases. Third, there is a chance to detect inconsistencies in informal definitions.

Formalizing the CFJ type system, variant generation and proving type soundness for CFJ is not straightforward. We documented our problems with the proof assistant Coq and present our solutions. Future work on product-line–aware type system—or even interactive theorem proving using Coq in general—might profit from our detailed problem descriptions. The discussion of our positive and negative experiences with Coq can also help others to make a decision to do an informal proof or a formal proof using a proof assistant.

Our main contribution is the machine-checked type soundness proof for CFJ. We present a revised type system for CFJ in informal math and in Coq, where we overcome redundant typing rules and an inconsistency due to informal notations.

9. Future Work

Improving our Proof

Our type soundness proof can probably be simplified. In Section 5.2.2, we presented the proof idea used to prove `method_present_reverse`. The idea is to use the command `cut` to cleverly eliminate the existence quantifiers. We believe that it can also be applied to prove `method_present`, which would possibly result in six facts and one lemma less. If so, we would not need the library with the classical logic for any of the remaining proofs.

In Section 5.2.4, we described a proof that we completed by adding premises, for which we shown informally that they are redundant. Future work could formally prove the redundancy or do the proof on the unchanged type system, to prove formally that the formalized type system is not more restrictive than the type system in informal math. This work includes large rewritings of definitions and proofs as explained in Section 5.2.4.

Our adopted FJ formalization has two inconsistencies to the informal definition of FJ, as described in Section 4.1.2. First, the formalization does not support casting. Second, constructors are not represented in this formalization. Future work might overcome one or both limitations and rewrite definitions and proof scripts of the FJ formalization.

Further Proofs on CFJ

Kästner and Apel informally proved backward compatibility of CFJ, i.e., every well-typed CFJ product line stripped of its annotations without removing code fragments is a well-typed FJ program [KA08]. Future work could provide a machine-checked proof of this property. Furthermore, we can think of a similar property that we name forward compatibility: every well-typed FJ program enriched with empty annotations and a possibly empty feature model is a well-typed CFJ product line. Both properties are especially interesting from a tool perspective, since with backward compatibility we can reuse tools of the host language and with forward compatibility we can start with a single program to develop a product line.

For some other product-line–aware type systems than CFJ, e.g., the type system of FFJ$_{PL}$, we can prove that if all generated programs are well-typed, then also the product line is well-typed. We cannot proof this for CFJ, because of its backward compatibility. But we might be able to find an understated, provable property. Research in that direction may lead to a less restrictive CFJ type system, for that we can still prove type soundness.

Beyond CFJ

FJ and CFJ cannot be seen as real programming languages as there are no base types and no operations on them. Additionally, there are no language constructs for conditions and repetitions. Even if FJ is expressible as the typed lambda-calculus, it cannot practically be used to write programs. Our examples are fare away from industrial practice. Future work might prove type soundness for richer languages or extend FJ if the effort is reasonable.

Beside CFJ, other product-line–aware languages can be formalized in Coq. For instance, the formalization of FFJ$_{PL}$ might be valuable as no informal and no formal type soundness proof exists so far. Therefore, a formalization of the variant language FFJ would be required. Future work might then formalize the refactorings between FFJ$_{PL}$ and CFJ presented by Kästner et al. [KAK09].

Based on an FJ formalization by de Fraine, we presented an adapted FJ formalization with different method overriding and a change concerning incomplete class tables. That our changes can be retraced and a certain change can be undone, we marked them in the formalization and kept the removed lines in comments. This way, we could produce a FJ formalization with our fix forbidding incomplete class tables and de Fraine's method overriding. Our vision are proof product lines that can be used to represent a product-line of formalizations and proofs. As for software product lines, checking all proof variants individually is not feasible as the number of changes increases and we need new techniques to check that all proof variants are valid.

Improving Tools for Coq

We want to come back down to earth. Essential for efficient interactive theorem proving are proof assistants providing sophisticated IDEs as known from programming. We worked with CoqIDE, a very simple tool for proof writing shipped with Coq. There is no support for incremental compilation, which would strongly reduce the user's time waiting for Coq's checks. While Coq uses a type system to check validity, the front-end has no such functionality as renaming identifiers or content assist. Finally, we would have profit of any kind of dependency overviews on definitions and theorems, e.g., determine all positions where a particular theorem is referenced. It took hours to keep the graphics in this thesis updated, while we needed them for proving. Hopefully, there will be future work on developing modern IDEs for interactive theorem proving.

A. FJ Formalization in Coq

Our formalization of CFJ and our type soundness proof are based on a formalization of FJ in Coq. We used a FJ formalization by de Fraine that we adapted to overcome two inconsistencies presented in Section 4.1.2. In Section A.1, we give a proof that incomplete class table are classified as well-typed with the formalization by de Fraine. We present a new FJ class typing that prohibits incomplete class tables. In Section A.2, we propose a new definition for valid method overriding.

Our adjusted FJ formalization is publicly available (see Appendix B). We use the following convention that our changes can be identified. Each edit begins with *(*EDITn BEGIN*)* and ends with *(*EDITn END*)*, where `n` indicates the correction (1 for the class table and 2 for the method overriding). In between these markers, we commented out the statements of the before version and added our new version. This way, also version can easily be obtained were only particular adjustments are applied. For instance, someone might need a formalization of FJ that prohibits incomplete class tables and realizes method overriding as formalized by de Fraine.

A.1 Incomplete Class Tables

The typing of classes in de Fraine's formalization does not fulfill the following sanity condition we presented in Section 3.1.3: for every class name C (except Object) appearing anywhere in CT, we have $\mathsf{C} \in \mathsf{dom}(\mathsf{CT})$. In Figure A.1, an example is presented that is well-typed according to the formalization. Apparently, the class name D appears in the class table, but $\mathsf{D} \notin \mathsf{dom}(\mathsf{CT})$. We give a machine-checked proof using Coq in Listing A.1.

The cause lies in the typing of classes; we give the according definition in Listing A.2. In Line 3, `fields D fs'` evaluates to true, if the class table contains a type `D` with the fields `fs'`. Since, `D` does not appear in our class table there is no such `fs'` that `fields D fs'` evaluates to true. Therefore, the implication is always true and it is not checked that `D` is in the class table. Note, that the following line only checks that the methods are well-typed.

```
class C extends D {
    C() { super(); }
}
```

```
new C()
```

Figure A.1: An Example of an Incomplete Class Table

```
Require Import Metatheory.
Require Import FJ_Definitions.
Require Import FJ_Facts.

Variable A : cname.
Variable B : cname.
Hypothesis A_fresh : Object <> A.
Hypothesis B_fresh : Object <> B ∧ A <> B.

Hypothesis ct_fix: CT = (A,(B,nil,nil)) :: nil.

Module ExOkTable: OkTable.

Fact nobinds_binds : forall (x:cname) (a:cname*flds*mths) E,
    no_binds x E → binds x a E → False.
Proof.
  unfold binds. unfold no_binds. intros.
  rewrite H in H0. discriminate.
Qed.

Lemma ok_ct: ok_ctable CT.
Proof.
  rewrite ct_fix in ⊢ *.
  unfold ok_ctable in ⊢ *.
  split.
    auto 8 using nobinds_nil, nobinds_cons.

    apply fa_cons; try apply fa_nil.
    unfold ok_class' in ⊢ *.
    unfold ok_class; repeat split;
      [intros | apply ok_nil | apply fa_nil].
    inversion H; clear H; subst.
      contradict H1; apply B_fresh.

      contradict H0; unfold not.
      apply nobinds_binds with (x:=B) (a:=(D, fs, ms)) (E:=CT).
      rewrite ct_fix.
      apply nobinds_cons; try apply nobinds_nil.
      set B_fresh; destruct a; intuition.
Qed.

End ExOkTable.
```

Listing A.1: Proof that an Incomplete FJ Class Table is Well-Typed

```
Definition ok_class (C: cname) (D: cname) (fs: flds) (ms: mths) :
  Prop :=
    (forall fs', fields D fs' → ok (fs' ++ fs)) ∧
    ok ms ∧ forall_env (ok_meth' C D) ms.
```

Listing A.2: A Definition Allowing Incomplete Class Tables [Fra09a]

In Listing A.3, we present a slightly different definition that satisfies our sanity condition. We changed the **`forall`** quantifier into an **`exists`** quantifier to make sure that the class table contains a class `D` with the possibly empty list of fields `fs'`. We can do this, since the class table maps a class to a list of fields if the class is contained in the class table. Additionally, we changed the implication to a conjunction, as we want to force that `fields D fs'` evaluates to true.

```
Definition ok_class (C: cname) (D: cname) (fs: flds) (ms: mths) :
  Prop :=
    (exists fs', fields D fs' ∧ ok (fs' ++ fs)) ∧
    ok ms ∧ forall_env (ok_meth' C D) ms.
```

Listing A.3: The Corrected Definition Prohibiting Incomplete Class Tables

Thereupon, we also need to change proofs based on this definition. Fortunately, these were only trivial changes to three proofs.

A.2 Method Overriding

In Section 3.1.3, valid method overriding implies that the super method and the method have the same return type. Full Java also allows the usage of subtypes and so de Fraine's formalization. We give an example in Figure A.2 that is allowed according to the formalization but not in FJ.

```
class D extends Object {
    D() { super(); }
    D create() {
      return new D();
    }
}
class C extends D {
    C() { super(); }
    C create() {
      return new C();
    }
}
```

```
new C()
```

Figure A.2: Method Overriding with Differing Return Types

Listing A.4 shows the definition allowing subtypes. The relation `sub t t'` evaluates to true if `t` is a subtype of or identical to `t'`.

```
Definition can_override (D: cname) (m: mname) (t: typ) (E: env) :
  Prop :=
    forall t' E' e, method D m (t',E',e) → sub t t' ∧ imgs E = imgs E'.
```

Listing A.4: A Definition Allowing Different Return Types [Fra09a]

We present a corrected definition in Listing A.5. The only change is that we force `t = t'` instead of `sub t t'`.

```
Definition can_override (D: cname) (m: mname) (t: typ) (E: env) :
  Prop :=
    forall t' E' e, method D m (t',E',e) → t = t' ∧ imgs E = imgs E'.
```

Listing A.5: The Corrected Definition Forcing Identical Return Types

As for incomplete class tables, this change needs to be propagated to theorems and proofs using this definition. In total, one theorem and two related proofs needed to be adapted.

B. CFJ Formalization in Coq

Our formalization of CFJ including the type soundness proof is publicly available.[1] In Section B.1, we provide information on our formalization necessary to verify it using Coq. Section B.2 and Section B.3 present definitions we omitted in previous chapters.

B.1 Verification of Our Formalization

Accordingly to the formalization of FJ by de Fraine, we partition our formalization into several files. In Table B.1, we give an overview on the files and their content. We separated the definitions for the CFJ type system and the variant generation from all proofs. Additionally, we split the proofs into the proofs of theorems, lemmas, and facts.

File	Content
`CFJ_Definitions.v`	Definitions for type system and variant generation
`CFJ_Facts.v`	Facts including Proofs
`CFJ_Lemmas.v`	Lemmas including Proofs
`CFJ_Properties.v`	Theorems including Proofs
`CFJ_Example.v`	Example for a well-typed CFJ product line and a generated FJ program including Proofs

Table B.1: The Files of Our Formalization

Furthermore, in `CFJ_Example.v` we give an example of a well-typed CFJ product line and a generated FJ program according to a given valid configuration. We give proofs that the product line is well-typed and the program is generated from the product line. We conclude that the FJ program is well-typed applying our generation preserves typing theorem.

Using a Coq file in another one, we need to import and compile the file. The order to compile the files really matters, since all imported files need to be compiled first.

[1] http://wwwiti.cs.uni-magdeburg.de/~tthuem/

Figure B.1 presents the dependencies between all files of the FJ formalization on the left side and all files of the CFJ formalization on the right side. For clarity, transitive dependencies are printed in light gray.

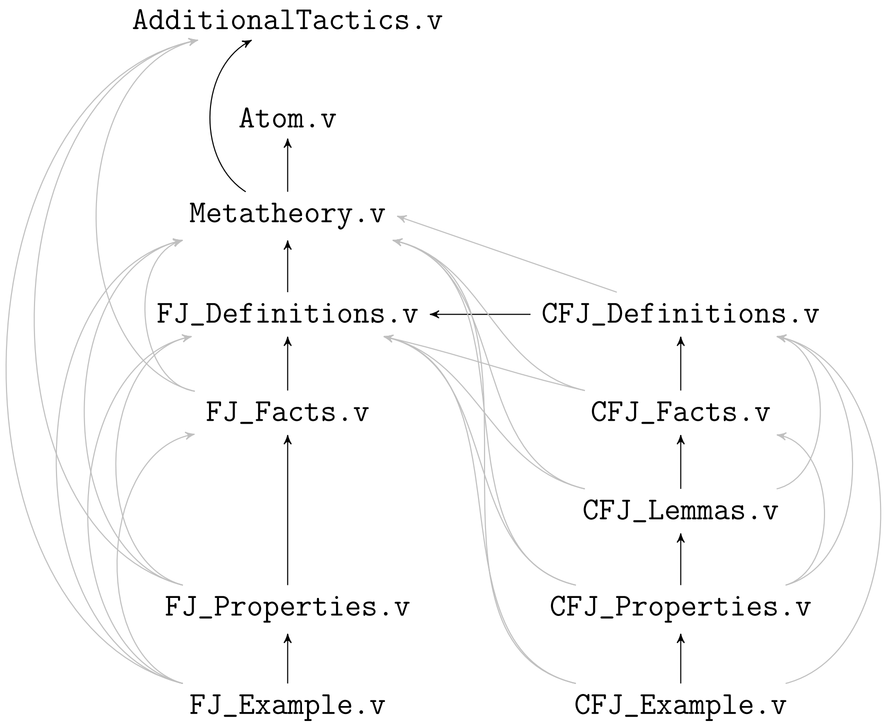


Figure B.1: Dependencies Between the FJ and the CFJ Formalization

An order for compilation can easily be derived. One could compile all FJ files top-down and then all CFJ files top-down. The overview also shows that we used a partition for CFJ similar to that of FJ. The additional file with lemmas contains all proves dealing with reachability checks, which we we do not have in FJ.

B.2 Annotation Lookup Functions

Annotation lookup at the class table was presented in Section 4.2.1. We present the definitions here, because they are highly repetitive (see Listing B.1). The functions lookup the annotation of a particular code fragment at the class table. There is a function for every kind of code fragment.

```
Definition AT (C:cname) : ann :=
    match (get C aCT) with
    | None ⇒ if C == Object then True else False
    | Some (a, _, _, _, _, _, _) ⇒ a
    end.
Definition ATp (C:cname) (f:fname) : ann :=
    match (get C aCT) with
    | None ⇒ False
    | Some (_, _, gs, _, _, _, _) ⇒
```

```
        match (get f gs) with
        | None ⇒ False
        | Some (a, _) ⇒ a
        end
    end.
Definition ATs (C:cname) (f:fname) : ann :=
    match (get C aCT) with
    | None ⇒ False
    | Some (_, _, _, hs, _, _, _) ⇒
        match (get f hs) with
        | None ⇒ False
        | Some (a, _) ⇒ a
        end
    end.
Definition ATa (C:cname) (f:fname) : ann :=
    match (get C aCT) with
    | None ⇒ False
    | Some (_, _, _, _, ts, _, _) ⇒
        match (get f ts) with
        | None ⇒ False
        | Some (a, _) ⇒ a
        end
    end.
Definition ATf (C:cname) (f:fname) : ann :=
    match (get C aCT) with
    | None ⇒ False
    | Some (_, _, _, _, _, fs, _) ⇒
        match (get f fs) with
        | None ⇒ False
        | Some (a, _) ⇒ a
        end
    end.
Definition ATm (C:cname) (m:mname) : ann :=
    match (get C aCT) with
    | None ⇒ False
    | Some (_, _, _, _, _, _, ms) ⇒
        match (get m ms) with
        | None ⇒ False
        | Some (a, _) ⇒ a
        end
    end.
```

Listing B.1: Annotation Lookup Functions for CFJ

B.3 Theorems, Lemmas, and Facts

In Section 5.1.2, we omitted the formal definitions of theorems, lemmas, and facts used to prove type soundness of CFJ. In Listing B.2, we present these definitions without the proofs, because they are very long and can be found on-line. The order is identical to that in our formalization.

```
Fact no_binds_env :
  forall (x:var) (E:a_env) (E':env),
    no_binds x E →
    variant_env E E' →
    no_binds x E'.

Fact ok_env :
  forall (E:a_env) (E':env),
    ok E →
    variant_env E E' →
    ok E'.

Fact ok_env_env :
  forall (E F:a_env) (E' F':env),
    ok (E ++ F) →
    variant_env E E' →
    variant_env F F' →
    ok (E' ++ F').

Fact no_binds_ms :
  forall (C:cname) (m:mname) (ms:a_mths) (ms':mths),
    no_binds m ms →
    variant_methods C ms ms' →
    no_binds m ms'.

Fact no_binds_ct :
  forall (C:cname) (ct:a_ctable) (ct':ctable),
    no_binds C ct →
    variant_ct ct ct' →
    no_binds C ct'.

Fact vtu_mut :
  ( forall (t:a_exp) (t':exp),
    variant_term t t' →
    forall (t'':exp), variant_term t t'' → t' = t'' ) ∧
  ( forall (ts:list (ann*a_exp)) (ts':list exp),
    variant_terms ts ts' →
    forall (ts'':list exp), variant_terms ts ts'' → ts' = ts'' ).

Fact variant_term_unique :
  forall (t:a_exp) (t' t'':exp),
    variant_term t t' →
    variant_term t t'' →
    t' = t''.

Fact variant_env_unique :
  forall (fs:a_flds) (fs' fs'':flds),
    variant_env fs fs' →
    variant_env fs fs'' →
    fs' = fs''.

Definition vtt_trick
  (REC : forall (t:a_exp), exists t':exp, variant_term t t') :
  forall (ts:list (ann*a_exp)),
    exists ts':list exp, variant_terms ts ts'.

```

```
Fact variant_term_total :
  forall (t:a_exp), exists t':exp, variant_term t t'.

Fact variant_env_total :
  forall (fs:a_flds),
    exists fs':flds, variant_env fs fs'.

Lemma class_superclass :
  forall (C D:cname) (gs hs ts fs:a_flds) (ms:a_mths),
    binds C (AT C,D,gs,hs,ts,fs,ms) aCT →
    (AT C → AT D).

Lemma field_class :
  forall (C:cname) (f:fname), ATf C f → AT C.

Lemma variable_present :
  forall (x:var) (a:ann) (t:typ) (E:a_env) (E':env),
    binds x (a,t) E →
    a →
    variant_env E E' →
    binds x t E'.

Lemma class_present :
  forall (C D:cname) (gs hs ts fs:a_flds) (ms:a_mths),
    AT C →
    binds C (AT C,D,gs,hs,ts,fs,ms) aCT →
    exists fs':flds, exists ms':mths,
      variant_env fs fs' ∧
      variant_methods C ms ms' ∧
      binds C (D,fs',ms') CT.

Lemma class_present_reverse :
  forall (C D:cname) (fs':flds) (ms':mths),
    binds C (D,fs',ms') CT →
    exists gs:a_flds, exists hs:a_flds, exists ts:a_flds,
    exists fs:a_flds, exists ms:a_mths,
      variant_env fs fs' ∧
      variant_methods C ms ms' ∧
      binds C (AT C,D,gs,hs,ts,fs,ms) aCT.

Lemma fields_present :
  forall (C:cname) (fs:a_flds),
    AT C →
    a_fields C fs →
    exists fs':flds, variant_env fs fs' ∧ fields C fs'.

Lemma field_present :
  forall (C D:cname) (f:fname),
    ATf C f →
    a_field C f (ATf C f) D →
    field C f D.

Lemma method_present :
  forall (a:ann) (C C0:cname) (m:mname) (Dys:a_env) (t:a_exp)
```

```
    (Dys':env) (t':exp),
    a →
    AT C →
    a_method C m a (C0,Dys,t) →
    variant_env Dys Dys' →
    variant_term t t' →
    method C m (C0,Dys',t').

Lemma method_present' :
  forall (a:ann) (C C0:cname) (m:mname) (Dys:a_env) (t:a_exp),
    a →
    AT C →
    a_method C m a (C0,Dys,t) →
    exists Dys':env, exists t':exp,
      variant_env Dys Dys' ∧
      variant_term t t' ∧
      method C m (C0,Dys',t').

Lemma method_present_reverse :
  forall (a:ann) (C C0:cname) (m:mname) (Dys':env) (t':exp),
    method C m (C0,Dys',t') →
    exists Dys:a_env, exists t:a_exp,
      variant_env Dys Dys' ∧
      variant_term t t' ∧
      a_method C m a (C0,Dys,t).

Lemma type_present :
  forall (a:ann) (E:a_env) (t:a_exp) (C:typ),
    a_typing a E t C → (a → AT C).

Lemma subtype_present :
  forall (C D:typ),
    AT C →
    a_sub C D →
    sub C D.

Theorem gpt_mutual_typing :
    ( forall (t:a_exp) (t':exp),
      variant_term t t' →
      ( forall (a:ann) (E:a_env) (C:cname) (E':env),
        a_typing a E t C →
        a →
        variant_env E E' →
        typing E' t' C ) ∧
      ( forall (a:ann) (E:a_env) (C:cname) (E':env),
        a_wide_typing a E t C →
        a →
        variant_env E E' →
        wide_typing E' t' C ) ) ∧
    ( forall (ts:list (ann*a_exp)) (ts':list exp),
      variant_terms ts ts' →
      forall (a:ann) (E:a_env) (Cxs:a_env) (E':env) (Cxs':env),
        a_wide_typings a E ts (imgs Cxs) →
        a →
        variant_env E E' →
        variant_terms ts ts' →
        variant_env Cxs Cxs' →
        wide_typings E' ts' (imgs Cxs') ).

Theorem gpt_wide_typing :
  forall (a:ann) (E:a_env) (t:a_exp) (C:cname) (E':env) (t':exp),
    a_wide_typing a E t C →
    a →
    variant_env E E' →
    variant_term t t' →
    wide_typing E' t' C.

Theorem gpt_typing :
  forall (a:ann) (E:a_env) (t:a_exp) (C:cname) (E':env) (t':exp),
    a_typing a E t C →
    a →
    variant_env E E' →
    variant_term t t' →
    typing E' t' C.

Theorem gpt_ok_meth :
  forall (C D C0:cname) (m:mname) (Cfs:a_env) (Cfs':env) (t:a_exp)
    (t':exp),
    a_ok_meth C D m C0 Cfs t →
    AT C →
    ATm C m →
    variant_env Cfs Cfs' →
    variant_term t t' →
    ok_meth C D m C0 Cfs' t'.

Theorem gpt_ok_class :
  forall (C D:cname) (fs:a_flds) (fs':flds) (ms:a_mths) (ms':mths),
    a_ok_class C D fs ms →
    AT C →
    variant_env fs fs' →
    variant_methods C ms ms' →
    ok_class C D fs' ms'.

Theorem gpt_ok_ctable :
  forall (ct:a_ctable) (ct':ctable),
    a_ok_ctable ct →
    variant_ct ct ct' →
    ok_ctable ct'.

Theorem generation_preserves_typing :
  forall (t:a_exp) (t':exp),
    FM →
    CFJ_product_line t →
    variant_ct aCT CT →
    variant_term t t' →
    FJ_program t'.
```

Listing B.2: Theorems, Lemmas, and Facts to Prove Type Soundness for CFJ

Bibliography

[AG01] Michalis Anastasopoules and Critina Gacek. Implementing Product Line Variabilities. *Software Engineering Notes*, 26(3):109–117, 2001. 5

[AKGL09] Sven Apel, Christian Kästner, Armin Größlinger, and Christian Lengauer. Type-Safe Feature-Oriented Product Lines. Technical Report MIP-0909, Department of Informatics and Mathematics, University of Passau, June 2009. 3, 4, 11, 84

[Bak95] Brenda S. Baker. On Finding Duplication and Near-Duplication in Large Software Systems. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 86–95, Washington, DC, USA, July 1995. IEEE Computer Society. 1

[Bat05] Don Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the International Software Product Line Conference (SPLC)*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer, 2005. 7

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions.* Springer, Germany, 2004. 13, 14, 15, 45, 67

[Beu03] Danilo Beuche. *Composition and Construction of Embedded Software Families.* PhD thesis, University of Magdeburg, Germany, 2003. 2

[Car97] Luca Cardelli. Type Systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 103, pages 2208–2236. CRC Press, 1997. 10, 11

[CDT09a] The Coq Development Team. The Coq Proof Assistant. Website, September 2009. Available online at http://coq.inria.fr/; visited on November 9th, 2009. 13

[CDT09b] The Coq Development Team. *The Coq Proof Assistant Reference Manual.* LogiCal Project, 2009. Version 8.2pl1. 14, 15, 16

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications.* ACM Press/Addison-Wesley, 2000. 1, 6, 7

[CP06] Krzysztof Czarnecki and Krzysztof Pietroszek. Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 211–220, New York, NY, USA, 2006. ACM. 3, 11, 83

[DCB09] Benjamin Delaware, William Cook, and Don Batory. A Machine-Checked Model of Safe Composition. In *Proceedings of the Workshop on Foundations of Aspect-Oriented Languages (FOAL)*, pages 31–35, New York, NY, USA, 2009. ACM. 11, 84

[Fra09a] Bruno De Fraine. Formalization and Type-Soundness Proof for Featherweight Java in Coq. Website, August 2009. Available online at http://soft.vub.ac.be/~bdefrain/featherj/featherj-20090813.zip; visited on October 21st, 2009. xiii, xiv, 40, 42, 91, 92

[Fra09b] Bruno De Fraine. *Language Facilities for the Deployment of Reusable Aspects*. PhD thesis, University of Brussel, Belgium, June 2009. 39, 40

[GFdA98] Martin L. Griss, John Favaro, and Massimo d' Alessandro. Integrating Feature Modeling with the RSEB. In *Proceedings of the International Conference on Software Reuse (ICSR)*, pages 76–85, 1998. 6

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. 1

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley, Amsterdam, June 2005. 17

[Gon04] Georges Gonthier. A Computer-Checked Proof of the Four Colour Theorem. Technical report, Microsoft Research Cambridge, 2004. 4, 13

[HZS07] Shan Shan Huang, David Zook, and Yannis Smaragdakis. cJ: Enhancing Java with Safe Type Conditions. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 185–198, New York, NY, USA, March 2007. ACM. 83

[IPW99] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 132–146. ACM, 1999. 17, 18, 60

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, 2001. 17, 19, 20, 41

[JDHW09] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do Code Clones Matter? In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2009. 1

[JKB08] Mikoláš Janota, Joseph Kiniry, and Goetz Botterweck. Formal Methods in Software Product Lines: Concepts, Survey, and Guidelines. Technical Report Lero-TR-SPL-2008-02, Lero, University of Limerick, May 2008. ix, 2

[Joh93] J. Howard Johnson. Identifying Redundancy in Source Code using Fingerprints. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 171–183. IBM Press, October 1993. 1

[KA08] Christian Kästner and Sven Apel. Type-Checking Software Product Lines - A Formal Approach. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 258–267. IEEE Computer Society, 2008. ix, 3, 4, 11, 19, 23, 24, 27, 28, 29, 42, 59, 60, 78, 87

[KAK08] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in Software Product Lines. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 311–320, New York, NY, USA, May 2008. ACM. ix, 1, 9

[KAK09] Christian Kästner, Sven Apel, and Martin Kuhlemann. A Model of Refactoring Physically and Virtually Separated Features. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 157–166. ACM, October 2009. 88

[KAS] Christian Kästner, Sven Apel, and Gunter Saake. Type Checking Software Product Lines - A Formal Approach for Annotation-Based Implementations. Unpublished Manuscript. Submitted on August 10th, 2009. ix, 28, 30, 42, 59, 60, 63, 64, 83, 85

[KCH$^+$90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, 1990. 6

[KG06] Cory J. Kapser and Michael W. Godfrey. Supporting the Analysis of Clones in Software Systems: A Case Study. *Journal of Software Maintenance and Evolution*, 18(2):61–82, 2006. 1

[KKL$^+$98] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Gerard Jounghyun Kim, and Euiseob Shin. FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. *Annals of Software Engineering*, 5(1):143–168, January 1998. 5

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997. 1

[KV06] Florian Kammüller and Matthias Vösgen. Towards Type Safety of Aspect-Oriented Languages. In *Proceedings of the Workshop on Foundations of Aspect-Oriented Languages (FOAL)*, Bonn, Germany, March 2006. ACM. 84

[MLM96] Jean Mayrand, Claude Leblanc, and Ettore Merlo. Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 244–253, Washington, DC, USA, November 1996. IEEE Computer Society. 1

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering : Foundations, Principles and Techniques*. Springer, September 2005. 1, 2, 5

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, Massachusetts, USA, 2002. 2, 10, 12, 13, 17, 21, 41, 49

[Pre97] Christian Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443. Springer, 1997. 1

[Ros09] Malte Rosenthal. Alternative Features in Colored Featherweight Java. Diplomarbeit, University of Passau, Germany, July 2009. 83

[SB00] Mikael Svahnberg and Jan Bosch. Issues Concerning Variability in Software Product Lines. In *Proceedings of the International Workshop on Software Architectures for Product Families (IW-SAPF)*, volume 1951 of *Lecture Notes in Computer Science*, pages 146–157, London, UK, 2000. Springer. 5

[SH04] Mark Staples and Derrick Hill. Experiences Adopting Software Product Line Development without a Product Line Architecture. In *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC)*, pages 176–183, Los Alamitos, CA, USA, 2004. IEEE Computer Society. 1

[SNO$^+$07] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Tom Ridge, Susmit Sarkar, and Rok Strnisa. Ott: Effective Tool Support for the Working Semanticist. In *Proceedings of the International Conference on Functional Programming (ICFP)*, pages 1–12. ACM, 2007. 84

[SSP07] Rok Strniša, Peter Sewell, and Matthew Parkinson. The Java Module System: Core Design and Semantic Definition. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 499–514, New York, NY, USA, October 2007. ACM. 84

[TBKC07] Sahil Thaker, Don Batory, David Kitchin, and William Cook. Safe Composition of Product Lines. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 95–104. ACM, 2007. 3, 11, 84

[Wei05] Stephanie Weirich. Formalization and Type-Soundness Proof for Featherweight Java in Coq. Website, August 2005. Available online at http://www.cis.upenn.edu/~plclub/wiki-static/fj-coq.tar.gz; visited on October 20th, 2009. 40

[WF94] Andrew K. Wright and Matthias Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, 115(1):38–94, November 1994. 11, 12, 13

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den 15. Januar 2010