Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Mechanization and Overhaul of Feature Featherweight Java

Pedro da C. Abreu Jr.

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Dr. Rodrigo Bonifácio de Almeida

Brasília
2017

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharelado em Ciência da Computação

Coordenador: Prof. Dr. Rodrigo Bonifácio de Almeida

Banca examinadora composta por:

Prof. Dr. Rodrigo Bonifácio de Almeida (Orientador) — CIC/UnB
Prof. Dr. Rodrigo B. de Almeida — CIC/UnB
Prof. Dr. Flávio L. C Moura — CIC/UnB

CIP — Catalogação Internacional na Publicação

Abreu Jr., Pedro da C..

Mechanization and Overhaul of Feature Featherweight Java / Pedro da C. Abreu Jr.. Brasília : UnB, 2017.
55 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2017.

1. Verificação Formal, 2. FFJ, 3. Coq

CDU 004.4

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Mechanization and Overhaul of Feature Featherweight Java

Pedro da C. Abreu Jr.

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. Dr. Rodrigo Bonifácio de Almeida (Orientador)
CIC/UnB

Prof. Dr. Rodrigo B. de Almeida	Prof. Dr. Flávio L. C Moura
CIC/UnB	CIC/UnB

Prof. Dr. Rodrigo Bonifácio de Almeida
Coordenador do Bacharelado em Ciência da Computação

Brasília, 6 de agosto de 2017

Dedicatória

Dedico este trabalho ao meu falecido pai Pedro da Costa Abreu, o qual mesmo ausente, sua vida e jornada é uma perene inspiração à minha vida.

E também à minha mãe Rosa Delmira de Souza Abreu, cuja resiliência, bravura e dedicação a iguala aos grandes heróis da antiguidade.

Agradecimentos

Abstract

Specifying a language using an Interactive Theorem Prover (ITP) is seldom faithful to its original pen-and-paper specification. However, the process of mechanizing a language and type safety proofs might also unearth insights for improving the original specification. In this paper, we detail some design decisions related to our process of first specifying *Featherweight Java (FJ)* in `Coq` and thus evolving such a specification to prove the type system properties of an overhaul version *Feature Featherweight Java (FFJ)*—a core-calculus for a family of languages that address variability management in highly configurable systems, such as software product lines (SPLs); which we name as *Overhaul Feature Featherweight Java (FFJ⋆)*. Indeed, FFJ⋆ is the first mechanization of FFJ, and as such it might also help researchers to derive proofs about software product line refinements without considering several assumptions about the underlining SPL assets. We believe that the whole process led us to a clearer, unambiguous, and equivalent syntax and semantics of FFJ, while keeping the proofs as well as our FJ extensions as simple as possible.

Keywords: Language Design, Language Semantics, Java, FOP, Coq

Contents

Glossary

CA Core Assets. 4

CK Configuration Knowledge. 4

FFJ Feature Featherweight Java. iii, iv, vii, 1, 2, 7–12

FFJ$\star$ Overhaul Feature Featherweight Java. iii, iv, vii, 2, 7–12, 15

FJ Featherweight Java. iii, 1, 2, 7–10, 12

FM Feature Model. 4

FOP Feature Oriented Programming. iv, 1–3, 5, 7, 9

ITP Interactive Theorem Prover. iii

SPL Software Product Line. 1, 3, 4

List of Figures

List of Tables

Chapter 1

Introduction

FOP [21] is a design methodology and tools for program synthesis [9]. It aims at the modularization of software systems in terms of features. A *feature* implements a stakeholder's requirement and is typically an increment in program functionality. When added to a software system, a feature introduce new structures, such as classes and methods, and refines existing ones, such as extending methods bodies.

There are several FOP languages and tools that provides varying mechanisms that support the specification and composition of features properly, such as AHEAD [6], FSTComposer [4], FeatureC++ [3], and more recently Delta-Oriented Programming [23]. FOP has mostly been used to develop *product-lines* in disparate domains, including compilers for extensible Java dialects [7], fire support simulators for U.S. Army [10], high-performance network [11], and program verification tools [17].

Since FOP provides such a powerful mechanism to deal with software variance, Software Product Lines (SPLs) have made great use of its concepts. Just like the Ford's product lines in the domain of automobile aims to provide customized automobiles at reasonible price by providing the means for cheap customization. SPL aims to provide customized software at a reasonable price by providing the means for cheap customization using FOP concepts. At the section **??** we shall use SPL to explain how the main FOP concepts.

Several attempts to formalize the type system of FOP languages have been made. For instance, *FFJ* [2] is a proposed type system for FOP languages and tools, which is developed on top of *FJ* [15] to provide a simple syntax and semantics conforming with common FOP languages, incorporating constructs for feature composition.

Nevertheless, very few effort was made to mechanize a FOP language. In matter of fact, only one FOP language was implemented with a proof assistant to date, it is known as LFJ [13] Mechanizing a language is interesting because it makes the proofs even more reliable than peer review. Take for an instance the Perko Pairs [19]. They were listed by C.N Little as different knots in 1885, and only almost a century latter, in 1974 Ken Perko discovered [22] them to actually be the same knot!

The idea behind mechanization is to check these proofs with the aid of a computer, reducing significantly the risk of errors, while taking full use of automation for the tedious or straightforward steps of the proof. As the system may grow, the mechanization makes the proof a lot more reliable.

And also, mechanized proofs leads a better organization when the system grows larger. Better organization of the proof process allows to build teams for these proofs. This allows to mechanize correctness properties of big, real world implementations, e.g. compilers [18], file systems [1, 5] and languages [14, 16].

The process of scrutinizing FFJ and defining *unambiguously* its semantics in `Coq` lead us to some language specification and implementation improvements. The biggest change was to review and simplify the lookup functions of the refinement table. The implementations of FJ and FFJ can both be found at github[1][2]. Henceforth, we refer this proposed calculus as FFJ$\star$ to distinguish it when comparing our implementation to the original FFJ design. Altogether the improvements proposed in FFJ$\star$ makes the transition more natural between FJ and FFJ, simplifying the auxiliary functions used in the language specification as well as the type safety proofs and lemmas. This allows defining FFJ$\star$ with incremental changes to FJ syntax and semantics, and consequently, incremental changes to proofs, leading to a clearer and simpler specification of *FFJ*.

The main goal of this paper is to present a novel mechanization of a FOP language. In particular, the mechanization of FFJ$\star$. Hence we can summarize the main contribution of this paper as follows:

1. The first mechanization of the FFJ type system
2. An improved specification of FFJ, which may help other researchers to reason about software product lines properties.
3. A report about the benefits of using a proof assistant to revamp an existing specification of a non-mechanized language type system.

This paper is organized as follows: in the Section **??** we give a brief introduction to `Coq` Section 2.1 briefly introduces software product lines, FOP and FFJ$\star$, Section 3 gives a brief introduction of FFJ$\star$ and explains the main differences with FJ Section 4 formally describes our revamped FFJ and states the lemmas needed to preserve FJ increment to FFJ type safety, Section 6 discuss the implications of these results to the product line research, Section 5 discuss related works and Section 7 is the conclusion and shows possible future works.

[1] https://github.com/hephaestus-pl/coqfj
[2] https://github.com/hephaestus-pl/coqffj

Chapter 2

Theory Fundamentals

Under the context of software engineering, a lot of effort have been spent in the scope of *reuse*. However most of the effort have been made code reuse, and not that much into software reuse as a whole.

In this chapter we provide the necessary definitions to understand FOP, and how this paradigm copes with software reuse. To simplify the understanding we will take the examples under software product lines. This will also make clear how mechanizing a FOP language shall benefit real world applications.

2.1 Feature Oriented Programming

Feature-oriented programming (FOP) is a development approach that supports the *stepwise refinement strategy* for software constructions [8]. Using FOP, a system is typically decomposed in (somewhat new) modular unities (named features) that resemble mixing layers [12], and thus are orthogonal to the typical object-oriented decomposition in terms of class hierarchies. Successful FOP usage scenarios have been reported in the literature for the domains of highly configurable systems and software product lines []. FOP has been implemented using both programming language extensions and tooling support, such as Java AHEAD Tool Suite [6] and FeatureC++ [3].

2.2 Software Product Line

In the 70's the concept of software families was introduced by Parnas [20]. It's main goal was to enhance the versatility to the development of the artefact's non-functional requirements. Upon this, the concept of SPL was formalized with the purpose of projecting several softwares with similar characteristics under a single domain.

Sommerville [24] defines SPL as one of the most effective approaches to reuse. And defines it as a set of applications with a common architecture and shared components.

As the name suggest, SPL idea comes from Ford's product lines. With a product line it is possible to build several different specializations of the same product, while improving efficiency and reducing cost. This allow mass individualization of the products, i.e. even though the industry is still delivering products in mass scale, it still provides somewhat individualized products for different kinds of clients. The analogue still holds for SPL, it

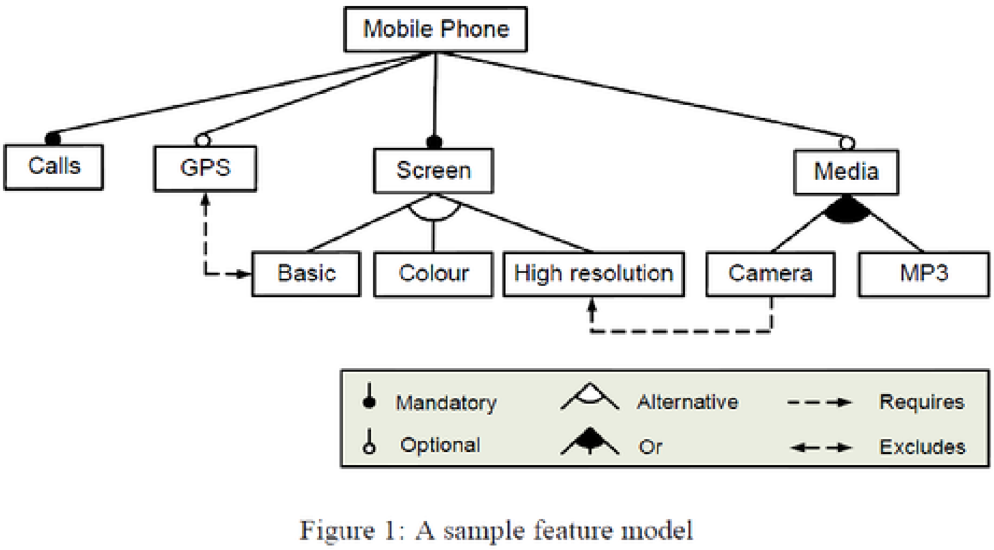

Figure 2.1: Cellphone OS feature model

proposes a framework which allows to build several different specializations of the software, while reducing delivery time which by its turn reduces cost.

Take for example a SPL illustrated in 2.1 for a mobile phone operational system. Every cellphone must be able to make calls and receive calls and having a screen. But there are optional features, such as having a GPS, being able to reproduce media, etc.

Formally a SPL is defined by a triple: the Feature Model (FM), Core Assets (CA), and the Configuration Knowledge (CK).

The FM is the set of all features. They may be: *obligatory*, *optional*, *alternative*, *and* and *or*.

The CA is everything useful in the process of development, such as documentation, test cases, code, and so on.

The CK is a mapping between features to assets, driving product generation.

With that in hand it is possible to compose the assets in order to provide a new product. However, it is not guaranteed that this composition process is safe, i.e. that every asset selected copes well with each other. This leads us to the safe composition problem.

In order to tackle this safe composition problem, one could manually inspect FM, CK and implementation to understand the dependencies between assets for all products. However, since SPLs can quickly scale to hundreds of products, this is often impractical.

Another approach would be to generate every single product, compile and test them. While this is an useful and safe approach, it does not scale given the exponential factor in every feature introduction.

This is where formal methods shines. With formal methods it is possible to study how features interact with each other, postulate properties and provide safety theorems for SPLs without having to generate every single product.

2.3 A Running Example: The Expression Product Line in FOP

In this section we illustrate the use of FOP through an AHEAD implementation of a slight adaptation of the Expression Product Line (EPL) []—Figure 2.3 shows the EPL feature model. Regarding our design decisions, in this case we implemented the mandatory features using a BASE AHEAD package (Figure 2.3), which declares a class hierarchy involving an interface (`Expression`) and several classes (`Value`, `BinaryExpression`, `AddExpression`, and `SubExpression`), and one AHEAD package for each non-mandatory feature (see Figure 2.4). Note that an AHEAD package contains either (a) plain Java entities (class or interface) declarations or (b) Java entities refinements. A refinement might override methods declared in other packages or introduce new attributes or methods in existing classes or interfaces. In this simple example, we do not implement any method overriding through class refinements—the refinements only introduce new elements to the BASE AHEAD package of Figure 2.3.

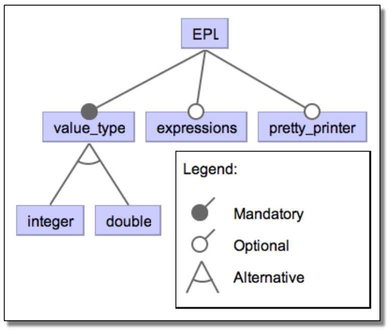

Figure 2.2: EPL feature model

```
interface Expression {
  public Value eval();
}

class Value implements Expression{
  public Value eval() { return this; }
}

abstract class BinaryExpression implements Expression {
  Expression lhs;
  Expression rhs;
}

class AddExpression extends BinaryExpression {
  public Value eval() { return lhs.eval() + rhs.eval(); }
}

class SubExpression extends BinaryExpression {
  public Value eval() { return lhs.eval() - rhs.eval(); }
}
```

Figure 2.3: The BASE package of the Expression Product Line

The details of the EPL AHEAD non-mandatory feature packages are as follows.

- Features `integer` and double refine the `Value` class of the BASE package by introducing a new attribute named `value`, either with type `int` or `double`. According to the EPL feature model, only one of these features might be selected for a given product.
- The `expressions` feature introduces two new expressions to those declared in the BASE package, one for multiplication and another for division. This particular feature does not refine existing classes, only introduces new ones.

- The `pretty_printer` feature introduces the support for *pretty printing* expressions. It refines the `Expression` interface and the `BinaryExpression` and `Value` classes, introducing a new method `print()` and also a new attribute (`operator`) for the `BinaryExpression` class.

```
refines Value {
  int value;
  int getValue() { return value; }
}
```

(a) integer feature package

```
refines Value {
  double value;
  double getValue() { return value; }
}
```

(b) double feature package

```
class TimesExpression extends BinaryExpression {
  public Value eval() { return lhs.eval() * rhs.eval(); }
}

class DivExpression extends BinaryExpression {
  public Value eval() { return lhs.eval() / rhs.eval(); }
}
```

(c) expressions feature package

```
refines Expression {
  public String print();
}

refines BinaryExpression {
  String operator;
  public String print() { return lhs.print() + operator + rhs.print(); }
}

refines Value {
  public String print() { return “” + value; }
}
```

(d) prety_printer feature package

Figure 2.4: Non-mandatory feature implementations of the Expression Product Line

In the case we generate a product with a feature selection consisting of `EPL`, `value_type(double)`, and `pretty_printer`, we will get a product as shown in Figure ??.

2.4 Software Product line formal verification

Chapter 3

Overview of FFJ and FFJ$\star$

FFJ is a core calculus for FOP, which was built upon an extension of FJ—a minimal subset of Java. In FFJ, classes can be added and modified by the introduction of a new feature, that is, an existing class can be extended by a class refinement. A class refinement is declared like a conventional class, though preceded by the keyword `refines`. For example, `refines class C {...}` refers to a class refinement that *refines* the class `C`. The same can be achieved for method introduction and modification. Methods refinement, however, override a previous definition of the corresponding method.

To fully mechanize FFJ, we had to disambiguate and enhance the language to some extent that it deserves the attention of formally documenting these changes. Even though these changes are significant, as discussed in Section 4, the philosophy of FFJ, FOP, and Stepwise Refinement are maintained. Due to the lack of space we will not provide neither the formal definition of FJ nor FFJ, but we refer the reader the original formalization of FJ [15]. and FFJ [2] In FFJ, as well as in FFJ$\star$, classes can be added and modified by the introduction of a new feature (as discussed in the previous section). An existing class can be extended by a class refinement. A class refinement is declared like a class but preceded by the keyword `refines`. For example, `refines class C@feat {...}` refers to a class refinement that refines the class `C`. The same can be achieved for method introduction and modification. Methods refinement, however, will override the previous definition.

A syntactical difference between FFJ and FFJ$\star$ is that, in FFJ$\star$, the feature notion appears in the abstract syntax tree (AST) of the language. While the designers of FFJ argue that the programmer does not have to explicitly state which feature a class or method belongs to, we favored the approach of stating the feature in the name of every refinement. This greatly simplifies the structure of the formalism of the language and can be seen as an information gathered by the parser to build the AST, and thus the actual code expressed using the concrete syntax of this language might not have these annotations.

In addition, an FFJ$\star$ program has a table with every class declaration (CT) and another table with every class refinement (RT). We make this distinction to simplify the extension from FJ in `Coq`, since with this decision we eliminate the need to match whether a class in the table is a refination or a declaration. From this RT we can retrieve the composition order of the refinements and build the refinement chain of the program, which is used to check if features were composed correctly and does not references features that have not been introduced yet. Notice that we redefine the denotation of RT from

FFJ. In the original version, it was used to retrieve the refinement name given a refinement declaration. This is no longer necessary in FFJ$\star$, since that information is already encoded in the syntax.

Finally, in the original definition of FFJ, the lookup functions are somewhat circumvoluted. Accordingly, we propose a very different approach for them, with the aim as been not only as formal and simple as possible, but also easy to evolve from our mechanized version of FJ. To this end, we eliminate the need for reverse field lookup, reverse method lookup, and the refinement relation. A formal description with all these changes is given in Section 4.2. Note that, we were only able to conceive these improvements while formalizing FFJ$\star$ in `Coq`.

Chapter 4

Overhaul Feature Featherweight Java

4.1 Syntax

The Syntax of FFJ is a straightfoward FOP extension of FJ. Due to the lack of space we do not present the formal definition FJ nor FFJ, but instead we follow the same scheme of the FFJ original definition in [2] and present the modified rules from FFJ to FFJ$\star$ highlighted with shaded yellow boxes and new rules highlighted by shaded purple boxes. Also notice that the sucessor and the refinement relations were simply dropped for being unecessary by now.

`R ::=` `C@feat`	*refinement names:*
`CD ::=` `class C extends D {C̄ f̄; K M̄}`	*class declarations:*
`CR ::=` `refines class R {C̄ f̄; KD M̄ MR̄}`	*class refinements:*
`K ::=` `C(C̄ f̄){super(f̄); this.f̄=f̄;}`	*constructor declarations:*
`KD ::=` `refines C(Ē h̄, C̄ f̄){original(f̄); this.f̄=f̄;}`	*constructor refinements:*
`M ::=` `C m (C̄ x̄) {return e;}`	*method declarations:*
`MR ::=` `refines C m (R̄ x̄) {return e;}`	*method refinements:*
`e ::=`	*expressions:*
`x`	*variable*
`e.f`	*field access*
`e.m(ē)`	*method invocation*
`new C(ē)`	*object creation*
`(C)e`	*cast*
`v ::=`	*values:*
`new C(ē)`	*object creation*

Table 4.1: FFJ Syntax

The syntax of FFJ$\star$ constructs is given at Table 4.1. The metavariables `A`, `B`, `C`, `D` and `E` ranges over class names, `f` and `g` range over field names; `m` ranges method name; `x` ranges over variable, `v` ranges over values, `feat` ranges over feature names. We assume that the set of variables includes the special variable `this`, which cannot be used as the name of an argument of a method.

We write $\bar{\texttt{f}}$ as a shorthand for a possible empty sequence $\texttt{f}_1, \ldots, \texttt{f}_n$ and similarly for $\bar{\texttt{C}}$, $\bar{\texttt{x}}$, $\bar{\texttt{e}}$, etc. We abbreviate the operations on pairs of sequences "$\bar{\texttt{C}}\ \bar{\texttt{f}}$" for "$\texttt{C}_1\ \texttt{f}_1, \ldots, \texttt{C}_n\ \texttt{f}_n$" and "$\texttt{this}.\bar{\texttt{f}}=\bar{\texttt{f}};$" as a shorthand for "$\texttt{this}.\bar{\texttt{f}}_1=\bar{\texttt{f}}_1;\ \ldots,\ \texttt{this}.\bar{\texttt{f}}_n=\bar{\texttt{f}}_n;$". We write empty sequence as $\bullet$.

A class declaration `class C extends D {`$\bar{\texttt{C}}\ \bar{\texttt{f}}$`; K `$\bar{\texttt{M}}$`}` introduces a class `C` with superclass `D`. This class has fields $\bar{\texttt{f}}$ of type $\bar{\texttt{C}}$, a constructor `K` and methods $\bar{\texttt{M}}$. The fields of the class `C` is $\bar{\texttt{f}}$ added to the fields of its superclass `D`, all of them must have distinct names. Methods, on the other hand, may override another superclass method with the same name. Method override in both FFJ and FFJ$\star$ is basically method rewriting. Methods are uniquely identified by its name, i.e. overloading is not supported.

A class refinement `refines class C@feat {`$\bar{\texttt{C}}\ \bar{\texttt{f}}$`; KD `$\bar{\texttt{M}}\ \overline{\texttt{MR}}$`}` introduces a refinement of the class `C` and belongs to the feature `feat`. This refinement contains the fields $\bar{\texttt{f}}$ of type $\bar{\texttt{C}}$, a constructor refinement `KR`, methods declarations $\bar{\texttt{M}}$ and method refinements $\overline{\texttt{MR}}$. Like class declarations, the fields of a class refinement `R` are added to the fields of its predecessor, which is explained in more detail in Section 4.2.

Constructor declaration `C(`$\bar{\texttt{C}}\ \bar{\texttt{f}}$`){super(`$\bar{\texttt{f}}$`); this.`$\bar{\texttt{f}}$`=`$\bar{\texttt{f}}$`;}` and a constructor refinement `refines C(`$\bar{\texttt{E}}\ \bar{\texttt{h}}$`, `$\bar{\texttt{C}}\ \bar{\texttt{f}}$`) {original(`$\bar{\texttt{f}}$`); this.`$\bar{\texttt{f}}$`=`$\bar{\texttt{f}}$`;}` introduce a constructor for the class `C` with fields $\bar{\texttt{f}}$ of type $\bar{\texttt{C}}$. The constructor declaration body is simply a list of assignment of the arguments with its correspondent field preceded by calling its superclass constructor with the correspondent arguments. The constructor refinement only differs from constructor declaration that instead of calling the superclass constructor it will call its predecessor constructor (denoted by `original`).

Method declaration `C m (`$\bar{\texttt{C}}\ \bar{\texttt{x}}$`) {return e;}` and method refinement `refines C m (`$\bar{\texttt{C}}\ \bar{\texttt{x}}$`) {return e;}` introduce a method `m` of return type `C` with arguments $\bar{\texttt{C}}\ \bar{\texttt{x}}$ and body `e`. Method declarations can only appear inside a class declarations or refinement, whereas method refinement should only appear inside a class refinement. There is such a distinction between method declaration and method refinement for allowing the type checker to recognize the difference between method refinement and inadvertent overriding/replacement.

A class table CT is a mapping from class names `C` to class declarations `CD`. A refinement table RT is a mapping from refinement name `C@feat` to refinement declarations. An FFJ program consists of a triple (CT, RT, `e`) of a class table, a refinement table and an expression. Throughout the rest of the paper the CT and the RT are assumed to be always fixed to lighten the notation.

4.2 Lookup Functions

In FFJ as well as in FJ types are classes and classes have a subclass relation defined by the syntax of class declaration. To navigate this subclass relation in the CT, the auxiliary operator `<:` is given in Table 4.2, this operator is the reflexive and transitive closure of the subclass relation.

The CT is expected to satisfy some sanity conditions:

- CT (`C`) = `class C`... for every `C` $\in$ dom(CT)
- `Object` $\notin$ dom(CT)

- for every class name `C` (except `Object`) appearing anywhere in CT, we have $\mathtt{C} \in \mathrm{dom}(\mathsf{CT})$
- there are no cycles in the subtype relation induced by CT, i.e., the relation `<:` is antisymmetric

Subtyping

$$\frac{}{\mathtt{C <: C}} \qquad \frac{\mathtt{C <: D} \quad \mathtt{C <: E}}{\mathtt{C <: E}} \qquad \frac{\mathtt{class\ C\ extends\ D\ \{\ ...\ \}}}{\mathtt{C <: D}}$$

Table 4.2: Subtype Relation

In FFJ$\star$ we fetch the refinement precedence via its position in the RT, i.e. if a refinement of a class appears first in the RT it will be applied first. These functions to navigate the RT are all defined in Table 4.3

First we have the function *class_name* which retrieves the name of a class refinement.

Next we define the function $refinements\_of$ `C` to retrieve the refinements of a given class in the same order as they were introduced in the RT.

To navigate the precedence we define the *pred* and the *last* functions. The *pred* function will get a class refinement as an argument, filter refinements of the same class of `R` as $\bar{\mathtt{R}}$, fetch the *index* n of `R` in $\bar{\mathtt{R}}$ and return the element `P` at the position $n-1$ in $\bar{\mathtt{R}}$ (denoted by the *get* function). Notice that *pred* is a partial function because it is not defined if the a refinement is the first refinement.

The *last* function retrieves the last refinement of a given class `C`. This is needed because in FFJ$\star$ we navigate the refinement chain backwards, from the last refinement to the first, looking for a given method or field.

Class Name

$$\frac{\mathtt{R} = \mathtt{C@feat}}{class\_name\ \mathtt{R} = \mathtt{C}}$$

Refinements of a class

$$\frac{filter\ (\lambda R \cdot class\_name\ \mathtt{R} == \mathtt{C})\ \mathsf{RT} = \bar{\mathtt{R}}}{refinements\_of\ \mathtt{C} = \bar{\mathtt{R}}}$$

Predecessor

$$\frac{refinements\_of\ (class\_name\ \mathtt{R}) = \bar{\mathtt{R}} \qquad index\ \mathtt{R}\ \bar{\mathtt{R}}\ = n \qquad get\ (n-1)\ \bar{\mathtt{R}}\ = \mathtt{P}}{pred\ \mathtt{R}\ = \mathtt{P}}$$

Last

$$\frac{refinements\_of\ \mathtt{C} = \bar{\mathtt{R}} \qquad tail\ \bar{\mathtt{R}}\ = \mathtt{R}}{last\ \mathtt{C}\ = \mathtt{R}}$$

Table 4.3: Refinement Relations

With this in hand we can define the actual lookup functions *fields*, *mtypes* and *mbody*. They are taken directly from FFJ definition, with a new hypothesis and an extra rule.

The extra rule and hypothesis makes reference to dealing with the refinements. This is necessary to make the proofs easier to maintain, since all we need to do is to provide a few acceptance lemmas about these new lookup functions which we name $fields_R$, $mtype_R$ and $mbody_R$.

$fields_R$ simply retrieves the fields of all refinements up to that point in the refinement chain.

$mtype_R$ and $mbody_R$ tries to find the last introduction to a method, and retrieves its type or body. Notice that these two definitions greatly differs from FFJ to FFJ$\star$. In FFJ *mtype* would retrieve the typing of the first method introduction, whereas in FFJ$\star$ it will retrieve the type of the last method refinement, and only later we define the rules for guarantying that the refinement always has the same type of the method declaration. This was made to greatly simplify the proof that states that if a method has *mtype* then it also has a *mbody*, since both functions follows the same structure the proof is straightforward.

$$\frac{\texttt{refines R \{}\bar{\texttt{C}}\ \bar{\texttt{f}}\texttt{; KR }\bar{\texttt{M}}\ \bar{\texttt{MR}}\texttt{\}} \qquad \neg pred\ \texttt{R}}{fields_R\ \texttt{R} = \bar{\texttt{C}}\ \bar{\texttt{f}}}$$

$$\frac{\texttt{refines R \{}\bar{\texttt{C}}\ \bar{\texttt{f}}\texttt{; KR }\bar{\texttt{M}}\ \bar{\texttt{MR}}\texttt{\}} \qquad pred\ \texttt{R} = \texttt{P}}{fields_R\ \texttt{C} = fields_R\ \texttt{P},\ \bar{\texttt{C}}\ \bar{\texttt{f}}}$$

$$fields\ \texttt{Object} = \bullet$$

$$\frac{\texttt{class C extends D \{}\bar{\texttt{C}}\ \bar{\texttt{f}}\texttt{; K }\bar{\texttt{M}}\texttt{\}} \qquad \neg last\ \texttt{C}}{fields\ \texttt{C} = fields\ \texttt{D},\ \bar{\texttt{C}}\ \bar{\texttt{f}}}$$

$$\frac{\texttt{class C extends D \{}\bar{\texttt{C}}\ \bar{\texttt{f}}\texttt{; K }\bar{\texttt{M}}\texttt{\}} \qquad last\ \texttt{C} = \texttt{R}}{fields\ \texttt{C} = fields\ \texttt{D},\ \bar{\texttt{C}}\ \bar{\texttt{f}}, fields_R\ \texttt{R}}$$

Table 4.4: Field lookup

Override function in 4.7 inductively guaranties that a method refinement respects the type a method was introduced for the first time, which can be in a super class or in a previous refinement.

Introduce in 4.8 function checks if a method was not yet declared earlier in the refinement chain.

Every class and refinement of a FFJ$\star$ program is assumed to respect the well-formednes rules defined in 4.10.

4.3 Typing and Reduction

The typing and computation rules for expressions are elided since are the same as FJ. An environment Γ is a finite mapping from variables to types, written $\bar{c} : \bar{C}$. The typing judgment for expressions has the form $\Gamma \vdash e : C$, read "in the environment Γ, expression e has type C".

$$\frac{\texttt{refines class R \{}\bar{\texttt{C}}\ \bar{\texttt{f}}\texttt{; KR }\bar{\texttt{M}}\ \overline{\texttt{MR}}\texttt{\}} \qquad \texttt{B m (}\bar{\texttt{B}}\ \bar{\texttt{x}}\texttt{) \{return e;\}} \in \bar{\texttt{M}}}{mtype_R\ (\texttt{m},\texttt{R}) \ = \ \bar{\texttt{B}} \ \to \ \texttt{B}}$$

$$\frac{\begin{array}{c}\texttt{refines class R \{}\bar{\texttt{C}}\ \bar{\texttt{f}}\texttt{; KR }\bar{\texttt{M}}\ \overline{\texttt{MR}}\texttt{\}} \qquad \texttt{m} \notin \bar{\texttt{M}} \\ \texttt{refines B m (}\bar{\texttt{B}}\ \bar{\texttt{x}}\texttt{) \{return e;\}} \in \overline{\texttt{MR}}\end{array}}{mtype_R\ (\texttt{m},\texttt{R}) \ = \ \bar{\texttt{B}} \ \to \ \texttt{B}}
\qquad
\frac{\begin{array}{c}\texttt{refines class R \{}\bar{\texttt{C}}\ \bar{\texttt{f}}\texttt{; KR }\bar{\texttt{M}}\ \overline{\texttt{MR}}\texttt{\}} \\ \texttt{m} \notin \bar{\texttt{M}} \quad \texttt{m} \notin \overline{\texttt{MR}} \quad pred\ \texttt{R} \ = \ \texttt{P}\end{array}}{mtype_R\ (\texttt{m},\texttt{R}) \ = \ mtype_R\ (\texttt{m},\texttt{P})}$$

$$\frac{\begin{array}{c}\texttt{class C extends D \{}\bar{\texttt{C}}\ \bar{\texttt{f}}\texttt{; K }\bar{\texttt{M}}\texttt{\}} \qquad \texttt{B m (}\bar{\texttt{B}}\ \bar{\texttt{x}}\texttt{) \{return e;\}} \in \bar{\texttt{M}} \\ last\ \texttt{C} \ = \ \texttt{R} \qquad \neg mtype_R\ (\texttt{m},\texttt{R})\end{array}}{mtype\ (\texttt{m},\texttt{C}) \ = \ \bar{\texttt{B}} \ \to \ \texttt{B}}$$

$$\frac{\begin{array}{c}\texttt{class C extends D \{}\bar{\texttt{C}}\ \bar{\texttt{f}}\texttt{; K }\bar{\texttt{M}}\texttt{\}} \qquad \texttt{m} \notin \bar{\texttt{M}} \\ last\ \texttt{C} \ = \ \texttt{R} \qquad \neg mtype_R\ (\texttt{m},\texttt{R})\end{array}}{mtype\ (\texttt{m},\texttt{C}) \ = \ mtype\ (\texttt{m},\texttt{D})}$$

$$\frac{\texttt{class C extends D \{}\bar{\texttt{C}}\ \bar{\texttt{f}}\texttt{; K }\bar{\texttt{M}}\texttt{\}} \qquad last\ \texttt{C} \ = \ \texttt{R}}{mtype\ (\texttt{m},\texttt{C}) \ = \ mtype_R\ (\texttt{m},\texttt{R})}$$

Table 4.5: Method type lookup

$$\frac{\texttt{refines class R \{}\bar{\texttt{C}}\ \bar{\texttt{f}}\texttt{; KR }\bar{\texttt{M}}\ \overline{\texttt{MR}}\texttt{\}} \qquad \texttt{B m (}\bar{\texttt{B}}\ \bar{\texttt{x}}\texttt{) \{return e;\}} \in \bar{\texttt{M}}}{mbody_R\ (\texttt{m},\texttt{R}) \ = \ \bar{\texttt{x}}\texttt{.e}}$$

$$\frac{\begin{array}{c}\texttt{refines class R \{}\bar{\texttt{C}}\ \bar{\texttt{f}}\texttt{; KR }\bar{\texttt{M}}\ \overline{\texttt{MR}}\texttt{\}} \qquad \texttt{m} \notin \bar{\texttt{M}} \\ \texttt{refines B m (}\bar{\texttt{B}}\ \bar{\texttt{x}}\texttt{) \{return e;\}} \in \overline{\texttt{MR}}\end{array}}{mbody_R\ (\texttt{m},\texttt{R}) \ = \ \bar{\texttt{x}}\texttt{.e}}
\qquad
\frac{\begin{array}{c}\texttt{refines class R \{}\bar{\texttt{C}}\ \bar{\texttt{f}}\texttt{; KR }\bar{\texttt{M}}\ \overline{\texttt{MR}}\texttt{\}} \\ \texttt{m} \notin \bar{\texttt{M}} \quad \texttt{m} \notin \overline{\texttt{MR}} \quad pred\ \texttt{R} \ = \ \texttt{P}\end{array}}{mbody_R\ (\texttt{m},\texttt{R}) \ = \ mbody_R\ (\texttt{m},\texttt{P})}$$

$$\frac{\begin{array}{c}\texttt{class C extends D \{}\bar{\texttt{C}}\ \bar{\texttt{f}}\texttt{; K }\bar{\texttt{M}}\texttt{\}} \qquad \texttt{B m (}\bar{\texttt{B}}\ \bar{\texttt{x}}\texttt{) \{return e;\}} \in \bar{\texttt{M}} \\ last\ \texttt{C} \ = \ \texttt{R} \qquad \neg mbody_R\ (\texttt{m},\texttt{R})\end{array}}{mbody\ (\texttt{m},\texttt{C}) \ = \ \bar{\texttt{x}}\texttt{.e}}$$

$$\frac{\begin{array}{c}\texttt{class C extends D \{}\bar{\texttt{C}}\ \bar{\texttt{f}}\texttt{; K }\bar{\texttt{M}}\texttt{\}} \qquad \texttt{m} \notin \bar{\texttt{M}} \\ last\ \texttt{C} \ = \ \texttt{R} \qquad \neg mbody_R\ (\texttt{m},\texttt{R})\end{array}}{mbody\ (\texttt{m},\texttt{C}) \ = \ mbody\ (\texttt{m},\texttt{D})}$$

$$\frac{\texttt{class C extends D \{}\bar{\texttt{C}}\ \bar{\texttt{f}}\texttt{; K }\bar{\texttt{M}}\texttt{\}} \qquad last\ \texttt{C} \ = \ \texttt{R}}{mbody\ (\texttt{m},\texttt{C}) \ = \ mbody_R\ (\texttt{m},\texttt{R})}$$

Table 4.6: Method Body lookup

The reduction relation is of the form $e \to e'$, read "expression e reduces to expression e' in one step", We write $\to *$ for the reflexive and transitive closure of $\to$.

There are three reduction rules, one for field access, one for method invocation, and one for casting. We write $[\bar{d} = \bar{x}, e = y]e_0$ for the result of replacing x_1 by d_1, x_2 by $d_2, \ldots, x_n$ by d_n, and y by e in the expression e_0.

Notice that with the absence of side effects, there is no need of stack or heap for variable binding.

$$\frac{\mathit{mtype}\ (\mathtt{m},\mathtt{D}) = \bar{\mathtt{D}} \rightarrow \mathtt{D}\ \mathit{implies}\ \bar{\mathtt{C}} = \bar{\mathtt{D}}\ \mathit{and}\ \mathtt{C}_0 = \mathtt{D}}{\mathit{override}\ \mathtt{m}\ \mathtt{D}\ \bar{\mathtt{C}}\ \mathtt{C}_0}$$

$$\frac{\mathtt{class\ C\ extends\ D\ \{}\bar{\mathtt{C}}\ \bar{\mathtt{f}}\mathtt{;\ K}\ \bar{\mathtt{M}}\mathtt{\}} \quad \mathtt{C}_0\ \mathtt{m}\ \mathtt{(}\bar{\mathtt{C}}\ \bar{\mathtt{x}}\mathtt{)\ \{return\ e;\}} \in \bar{\mathtt{M}} \quad \neg\, \mathit{pred}\ \mathtt{R} \quad \mathtt{R} = \mathtt{C@feat}}{\mathit{override}_R\ \mathtt{m}\ \mathtt{R}\ \bar{\mathtt{C}}\ \mathtt{C}_0}$$

$$\frac{\mathtt{refines\ class\ P\ \{}\bar{\mathtt{C}}\ \bar{\mathtt{f}}\mathtt{;\ KR}\ \bar{\mathtt{M}}\ \bar{\mathtt{M}}\mathtt{R\}} \quad \mathtt{C}_0\ \mathtt{m}\ \mathtt{(}\bar{\mathtt{C}}\ \bar{\mathtt{x}}\mathtt{)\ \{return\ e;\}} \in \bar{\mathtt{M}} \quad \mathit{pred}\ \mathtt{R} = \mathtt{P}}{\mathit{override}_R\ \mathtt{m}\ \mathtt{R}\ \bar{\mathtt{C}}\ \mathtt{C}_0}$$

$$\frac{\mathtt{refines\ class\ P\ \{}\bar{\mathtt{C}}\ \bar{\mathtt{f}}\mathtt{;\ KR}\ \bar{\mathtt{M}}\ \bar{\mathtt{M}}\mathtt{R\}} \quad \mathtt{m} \notin \bar{\mathtt{M}} \quad \mathit{pred}\ \mathtt{R} = \mathtt{P} \quad \mathit{override}_R\ \mathtt{m}\ \mathtt{P}\ \bar{\mathtt{C}}\ \mathtt{C}_0}{\mathit{override}_R\ \mathtt{m}\ \mathtt{R}\ \bar{\mathtt{C}}\ \mathtt{C}_0}$$

Table 4.7: Override Function

$$\frac{\mathit{pred}\ \mathtt{R} = \mathtt{S} \quad \neg\, \mathit{mtype}_R\ (\mathtt{m},\mathtt{S})}{\mathit{introduce}\ \mathtt{m}\ \mathtt{R}}$$

$$\frac{\neg\, \mathit{pred}\ \mathtt{R} \quad \mathtt{R} = \mathtt{C@feat} \quad \mathtt{class\ C\ extends\ D\ \{}\bar{\mathtt{C}}\ \bar{\mathtt{f}}\mathtt{;\ K}\ \bar{\mathtt{M}}\mathtt{\}} \quad \mathtt{m} \notin \bar{\mathtt{M}}}{\mathit{introduce}\ \mathtt{m}\ \mathtt{R}}$$

Table 4.8: Introduce Function

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

$$\frac{\bar{\mathtt{x}} : \bar{\mathtt{C}}, \mathtt{this} : \mathtt{C} \vdash \mathtt{t}_0 : \mathtt{E}_0 \qquad \mathtt{E}_0 <: \mathtt{C}_0 \qquad \mathtt{R} = \mathtt{C@feat} \qquad \mathtt{CT(C)} = \mathtt{class\ C\ extends\ D}\ \{\dots\} \qquad \mathtt{RT(R)} = \mathtt{refines\ R}\ \{\dots\ \bar{\mathtt{M}}\ \dots\} \qquad \mathit{override}(\mathtt{m}, \mathtt{D}, \bar{\mathtt{C}} \to \mathtt{C}) \qquad \mathit{introduce}\ \mathtt{m}\ \mathtt{R} \qquad \mathtt{m} \in \bar{\mathtt{M}}}{\mathtt{C}_0\ \mathtt{m}\ (\bar{\mathtt{C}}\ \bar{\mathtt{x}})\{\mathtt{return}\ \mathtt{t}_0;\}\ \mathtt{OK\ in\ R}}$$

$$\frac{\bar{\mathtt{x}} : \bar{\mathtt{C}}, \mathtt{this} : \mathtt{C} \vdash \mathtt{t}_0 : \mathtt{E}_0 \qquad \mathtt{E}_0 <: \mathtt{C}_0 \qquad \mathtt{R} = \mathtt{C@feat} \qquad \mathtt{RT(R)} = \mathtt{refines\ R}\ \{\dots\ \bar{\mathtt{M}},\ \overline{\mathtt{MR}}\ \dots\} \qquad \mathtt{m} \notin \bar{\mathtt{M}} \qquad \mathtt{m} \in \overline{\mathtt{MR}} \qquad \mathit{override}_R(\mathtt{m}, \mathtt{R}, \bar{\mathtt{C}} \to \mathtt{C}) \qquad \mathit{introduce}\ \mathtt{m}\ \mathtt{R}}{\mathtt{refines}\ \mathtt{C}_0\ \mathtt{m}\ (\bar{\mathtt{C}}\ \bar{\mathtt{x}})\{\mathtt{return}\ \mathtt{t}_0;\}\ \mathtt{OK\ in\ R}}$$

Table 4.9: Method typing in FFJ⋆

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

$$\frac{\bar{\mathtt{M}}\ \mathtt{OK\ in\ R} \qquad \overline{\mathtt{MR}}\ \mathtt{OK\ in\ R}}{\mathtt{refines\ class\ R}\ \{\bar{\mathtt{C}}\ \bar{\mathtt{f}};\ \mathtt{KR}\ \bar{\mathtt{M}}\ \overline{\mathtt{MR}}\}\ \mathtt{OK}}$$

Table 4.10: Class and refinement typing in FFJ⋆

Chapter 5

Related Work

Chapter 6

Implications

Chapter 7

Conclusion

References

[1] Sidney Amani and Toby Murray. Specifying a realistic file system. *arXiv preprint arXiv:1511.04169*, 2015. 2

[2] Sven Apel, Christian Kästner, and Christian Lengauer. Feature Featherweight Java: A Calculus for Feature-oriented Programming and Stepwise Refinement. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*, GPCE '08, pages 101–112, New York, NY, USA, 2008. ACM. 1, 7, 9

[3] Sven Apel, Thomas Leich, Marko Rosenmüller, and Gunter Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Generative Programming and Component Engineering*, pages 125–140. Springer, Berlin, Heidelberg, September 2005. 1, 3

[4] Sven Apel and Christian Lengauer. Superimposition: A Language-Independent Approach to Software Composition. In *Software Composition*, pages 20–35. Springer, Berlin, Heidelberg, March 2008. 1

[5] Konstantine Arkoudas, Karen Zee, Viktor Kuncak, and Martin Rinard. Verifying a file system implementation. In *International Conference on Formal Engineering Methods*, pages 373–390. Springer, 2004. 2

[6] D. Batory. Feature-oriented programming and the AHEAD tool suite. In *Proceedings. 26th International Conference on Software Engineering*, pages 702–703, May 2004. 1, 3

[7] D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: tools for implementing domain-specific languages. In *Proceedings. Fifth International Conference on Software Reuse (Cat. No.98TB100203)*, pages 143–153, June 1998. 1

[8] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, June 2004. 3

[9] Don Batory. A Tutorial on Feature Oriented Programming and Product-lines. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 753–754, Washington, DC, USA, 2003. IEEE Computer Society. 1

[10] Don Batory, Clay Johnson, Bob MacDonald, and Dale von Heeder. Achieving Extensibility through Product-Lines and Domain-Specific Languages: A Case Study. In *Software Reuse: Advances in Software Reusability*, pages 117–136. Springer, Berlin, Heidelberg, June 2000. 1

[11] Don Batory and Sean O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Trans. Softw. Eng. Methodol.*, 1(4):355–398, October 1992. 1

[12] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings of the European Conference on Object-oriented Programming on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA/ECOOP '90, pages 303–311, New York, NY, USA, 1990. ACM. 3

[13] Benjamin Delaware, William Cook, and Don Batory. A machine-checked model of safe composition. In *Proceedings of the 2009 workshop on Foundations of aspect-oriented languages*, pages 31–35. ACM, 2009. 1

[14] Pieter H Hartel. Formalising java safety—an overview. In *Smart Card Research and Advanced Applications*, pages 115–134. Springer, 2000. 2

[15] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001. 1, 7

[16] Gerwin Klein and Tobias Nipkow. A machine-checked model for a java-like language, virtual machine, and compiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(4):619–695, 2006. 2

[17] R. E. Kurt Stirewalt and Laura K. Dillon. A Component-based Approach to Building Formal Analysis Tools. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 167–176, Washington, DC, USA, 2001. IEEE Computer Society. 1

[18] Xavier Leroy et al. The compcert verified compiler. *Documentation and user's manual. INRIA Paris-Rocquencourt*, 2012. 2

[19] CN Little. Xxx.—non-alternate±knots. *Earth and Environmental Science Transactions of The Royal Society of Edinburgh*, 39(3):771–778, 1900. 1

[20] David Lorge Parnas. On the design and development of program families. *IEEE Transactions on software engineering*, (1):1–9, 1976. 3

[21] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *ECOOP'97 — Object-Oriented Programming*, pages 419–443. Springer, Berlin, Heidelberg, June 1997. 1

[22] Dale Rolfsen. *Knots and links*, volume 346. American Mathematical Soc., 1976. 1

[23] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-Oriented Programming of Software Product Lines. In *Software Product Lines: Going Beyond*, pages 77–91. Springer, Berlin, Heidelberg, September 2010. 1

[24] Ian Sommerville. *Software Engineering*. Addison-Wesley Publishing Company, USA, 9th edition, 2010. 3