Mechanization and Overhaul of Feature Featherweight Java

Pedro Abreu(✉) and Rodrigo Bonifácio

Departamento de Ciência da Computação, Universidade de Brasília, Brazil
abreu223@gmail.com

Abstract. Specifying a language in an Interactive Theorem Prover (ITP) is seldom faithful to its original pen-and-paper specification. However the process of mechanizing the language and type safety proofs might unearth insights for clearer overall specification. In the present we discuss the process of extending a mechanization of *Featherweight Java (FJ)* with *Feature Featherweight Java (FFJ)* in `Coq` and how that lead us to a clearer, unambiguous and equivalent syntax and semantics while keeping the proofs as simple as possible. This is also the first mechanization of a *Feature Oriented Programming (FOP)* language to date.

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

1 Introduction

Feature Oriented Programming (FOP) [11] is a design methodology and tools for program synthesis [6]. 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 [4], FSTComposer [3], FeatureC++ [2], and more recently Delta-Oriented Programming [12]. FOP has also been used to develop *product-lines* in disparate domains, including compilers for extensible Java dialects [5], fire support simulators for U.S. Army [7], high-performance network [8], and program verification tools [10].

Due to the relevance of FOP, falar sobre lps...

Several attempts to formalize the type system of FOP languages have been made. For instance, *Feature Featherweight Java (FFJ)* [1] is a proposed type system for FOP languages and tools, which is developed on top of *Featherweight Java (FJ)* [9] to provide a simple syntax and semantics conforming with common FOP languages, incorporating constructs for feature composition.

Nevertheless, none of the existing efforts for specifying FOP type system have been mechanized to date. Which means that we have no formal guarantees that

the specification is type-safe other than peer review. Such a method is known for enabling small errors to remain hidden for several years, specially as the proofs grow larger and harder to follow. 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.

In this paper, we present an implementation of FJ which we extend with FFJ using `Coq`. The process of scrutinizing FFJ and defining *unambigously* its semantics in `Coq` lead us to some language specification and implementation improvements. Altogether these improvements makes more natural the transition between FJ and FFJ, simplifying the auxiliary functions used in the language specification as well as the type safety proofs and lemmas. Not only to be easier to increment FJ syntax and semantics, but also easier increment its proofs leading to a clearer and simpler specification of *FFJ*. Hence we can sumarize the main contribution of this paper as follows:

1. The first mechanization of a FOP type system
2. A better 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 2 we give a brief introduction to `Coq` Section 3 briefly introduces software product lines, FOP and FFJ, Section 4 formally describes our revamped FFJ and states the lemmas needed to preserve FJ increment to FFJ type safety, Section 5 discuss the implications of these results to the product line research, Section 6 discuss related works and Section 7 is the conclusion and shows possible future works.

2 Introduction to Coq

Lorem Ipsum

3 Feature Oriented Programming

4 Feature Featherweight Java

4.1 Syntax

To present the abstract syntax of FJ, first we need to define the meaning of the following metavariables, which will apear frequently in the rules of FJ's grammar: A, B, C, D and E range over class names; f and g range over field names; m ranges over method names; x ranges over variables; d and e range over expressions; L ranges over class declarations; K range over constructor declarations and M ranges over method declarations.

CD	::=	`class C extends C {`$\bar{\texttt{C}}$ $\bar{\texttt{f}}$`; K` $\bar{\texttt{M}}$`}`
CR	::=	`refines class C {`$\bar{\texttt{C}}$ $\bar{\texttt{f}}$`; KD` $\bar{\texttt{M}}$`}`
KD	::=	`C (`$\bar{\texttt{C}}$ $\bar{\texttt{f}}$`) {super (`$\bar{\texttt{f}}$`); this.`$\bar{\texttt{f}}$`=`$\bar{\texttt{f}}$`;}`
MD	::=	`C m (`$\bar{\texttt{C}}$ $\bar{\texttt{x}}$`) {return e;}`
MR	::=	`refines C m (`$\bar{\texttt{C}}$ $\bar{\texttt{x}}$`) {return e;}`
e	::=	`x | e.f | e.m (`$\bar{\texttt{e}}$`) | new C (`$\bar{\texttt{e}}$`) | (C) e`
v	::=	`new C (`$\bar{\texttt{e}}$`)`

Table 1. FFJ Syntax

The abstract syntax of FJ class declarations, constructor declarations, method declarations and expressions is given at Table 1.

The variable `this` is assumed to be included in the set of variables, but it cannot be used as the name of an argument to a method.

It is written $\bar{f}$ as a shorthand for a possibly empty sequence $f_1, \ldots, f_n$ (and similary for $\bar{C}$, $\bar{x}$, $\bar{e}$ etc.) and $\bar{M}$ as a shorthand for $M_1 \ldots M_n$ (with no commas).

A class table CT is a mapping from class names C to class declarations L. A program is a pair (CT, e) of a class table and an expression. In FJ, every class has a superclass declared with extends, with the exception of `Object`, which is taken as a distinguished class name whose definition does *not* appear in the class table. The class table contains the subtype relation between all the classes. From the Table 2, one can infer that subtyping is the reflexive and transitive closure of the immediate subclass ralation given by the `extends` clauses in CT.

Table 2. Subtyping

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

The function *fields* defined at Table 3 returns a list of all the fields declared at the current class definition, as well as the fields declared at the superclass of it. It returns an empty list in the case of `Object`.

Table 3. Field lookup

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

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

The method declaration $D\ m\ (\bar{C}\ \bar{x})\ \{return\ e;\}$ introduces a method name m with result type D and parameters $\bar{x}$ of types $\bar{C}$. The body of the method is the single statement $returne$;. The variables x and the special variable *this* are bound in e. The rules that define these functions are ilustrated in Tables 4 and 5.

Table 4. Method type lookup

$$\frac{class\ C\ extends\ D\ \{\bar{C}\ \bar{f};\ K\ \bar{M}\} \qquad B\ m\ (\bar{B}\ \bar{x})\{return\ e;\} \in\ \bar{M}}{mtype\ (m,\ C) = \bar{B} \rightarrow\ B}$$

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

Table 5. Method body lookup

$$\frac{class\ C\ extends\ D\ \{\bar{C}\ \bar{f};\ K\ \bar{M}\} \qquad B\ m\ (\bar{B}\ \bar{x})\{return\ e;\} \in\ \bar{M}}{mbody\ (m,\ C) = \bar{x}.e}$$

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

The given class table is assumed to satisfy the following conditions:

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

4.2 Typing

The typing rules for expressions are in Table 6. 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".

4.3 Computation

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

Table 6. Expression typing

$$\Gamma \vdash x : \Gamma(x) \qquad \text{(T-Var)}$$

$$\frac{\Gamma \vdash e_0 : C_0 \qquad \mathit{fields}\ (C_0) = \bar{C}\ \bar{f}}{\Gamma \vdash e_0.f_i : C_i} \qquad \text{(T-Field)}$$

$$\frac{\Gamma \vdash e_0 : C_0 \qquad \mathit{mtypes}\ (m,\ C_0) = \bar{D} \to C \qquad \Gamma \vdash \bar{e} : \bar{C} \qquad \bar{C}\ <:\ \bar{D}}{\Gamma \vdash e_0.m(\bar{e}) : C} \qquad \text{(T-Invk)}$$

$$\frac{\mathit{fields}(C) = \bar{D}\ \bar{f} \qquad \Gamma \vdash \bar{e} : \bar{C} \qquad \bar{C}\ <:\ \bar{D}}{\Gamma \vdash \mathit{new}\ C(\bar{e}) : C} \qquad \text{(T-New)}$$

$$\frac{\Gamma \vdash e_0 : D \qquad D\ <:\ C}{\Gamma \vdash (C)\ e_0 : C} \qquad \text{(T-UCast)}$$

$$\frac{\Gamma \vdash e_0 : D \qquad C\ <:\ D \qquad C \neq D}{\Gamma \vdash (C)\ e_0 : C} \qquad \text{(T-DCast)}$$

$$\frac{\Gamma \vdash e_0 : D \qquad C\ \not<:\ D \qquad D\ \not<:\ C \qquad \mathit{stupid\ warning}}{\Gamma \vdash (C)\ e_0 : C} \qquad \text{(T-SCast)}$$

The reduction rules are given in 7. There are three reduction rules, one for field acess, one for method invocation, and one for casting. These were already explained above. 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 again that with the absense of side effects, there is no need of stack or heap for variable binding.

The reduction rules may be applied at any point in an expression, so we also need the obvious congruence rules.

Table 7. Expression computation

$$\frac{\mathit{fields}\ (C) = \bar{C}\bar{f}}{(\mathit{new}\ C(\bar{e})).f_i \to e_i} \qquad \text{(R-Field)}$$

$$\frac{\mathit{mbody}\ (m, C) = \bar{x}.e_0}{(\mathit{new}\ C\ (\bar{e})).m\ (\bar{d}) \to [\bar{d}/\bar{x}, \mathit{new}\ C\ (\bar{e})/\mathit{this}]e_0} \qquad \text{(R-Invk)}$$

$$\frac{C <: D}{(D)(\mathit{new}\ C\ (\bar{e})) \to \mathit{new}\ C\ (\bar{e})} \qquad \text{(R-Cast)}$$

Table 8. Congruence

$$\frac{e_0 \to e_0'}{e_0.f \to e_0'.f} \quad \text{(RC-Field)}$$

$$\frac{e_0 \to e_0'}{e_0.m\ (\bar{e}) \to e_0'.m\ (\bar{e})} \quad \text{(RC-Invk-Recv)}$$

$$\frac{e_i \to e_i'}{e_0.m\ (\ldots, e_i, \ldots) \to e_0'.m\ (\ldots, e_i, \ldots)} \quad \text{(RC-Invk-Arg)}$$

$$\frac{e_i \to e_i'}{new\ C\ (\ldots, e_i, \ldots) \to new\ C\ (\ldots, e_i', \ldots)} \quad \text{(RC-New-Arg)}$$

$$\frac{e_0 \to e_0'}{(C)e_0 \to (C)e_0'} \quad \text{(RC-Cast)}$$

5 Implications

6 Related Work

7 Conclusion

References

1. Apel, S., Kästner, C., Lengauer, C.: 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. pp. 101–112. GPCE '08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1449913.1449931
2. Apel, S., Leich, T., RosenmÃijller, M., Saake, G.: FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In: Generative Programming and Component Engineering. pp. 125–140. Springer, Berlin, Heidelberg (Sep 2005), https://link.springer.com/chapter/10.1007/11561347_10
3. Apel, S., Lengauer, C.: Superimposition: A Language-Independent Approach to Software Composition. In: Software Composition. pp. 20–35. Springer, Berlin, Heidelberg (Mar 2008), https://link.springer.com/chapter/10.1007/978-3-540-78789-1_2
4. Batory, D.: Feature-oriented programming and the AHEAD tool suite. In: Proceedings. 26th International Conference on Software Engineering. pp. 702–703 (May 2004)
5. Batory, D., Lofaso, B., Smaragdakis, Y.: JTS: tools for implementing domain-specific languages. In: Proceedings. Fifth International Conference on Software Reuse (Cat. No.98TB100203). pp. 143–153 (Jun 1998)
6. Batory, D.: A Tutorial on Feature Oriented Programming and Product-lines. In: Proceedings of the 25th International Conference on Software Engineering. pp. 753–754. ICSE '03, IEEE Computer Society, Washington, DC, USA (2003), http://dl.acm.org/citation.cfm?id=776816.776935

7. Batory, D., Johnson, C., MacDonald, B., Heeder, D.v.: Achieving Extensibility through Product-Lines and Domain-Specific Languages: A Case Study. In: Software Reuse: Advances in Software Reusability. pp. 117–136. Springer, Berlin, Heidelberg (Jun 2000), https://link.springer.com/chapter/10.1007/978-3-540-44995-9_8
8. Batory, D., O'Malley, S.: The Design and Implementation of Hierarchical Software Systems with Reusable Components. ACM Trans. Softw. Eng. Methodol. 1(4), 355–398 (Oct 1992), http://doi.acm.org/10.1145/136586.136587
9. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: A Minimal Core Calculus for Java and GJ. ACM Trans. Program. Lang. Syst. 23(3), 396–450 (May 2001), http://doi.acm.org/10.1145/503502.503505
10. Kurt Stirewalt, R.E., Dillon, L.K.: A Component-based Approach to Building Formal Analysis Tools. In: Proceedings of the 23rd International Conference on Software Engineering. pp. 167–176. ICSE '01, IEEE Computer Society, Washington, DC, USA (2001), http://dl.acm.org/citation.cfm?id=381473.381491
11. Prehofer, C.: Feature-oriented programming: A fresh look at objects. In: ECOOP'97 âĂŤ Object-Oriented Programming. pp. 419–443. Springer, Berlin, Heidelberg (Jun 1997), https://link.springer.com/chapter/10.1007/BFb0053389
12. Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N.: Delta-Oriented Programming of Software Product Lines. In: Software Product Lines: Going Beyond. pp. 77–91. Springer, Berlin, Heidelberg (Sep 2010), https://link.springer.com/chapter/10.1007/978-3-642-15579-6_6