A Co-contextual Type Checker for Featherweight Java

Kuci, Edlira; Erdweg, Sebastian; Bračevac, Oliver; Bejleri, Andi; Mezini, Mira

DOI
10.4230/LIPIcs.ECOOP.2017.18

Publication date
2017

Document Version
Final published version

Published in
31st European Conference on Object-Oriented Programming (ECOOP 2017)

A Co-contextual Type Checker for Featherweight Java

Edlira Kuci[1], Sebastian Erdweg[2], Oliver Bračevac[3], Andi Bejleri[4], and Mira Mezini[5]

1 **Technische Universität Darmstadt, Germany**
2 **TU Delft, The Netherlands**
3 **Technische Universität Darmstadt, Germany**
4 **Technische Universität Darmstadt, Germany**
5 **Technische Universität Darmstadt, Germany and Lancaster University, UK**

Abstract

This paper addresses compositional and incremental type checking for object-oriented programming languages. Recent work achieved incremental type checking for structurally typed functional languages through *co-contextual typing rules*, a constraint-based formulation that removes any context dependency for expression typings. However, that work does not cover key features of object-oriented languages: Subtype polymorphism, nominal typing, and implementation inheritance. Type checkers encode these features in the form of class tables, an additional form of typing context inhibiting incrementalization.

In the present work, we demonstrate that an appropriate co-contextual notion to class tables exists, paving the way to efficient incremental type checkers for object-oriented languages. This yields a novel formulation of Igarashi et al.'s Featherweight Java (FJ) type system, where we replace class tables by the dual concept of class table requirements and class table operations by dual operations on class table requirements. We prove the equivalence of FJ's type system and our co-contextual formulation. Based on our formulation, we implemented an incremental FJ type checker and compared its performance against javac on a number of realistic example programs.

1998 ACM Subject Classification D.3.3 Language Constructs and Features, F.3.1 Specifying and Verifying and Reasoning about Programs, F.3.2 Semantics of Programming Languages

Keywords and phrases type checking; co-contextual; constraints; class table; Featherweight Java

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.18

1 Introduction

Previous work [6] presented a *co-contextual formulation* of the PCF type system with records, parametric polymorphism, and subtyping by duality of the traditional contextual formulation. The contextual formulation is based on a typing context and operations for looking up, splitting, and extending the context. The co-contextual formulation replaces the typing context and its operations with the dual concepts of context requirements and operations for generating, merging, and satisfying requirements. This enables bottom-up type checking that starts at the leaves of an expression tree. Whenever a traditional type checker would look up variable types in the typing context, the bottom-up co-contextual type checker generates fresh type variables and generates context requirements stating that these type variables need to be bound to actual types; it merges and satisfies these requirements as it visits the syntax

$$\textbf{new}\ \mathsf{List}().add(1).size() \ + \ \textbf{new}\ \mathsf{LinkedList}().add(2).size();$$

(R_1) $\mathsf{List}.init()$	(R_4) $\mathsf{LinkedList}.init()$
(R_2) $\mathsf{List}.add : \mathsf{Int} \to U_1$	(R_5) $\mathsf{LinkedList}.add : \mathsf{Int} \to U_2$
(R_3) $U_1.size : () \to U_3$	(R_6) $U_2.size : () \to U_4$

Figure 1 Requirements generated from co-contextually type checking the + expression.

tree upwards to the root. The co-contextual type formulation of PCF enables incremental type checking giving rise to order-of-magnitude speedups [6].

These results motivated us to investigate co-contextual formulation of the type systems for statically typed object-oriented (OO) languages, the state-of-the-art programming technology for large-scale systems. We use Featherweight Java [8] (FJ) as a representative calculus for these languages. Specifically, we consider two research questions: (a) Can we formulate an equivalent co-contextual type system for FJ by duality to the traditional formulation, and (b) if yes, how to define an incremental type checker based on it with significant speedups? Addressing these questions is an important step towards a general theory of incremental type checkers for statically typed OO languages, such as Java, C♯, or Eiffel.

We observe that the general principle of replacing the typing context and its operations with co-contextual duals carries over to the *class table*. The latter is propagated top-down and completely specifies the available classes in the program, e.g., member signatures and super classes. Dually, a co-contextual type checker propagates *class table requirements* bottom-up. This data structure specifies requirements on classes and members and accompanying operations for generating, merging, and removing these requirements.

However, defining appropriate merge and remove operations on co-contextual class table requirements poses significant challenges, as they substantially differ from the equivalent operations on context requirements. Unlike the global namespace and structural typing of PCF, FJ features context dependent member signatures (subtype polymorphism), a declared type hierarchy (nominal typing), and inherited definitions (implementation inheritance).

For an intuition of class table requirements and the specific challenges concerning their operations, consider the example in Figure 1. Type checking the operands of + yields the class table requirements R_1 to R_6. Here and throughout the paper we use metavariable U to denote unification variables as placeholders for actual types. For example, the invocation of method *add* on **new** $\mathsf{List}()$ yields a class table requirement R_2. The goal of co-contextual type checking is to avoid using any context information, hence we cannot look up the signature of $\mathsf{List}.add$ in the class table. Instead, we use a placeholder U_1 until we discover the definition of $\mathsf{List}.add$ later on. As consequence, we lack knowledge about the receiver type of any subsequent method call, such as *size* in our example. This leads to requirement R_3, which states that (yet unknown) class U_1 should exist that has a method *size* with no arguments and (yet unknown) return type U_3. Assuming + operates on integers, type checking the + operator later unifies U_3 and U_4 with Int, thus refining the class table requirements.

To illustrate issues with merging requirements, consider the requirements R_3 and R_6 regarding *size*. Due to nominal typing, the signature of this method depends on U_1 and U_2, where it is yet unknown how these classes are related to each other. It might be that U_1 and U_2 refer to the same class, which implies that these two requirements overlap and the corresponding types of *size* in R_3 and R_6 are unified. Alternatively, it might be the case that U_1 and U_2 are distinct classes, individually declaring a method *size*. Unifying the types of *size* from R_3 and R_6 would be wrong. Therefore, it is locally indeterminate whether a merge should unify or keep the requirements separate.

To illustrate issues with removing class requirements, consider the requirement R_5. Suppose that we encounter a declaration of *add* in LinkedList. Just removing R_5 is not sufficient because we do not know whether LinkedList overrides *add* of a yet unknown superclasss U, or not. Again, the situation is locally indeterminate. In case of overriding, FJ requires that the signatures of overriding and overridden methods be identical. Hence, it would necessary add constraints equating the two signatures. However, it is equally possible that LinkedList.*add* overrides nothing, so that no additional constraints are necessary. If, however, LinkedList inherits *add* from List without overriding it, we need to record the inheritance relation between these two classes, in order to be able to replace U_2 with the actual return type of *size*.

The example illustrates that a co-contextual formulation for nominal typing with subtype polymorphism and implementation inheritance poses new research questions that the work on co-contextual PCF did not address. A key contribution of the work presented in this paper is to answer these questions. The other key contribution is an incremental type checker for FJ based on the co-contextual FJ formulation. We evaluate the initial and incremental performance of the co-contextual FJ type checker on synthesized FJ programs and realistic java programs by comparison to javac and a context-based implementation of FJ.

To summarize, the paper makes the following contributions:

- We present a co-contextual formulation of FJ's type system by duality to the traditional type system formulation by Igarashi et al. [8]. Our formulation replaces the class table by its dual concept of class table requirements and it replaces field/method lookup, class table duplication, and class table extension by the dual operations of requirement generation, merging, and removing. In particular, defining the semantics of merging and removing class table requirements in the presence of nominal types, OO subtype polymorphism, and implementation inheritance constitute a key contribution of this work.
- We present a method to derive co-contextual typing rules for FJ from traditional ones and provide a proof of equivalence between contextual and co-contextual FJ.
- We provide a description of type checker optimizations for co-contextual FJ with incrementalization and a performance evaluation.

2 Background and Motivation

In this section, we present the FJ typing rules from [8] and give an example to illustrate how contextual and co-contextual FJ type checkers work.

2.1 Featherweight Java: Syntax and Typing Rules

Featherweight Java [8] is a minimal core language for modeling Java's type system. Figure 2 shows the syntax of classes, constructors, methods, expressions, and typing contexts. Metavariables C, D, and E denote class names and types; f denotes fields; m denotes method names; **this** denotes the reference to the current object. As is customary, an overline denotes a sequence in the metalanguage. Γ is a set of bindings from variables and **this** to types.

The type system (Figure 3) ensures that variables, field access, method invocation, constructor calls, casting, and method and class declarations are well-typed. The typing judgment for expressions has the form $\Gamma; CT \vdash e : C$, where Γ denotes the typing context, CT the class table, e the expression under analysis, and C the type of e. The typing judgment for methods has the form $C; CT \vdash M$ OK and for classes $CT \vdash L$ OK.

In contrast to the FJ paper [8], we added some cosmetic changes to the presentation. For example, the class table CT is an implicit global definition in FJ. Our presentation explicitly

$$
\begin{array}{rcll}
L & ::= & \textbf{class}\ C\ \textbf{extends}\ D\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\} & \text{class declaration} \\
K & ::= & C(\overline{C}\ \overline{f})\{\textbf{super}(\overline{f});\ \textbf{this}.\overline{f} = \overline{f}\} & \text{constructor} \\
M & ::= & C\ m(\overline{C}\ \overline{x})\{\ \textbf{return}\ e;\} & \text{method declaration} \\
e & ::= & x \mid \textbf{this} \mid e.f \mid e.m(\overline{e}) \mid \textbf{new}\ C(\overline{e}) \mid (C)e & \text{expression} \\
\Gamma & ::= & \emptyset \mid \Gamma; x : C \mid \Gamma; \textbf{this} : C & \text{typing contexts}
\end{array}
$$

Figure 2 Featherweight Java syntax and typing context.

propagates CT top-down along with the typing context. Another difference to Igarashi et al. is in the rule T-New: Looking up all fields of a class returns a constructor signature, i.e., $\text{fields}(C, CT) = C.\textbf{init}(\overline{D})$ instead of returning a list of fields with their corresponding types. We made this subtle change because it clearer communicates the intention of checking the constructor arguments against the declared parameter types. Later on, these changes pay off, because they enable a systematic translation of typing rules to co-contextual FJ (Sections 3 and 4) and give a strong and rigorous equivalence result for the two type systems (Section 5).

Furthermore, we explicitly include a typing rule T-Program for programs, which is implicit in Igarashi et al.'s presentation. The typing judgment for programs has the form $\overline{L}$ OK: A program is well-typed if all class declarations are well-typed. The auxiliary functions addExt, addCtor, addFs, and addMs extract the supertype, constructor, field and method declarations from a class declaration into entries for the class table. Initially, the class table is empty, then it is gradually extended with information from every class declaration by using the above-mentioned auxiliary functions. This is to emphasize that we view the class table as an additional form of typing context, having its own set of extension operations. We describe the class table extension operations and their co-contextual duals formally in Section 3.

2.2 Contextual and Co-Contextual Featherweight Java by Example

We revisit the example from the introduction to illustrate that, in absence of context information, maintaining requirements on class members is non-trivial:

new List().*add*(1).*size*() + **new** LinkedList().*add*(2).*size*().

```
class List extends Object {
  Int size() {...}
  List add(Int a){...}
}
class LinkedList extends List { }
```

Here we assume the class declarations on the right-hand side: List with methods *add*() and *size*() and LinkedList inheriting from List. As before, we assume there are typing rules for numeric Int literals and the + operator over Int values. We use LList instead of LinkedList in Figure 4 for space reasons.

Figure 4 (a) depicts standard type checking with typing contexts in FJ. The type checker in FJ visits the syntax tree "down-up", starting at the root. Its inputs (propagated downwards) are the context Γ, class table CT, and the current subexpression e. Its output (propagated upwards) is the type C of the current subexpression. The output is computed according to the currently applicable typing rule, which is determined by the shape of the current subexpression. The class table used by the standard type checker contains classes List and LinkedList shown above. The type checker retrieves the signatures for the method invocations of *add* and *size* from the class table CT.

To recap, while type checking constructor calls, method invocations, and field accesses the context and the class table flow top-down; types of fields/methods are looked up in the class table.

$$\text{T-Var}\ \frac{\Gamma(x) = C}{\Gamma; CT \vdash x : C} \qquad \text{T-Field}\ \frac{\Gamma; CT \vdash e : C_e \quad \text{field}(f_i, C_e, CT) = C_i}{\Gamma; CT \vdash e.f_i : C_i}$$

$$\text{T-Invk}\ \frac{\Gamma; CT \vdash e : C_e \quad \Gamma; CT \vdash \overline{e} : \overline{C} \quad \text{mtype}(m, C_e, CT) = \overline{D} \to C \quad \overline{C} <: \overline{D}}{\Gamma; CT \vdash e.m(\overline{e}) : C}$$

$$\text{T-New}\ \frac{\Gamma; CT \vdash \overline{e} : \overline{C} \quad \text{fields}(C, CT) = C.\mathbf{init}(\overline{D}) \quad \overline{C} <: \overline{D}}{\Gamma; CT \vdash \mathbf{new}\ C(\overline{e}) : C}$$

$$\text{T-UCast}\ \frac{\Gamma; CT \vdash e : D \quad D <: C}{\Gamma; CT \vdash (C)e : C} \qquad \text{T-DCast}\ \frac{\Gamma; CT \vdash e : D \quad C <: D \quad C \neq D}{\Gamma; CT \vdash (C)e : C}$$

$$\text{T-SCast}\ \frac{\Gamma; CT \vdash e : D \quad C \not<: D \quad D \not<: C}{\Gamma; CT \vdash (C)e : C}$$

$$\text{T-Method}\ \frac{\begin{array}{c}\overline{x} : \overline{C}; \mathbf{this} : C; CT \vdash e : E_0 \qquad E_0 <: C_0 \\ \text{extends}(C, CT) = D \\ \text{if}\ \ \text{mtype}(m, D, CT) = \overline{D} \to D_0, \text{then}\ \overline{C} = \overline{D};\ C_0 = D_0\end{array}}{C; CT \vdash C\ m(\overline{C}\ \overline{x})\{\mathbf{return}\ e\}\ \text{OK}}$$

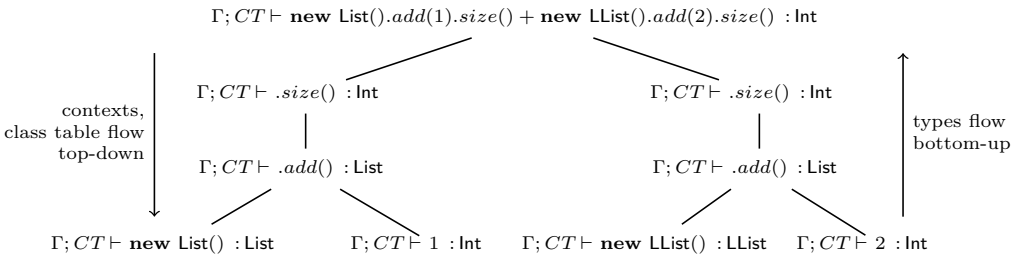
$$\text{T-Class}\ \frac{\begin{array}{c}K = C(\overline{D}'\ \overline{g}, \overline{C}'\ \overline{f})\{\mathbf{super}(\overline{g}); \mathbf{this}.\overline{f} = \overline{f}\} \quad \text{fields}(D, CT) = D.\mathbf{init}(\overline{D}') \\ C; CT \vdash \overline{M}\ \text{OK}\end{array}}{CT \vdash \mathbf{class}\ C\ \mathbf{extends}\ D\ \{\overline{C}\ \overline{f}; K\ \overline{M}\}\ \text{OK}}$$

$$\text{T-Program}\ \frac{\begin{array}{c}CT = \bigcup_{L' \in \overline{L}} (\text{addExt}(L') \cup \text{addCtor}(L') \cup \text{addFs}(L') \cup \text{addMs}(L')) \\ (CT \vdash L'\ \text{OK})_{L' \in \overline{L}}\end{array}}{\overline{L}\ \text{OK}}$$

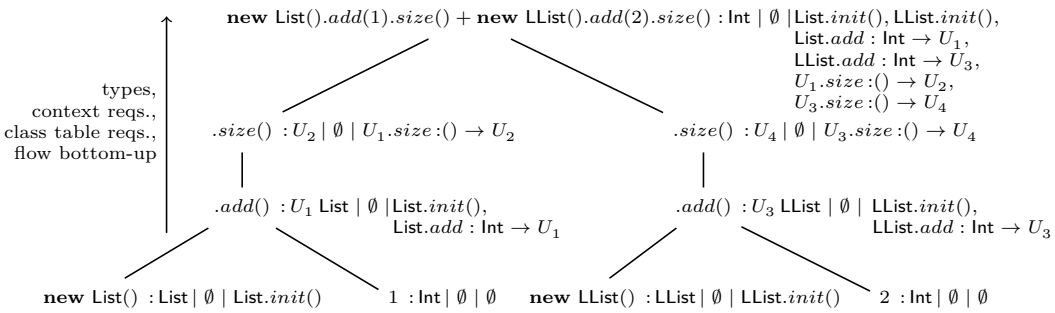
Figure 3 Typing rules of Featherweight Java.

Figure 4 (b) depicts type checking of the same expression in co-contextual FJ. Here, the type checker starts at the leaves of the tree with no information about the context or the class table. The expression type T, the context requirements R, and class table requirements CR all are outputs and only the current expression e is input to the type checker, making the type checker context-independent. At the leaves, we do not know the signature of the constructors of List and LinkedList. Therefore, we generate requirements for the constructor calls List.*init*() and LinkedList.*init*() and propagate them as class table requirements. For each method invocation of *add* and *size* in the tree, we generate requirements on the receiver type and propagate them together with the requirements of the subexpressions.

In addition to generating requirements and propagating them upwards as shown in Figure 4 (b), a co-contextual type checker also *merges requirements* when they have compatible receiver types. In our example, we have two requirements for method *add* and two requirements for method *size*. The requirements for method *add* have incompatible ground receiver types and therefore cannot be merged. The requirements for method *size* both have placeholder receivers and therefore cannot be merged just yet. However, for the *size* requirements, we can already extract a conditional constraint that must hold if the requirements become mergeable, namely ($U_2 = U_4$ *if* $U_1 = U_3$). This constraint ensures the



(a) Contextual type checking propagates contexts and class tables top-down.



(b) Co-contextual type checking propagates context and class table requirements bottom-up.

Figure 4 Contextual and co-contextual type checking.

signatures of both *size* invocations are equal in case their receiver types U_1 and U_3 are equal. This way, we enable early error detection and incremental solving of constraints. Constraints can be solved continuously as soon as they have been generated in order to not wait for the whole program to be type checked. We discuss incremental type checking in more detail in Section 6.

After type checking the + operator, the type checker encounters the class declarations of List and LinkedList. When type checking the class header LinkedList **extends** List, we have to record the inheritance relation between the two classes because methods can be invoked by LinkedList, but declared in List. For example, if List is not known to be a superclass of LinkedList and given the declaration List.*add*, then we cannot just yet satisfy the requirement LinkedList.$add : Num \to U_3$. Therefore, we duplicate the requirement regarding *add* having as receiver List, i.e., List.$add : Num \to U_3$. By doing so, we can deduce the actual type of U_3 for the given declaration of *add* in List. Also, requirements regarding *size* are duplicated.

In the next step, the method declaration of *size* in List is type checked. Hence, we consider all requirements regarding *size*, i.e, $U_1.size : () \to U_2$ and $U_3.size : () \to U_4$. The receivers of *mathitsize* in both requirements are unknown. We cannot yet satisfy these requirements because we do not know whether U_1 and U_3 are equal to List, or not. To solve this, we introduce conditions as part of the requirements, to keep track of the relations between the unknown required classes and the declared ones. By doing so, we can deduce the actual types of U_2 and U_4, and satisfy the requirements later, when we have more information about U_1 and U_3.

Next, we encounter the method declaration *add* and satisfy the corresponding requirements. After satisfying the requirements regarding *add*, the type checker can infer the actual types of U_1 and U_3. Therefore, we can also satisfy the requirements regarding *size*.

To summarize, during the co-contextual type checking of constructor calls, method invocations, and field accesses, the requirements flow bottom-up. Instead of looking up types of fields/methods in the class table, we introduce new class table requirements. These requirements are satisfied when the actual types of fields/methods become available.

3 Co-Contextual Structures for Featherweight Java

In this section, we present the dual structures and operations for the co-contextual formulation of FJ's type system. Specifically, we introduce bottom-up propagated *context and class table requirements*, replacing top-down propagated typing contexts and class tables.

3.1 Class Variables and Constraints

For co-contextual FJ, we reuse the syntax of FJ in Figure 2, but extend the type language to *class types*:

$U, V, \ldots$	Class Variable
$T ::= C \mid U$	Class Type

We use constraints for refining class types, i.e., co-contextual FJ is a constraint-based type system. That is, next to class names, the type system may assign *class variables*, designating unknowns in constraints. We further assume that there are countably many class variables, equality of class variables is decidable and that class variables and class names are disjoint.

During bottom-up checking, we propagate sets S of constraints:

$s ::= T = T \mid T \neq T \mid T <: T \mid T \not<: T \mid T = T$ if *cond*	constraint
$S ::= \emptyset \mid S; s$	constraint set

A constraint s either states that two class types must be equal, non-equal, in a subtype relation, non-subtype, or equal if some condition holds, which we leave underspecified for the moment.

3.2 Context Requirements

A typing context is a set of bindings from variables to types, while a context requirement is a set of bindings from variables to class variables U. Below we show the operations on typing contexts and their co-contextual correspondences, reproduced from [6]. Operations on typing context are lookup, extension, and duplication; their respective requirement context duals are: generating, removing, and merging. Co-contextual FJ adopts context requirements and operations for method parameters and **this** unchanged.

Contextual	Co-contextual
Context syntax $\Gamma ::= \emptyset \mid \Gamma; x:T$	Requirements $R \subset x \times T$ map variables to their types
Context lookup $\Gamma(x) = T$	Requirement introduction $R = \{x:U\}$ with fresh unification variable U
Context extension $\Gamma; x:T$	Requirement satisfaction $R - x$ if $(R(x) = T)$ holds
Context duplication $\Gamma \to (\Gamma, \Gamma)$	Requirement merging $merge_R(R_1, R_2) = R\|_S$ if all constraints $(T_1 = T_2) \in S$ hold
Context is empty $\Gamma = \emptyset$	No unsatisfied requirements $R \stackrel{!}{=} \emptyset$

Contextual

CT	$::= \emptyset$		class table
	$\mid CTcls \cup CT$		
$CTcls$	$::=$		def. clause
	$\mid C$ **extends** D		extends clause
	$\mid C.\mathbf{init}(\overline{C})$		ctor clause
	$\mid C.f : C'$		field clause
	$\mid C.m : \overline{C} \to C'$		method clause

Co-Contextual

CR	$::= \emptyset$		class table req.
	$\mid (CReq, cond) \cup CR$		
$CReq$	$::=$		class req.
	$\mid T$.**extends**: T'		inheritance req.
	$\mid T.\mathbf{init}(\overline{T})$		ctor req.
	$\mid T.f : T'$		field req.
	$\mid T.m : \overline{T} \to T'$		method req.
	$\mid (T.m : \overline{T} \to T')_{opt}$		optional method req.
$cond$	$::= \emptyset \mid T = T'; cond$		condition
	$\mid T \neq T'; cond$		

Figure 5 Class Table and Class Table Requirements Syntax.

3.3 Structure of Class Tables and Class Table Requirements

In the following, we describe the dual notion of a class table, called *class table requirements* and their operations. We first recapitulate the structure of FJ class tables [8], then stipulate the structure of class table requirements. Figure 5 shows the syntax of both. A class table is a collection of class definition clauses $CTcls$ defining the available classes.[1] A clause is a class name C followed by either the superclass, the signature of the constructor, a field type, or a method signature of C's definition.

As Figure 5 suggests, class tables and definition clauses in FJ have a counterpart in co-contextual FJ. Class tables become *class table requirements* CR, which are collections of pairs $(CReq, cond)$, where $CReq$ is a *class requirement* and $cond$ is its *condition*. Each class definition clause has a corresponding class requirement $CReq$, which is one of the following:

- A *inheritance requirement* T .**extends**: T', i.e., class type T must inherit from T'.
- A *constructor requirement* $T.\mathbf{init}(\overline{T}')$, i.e., class type T's constructor signature must match $\overline{T}'$.
- A *field requirement* $T.f : T'$, i.e., class T (or one of its *supertypes*) must declare field f with class type T'.
- A *method requirement* $T.m : \overline{T}' \to T''$, i.e., class T (or one of its *supertypes*) must declare method m matching signature $\overline{T}' \to T''$.
- An *optional method requirement* $(T.m : \overline{T}' \to T'')_{opt}$, i.e., if the class type T declares the method m, then its signature must match $\overline{T}' \to T''$. While type checking method declarations, this requirement is used to ensure that method overrides in subclasses are well-defined. An optional method requirement is used as a counterpart of the conditional method lookup in rule T-Method of standard FJ, i.e., if $\text{mtype}(m, D, CT) = \bar{D} \to D_0, then\ \bar{C} = \bar{D};\ C_0 = D_0$, where D is the superclass of the class C, in which the method declaration m under scrutiny is type checked, and $\bar{C}$, C_0 are the parameter and returned types of m as part of C.

A condition $cond$ is a conjunction of equality and nonequality constraints on class types. Intuitively, $(CReq, cond)$ states that if the condition $cond$ is satisfied, then the requirement

[1] To make the correspondence to class table requirements more obvious, we show a decomposed form of class tables. The original FJ formulation [8] groups clauses by the containing class declaration.

Contextual	Co-contextual
Field name lookup $\text{field}(f_i, C, CT) = C_i$	Class requirement for field $(C.f_i : U, \emptyset)$
Fields lookup $\text{fields}(C, CT) = C.\textbf{init}(\overline{C})$	Class requirement for constructor $(C.\textbf{init}(\overline{U}), \emptyset)$
Method lookup $\text{mtype}(m, C, CT) = \overline{C} \to C$	Class requirement for method $(C.m : \overline{U} \to U, \emptyset)$
Conditional method override $if\ \text{mtype}(m, C, CT) = \overline{C} \to C$	Optional class requirement for method $(C.m : \overline{U} \to U, \emptyset)_{opt}$
Super class lookup $\text{extends}(C, CT) = D$	Class requirement for super class $(C\ .\textbf{extends}: U, \emptyset)$
Class table duplication $CT \to (CT, CT)$	Class requirement merging $merge_{CR}(CR_1, CR_2) = CR\|_S$ if all constraints in S hold

Figure 6 Operations on class table and their co-contextual correspondence.

CReq must be satisfied, too. Otherwise, we have unsolvable constraints, indicating a typing error. With conditional requirements and constraints, we address the feature of nominal typing and inheritance for co-contextual FJ. In the following, we will describe their usage.

3.4 Operations on Class Tables and Requirements

In this section, we describe the co-contextual dual to FJ's class table operations as outlined in Figure 6. We first consider FJ's lookup operations on class tables, which appear in premises of typing rules shown in Figure 3 to look up (1) fields, (2) field lists, (3) methods and (4) superclass lookup. The dual operation is to introduce a corresponding class requirement for the field, list of fields, method, or superclass.

Let us consider closely field lookup, i.e., $\text{field}(f_i, C, CT) = C_i$, meaning that class C in the class table CT has as member a field f_i of type C_i. We translate it to the dual operation of introducing a new class requirement $(C.f_i : U, \emptyset)$. Since we do not have any information about the type of the field, we choose a *fresh* class variable U as type of field f_i. At the time of introducing a new requirement, its condition is empty.

Consider the next operation $\text{fields}(C, CT)$, which looks up all field members of a class. This lookup is used in the constructor call rule T-New; the intention is to retrieve the *constructor signature* in order to type check the subtyping relation between this signature and the types of expressions as parameters of the *constructor call*, i.e., $\bar{C} <: \bar{D}$ (rule T-New). As we can observe, the field names are not needed in this rule, only their types. Hence, in contrast to the original FJ rule [8], we deduce the constructor signature from fields lookup, rather than field names and their corresponding types, i.e., $\text{fields}(C, CT) = C.\textbf{init}(\bar{D})$. The dual operation on class requirements is to add a new class requirement for the constructor, i.e., $(C.\textbf{init}(\bar{U}), \emptyset)$. Analogously, the class table operations for method signature lookup and super class lookup map to corresponding class table requirements.

Finally, standard FJ uses class table duplication to forward the class table to all parts of an FJ program, thus ensuring all parts are checked against the same context. The dual co-contextual operation, $merge_{CR}$, merges class table requirements originating from different parts of the program. Importantly, requirements merging needs to assure all parts of the program require compatible inheritance, constructors, fields, and methods for any given

$$
\begin{aligned}
CR_m = \{&(T_1.m : \overline{T_1} \to T_1', cond_1 \cup (T_1 \neq T_2)) \\
&\cup (T_2.m : \overline{T_2} \to T_2', cond_2 \cup (T_1 \neq T_2)) \\
&\cup (T_1.m : \overline{T_1} \to T_1', cond_1 \cup cond_2 \cup (T_1 = T_2)) \\
&\mid (T_1.m : \overline{T_1} \to T_1', cond_1) \in CR_1 \wedge (T_2.m : \overline{T_2} \to T_2', cond_2) \in CR_2\}
\end{aligned}
$$

$$
\begin{aligned}
S_m = \{&(T_1' = T_2' \; if \; T_1 = T_2) \cup (\overline{T_1 = T_2} \; if \; T_1 = T_2) \\
&\mid (T_1.m : \overline{T_1} \to T_1', cond_1) \in CR_1 \wedge (T_2.m : \overline{T_2} \to T_2', cond_2) \in CR_2\}
\end{aligned}
$$

Figure 7 Merge operation of method requirements CR_1 and CR_2.

class. To merge two sets of requirements, we first identify the field and method names used in both sets and then compare the classes they belong to. The result of merging two sets of class requirements CR_1 and CR_2 is a new set CR of class requirements and a set of constraints, which ensure compatibility between the two original sets of overlapping requirements. Non-overlapping requirements get propagated unchanged to CR whereas potentially overlapping requirements receive special treatment depending on the requirement kind.

The full merge definition appears in our technical report [10]. Figure 7 shows the merge operation for overlapping method requirements, which results in a new set of requirements CR_m and constraints S_m. To compute CR_m, we identify method requirements on the equally-named methods m in both sets and distinguish two cases. First, if the receivers are different $T_1 \neq T_2$, then the requirements are not actually overlapping. We retain the two requirements unchanged, except that we remember the failed condition for future reference. Second, if the receivers are equal $T_1 = T_2$, then the requirements are actually overlapping. We merge them into a single requirement and produce corresponding constraints in S_m. One of the key benefits of keeping track of conditions in class table requirements is that often these conditions allow us to discharge requirements early on when their conditions are unsatisfiable. In particular, in Section 6 we describe a compact representation of conditional requirements that facilitates early pruning and is paramount for good performance. However, the main reason for conditional class table requirements is their removal, which we discuss subsequently.

3.5 Class Table Construction and Requirements Removal

Our formulation of the contextual FJ type system differs in the presentation of the class table compared to the original paper [8]. Whereas Igarashi et al. assume that the class table is a pre-defined static structure, we explicitly consider its formation through a sequence of operations. The class table is initially empty and gradually extended with class table clauses $CTcls$ for each class declaration L of a program. Dual to that, class table requirements are initially unsatisfied and gradually removed. We define an operation for *adding* clauses to the class table and a corresponding co-contextual dual operation on class table requirements for *removing* requirements. Figure 8 shows a collection of adding and removing operations for every possible kind of class table clause $CTcls$.

In general, clauses are added to the class table starting from superclass to subclass declarations. For a given class, the class header with **extends** is added before the other

Contextual	Co-contextual
Class table empty $CT = \emptyset$	Unsatisfied class requirements CR
Adding extend addExt(L, CT)	Remove extend removeExt(L, CR)
Adding constructor addCtor(L, CT)	Remove constructor removeCtor(L, CR)
Adding fields addFs(L, CT)	Remove fields removeFs(L, CR)
Adding methods addMs(L, CT)	Remove methods removeMs(L, CR)

Figure 8 Constructing class table and their co-contextual correspondence.

clauses. Dually, we start removing requirements that correspond to clauses of a subclass, followed by those corresponding to clauses of superclass declarations. For a given class, we first remove requirements corresponding to method, fields, or constructor clauses, then those corresponding to the class header **extends** clause. Note that our sequencing still allows for mutual class dependencies. For example, the following is a valid sequence of clauses where A depends on B and vice versa:

class A **extends** Object; **class** B **extends** Object; A.m: () → B; B.m: () → A.

The full definition of the addition and removal operations for all cases of clause definition appears in our technical report [10]; Figure 9 presents the definitions of adding and removing method and **extends** clauses.

Remove operations for method clauses. The function removeMs removes a list of methods by applying the function removeM to each of them. removeM removes a single method declaration defined in class C. To this end, removeM identifies requirements on the same method name m and refines their receiver to be different from the removed declaration's defining class. That is, the refined requirement $(T.m : \ldots, cond \cup (T \neq C))$ only requires method m if the receiver T is different from the defining class C. If the receiver T is, in fact, equal to C, then the condition of the refined requirement is unsatisfiable and can be discharged. To ensure the required type also matches the declared type, removeM also generates conditional constraints in case $T = C$. Note that whether $T = C$ can often not be determined immediately because T may be a placeholder type U.

We illustrate the removal of methods using the class declaration of List shown in Section 2.2. Consider the class requirement set $CR = (U_1.size() \to U_2, \emptyset)$. Encountering the declaration of method *add* has no effect on this set because there is no requirement on *add*. However, when encountering the declaration of method *size*, we refine the set as follows:

$$removeM(\mathsf{List}, \mathsf{Int}\ size()\ \{\ldots\}, CR) = \{(U_1.size : () \to U_2, \mathsf{U}_1 \neq \mathsf{List})\}|_S$$

with a new constraint $S = \{U_2 = \mathsf{Int}\ if\ \mathsf{U}_1 = \mathsf{List}\}$. Thus, we have satisfied the requirement in CR for $U_1 = \mathsf{List}$, only leaving the requirement in case U_1 represents another type. In particular, if we learn at some point that U_1 indeed represents List, we can discharge the requirement because its condition is unsatisfiable. This is important because a program is only closed and well-typed if its requirement set is empty.

Remove operations for extends clauses. The function removeExt removes the class header clauses (C. **extends** D). This function, in addition to identifying the requirements regarding **extends** and following the same steps as above for removeM, duplicates all requirements for fields and methods. The duplicate introduces a requirement the same as the existing one, but

$$
\begin{aligned}
&\text{addMs}(C, \overline{M}, CT) = \overline{C.m : \overline{C} \to C'} \cup CT \\
&\text{removeM}(C, C'\ m(\overline{C}\ \overline{x})\ \{\textbf{return}\ e\}, CR) = CR'|_S \\
&\textit{where}\ CR' = \{(T.m : \overline{T} \to T', \mathit{cond} \cup (T \neq C)) \mid (T.m : \overline{T} \to T', \mathit{cond}) \in CR\} \\
&\qquad\qquad \cup (CR \setminus \overline{(T.m : \overline{T} \to T', \mathit{cond})}) \\
&\qquad S = \{(T' = C'\ \mathit{if}\ T = C) \cup (\overline{T = C}\ \mathit{if}\ T = C) \mid (T.m : \overline{T} \to T', \mathit{cond}) \in CR\} \\
&\text{removeMs}(C, \overline{M}, CR) = CR'|_S \\
&\textit{where}\ CR' = \{CR_m \mid (C'\ m(\overline{C}\ \overline{x})\ \{\textbf{return}\ e\}) \in \overline{M} \\
&\qquad\qquad \wedge\, \text{removeM}(C, C'\ m(\overline{C}\ \overline{e})\ \{\textbf{return}\ e\}, CR) = CR_m|_{S_m}\} \\
&\qquad\quad S = \{S_m \mid (C'\ m(\overline{C}\ \overline{x})\ \{\textbf{return}\ e\}) \in \overline{M} \\
&\qquad\qquad \wedge\, \text{removeM}(C, C'\ m(\overline{C}\ \overline{x})\ \{\textbf{return}\ e\}, CR) = CR_m|_{S_m}\}
\end{aligned}
$$

$$
\begin{aligned}
&\text{addExt}(\textbf{class}\ C\ \textbf{extends}\ D, CT) = (C\ \textbf{extends}\ D) \cup CT \\
&\text{removeExt}(\textbf{class}\ C\ \textbf{extends}\ D, CR) = CR'|_S \\
&\textit{where}\ CR' = \{(T.\textbf{extends} : T', \mathit{cond} \cup (T \neq C)) \mid (T.\textbf{extends} : T', \mathit{cond}) \in CR\} \\
&\qquad\qquad \cup \{(T.m : \overline{T} \to T', \mathit{cond} \cup (T \neq C)) \\
&\qquad\qquad\quad \cup (D.m : \overline{T} \to T', \mathit{cond} \cup (T = C)) \mid (T.m : \overline{T} \to T', \mathit{cond}) \in CR\} \\
&\qquad\qquad \cup \{(T.m : \overline{T} \to T', \mathit{cond} \cup (T \neq C))_{opt} \\
&\qquad\qquad\quad \cup (D.m : \overline{T} \to T', \mathit{cond} \cup (T = C))_{opt} \\
&\qquad\qquad\quad \mid (T.m : \overline{T} \to T', \mathit{cond})_{opt} \in CR\} \\
&\qquad\qquad \cup \{(T.f : T', \mathit{cond} \cup (T \neq C)) \cup (D.f : T', \mathit{cond} \cup (T = C)) \\
&\qquad\qquad\quad \mid (T.f : T', \mathit{cond}) \in CR\} \\
&\qquad\quad S = \{(T' = D\ \mathit{if}\ T = C) \mid (T.\textbf{extends} : T', \mathit{cond}) \in CR\}
\end{aligned}
$$

Figure 9 Add and remove operations of method and extends clauses.

with a different receiver, which is the superclass D that potentially declares the required fields and methods. The conditions also change. We add to the existing requirements an inequality condition $(T \neq C)$, to encounter the case when the receiver T is actually replaced by C, but it is required to have a certain field or method, which is declared in D, the superclass of T. This requirement should be discharged because we know the actual type of the required field or method, which is inherited from the given declaration in D. Also, we add an equality condition to the duplicate requirement $T = C$, because this requirement will be discharged when we encounter the actual declarations of fields or methods in the superclass.

We illustrate the removal of **extends** using the class declaration LinkedList **extends** List. Consider the requirement set $CR = (U_3.\mathit{size} : () \to U_4, \emptyset)$. We encounter the declaration for LinkedList and the requirement set changes as follows:

$$
\begin{aligned}
&\text{removeExt}(\textbf{class}\ \mathsf{LinkedList}\ \textbf{extends}\ \mathsf{List}, CR) = \\
&\quad \{(U_3.\mathit{size} : () \to U_4, U_3 \neq \mathsf{LinkedList}), (\mathsf{List}.\mathit{size} : () \to U_4, U_3 = \mathsf{LinkedList})\}|_S,
\end{aligned}
$$

where $S = \emptyset$. S is empty, because there are no requirements on **extends**. If we learn at some point that $U_3 = \mathsf{LinkedList}$, then the requirement $(U_3.\mathit{size} : () \to U_4, U_3 \neq \mathsf{LinkedList})$

is discharged because its condition is unsatisfiable. Also, if we learn that *size* is declared in List, then (List.*size* : () → U_4, U_3 = LinkedList) is discharged applying removeM, as shown above, and U_4 can be replaced by its actual type.

Usage and necessity of conditions. As shown throughout this section, conditions play an important role to enable merging and removal of requirements over nominal receiver types and to support inheritance. Because of nominal typing, field and method lookup depends on the name of the defining context and we do not know the actual type of the receiver class when encountering a field or method reference. Thus, it is impossible to deduce their types until more information is known. Moreover, if a class is required to have fields/methods, which are actually declared in a superclass of the required class, then we need to deduce their actual type/signature and meanwhile fulfill the respective requirements. For example, considering the requirement U_3.*size* : () → U_4, if U_3 = LinkedList, LinkedList **extends** List, and *size* is declared in List, then we have to deduce the actual type of U_4 and satisfy this requirement. To overcome these obstacles we need additional structure to maintain the relations between the required classes and the declared ones, and also to reason about the partial fulfillment of requirements. Conditions come to place as the needed structure to maintain these relations and indicate the fulfillment of requirements.

4 Co-Contextual Featherweight Java Typing Rules

In this section we derive co-contextual FJ's typing rules systematically from FJ's typing rules. The main idea is to transform the rules into a form that eliminates any context dependencies that require top-down propagation of information.

Concretely, context and class table requirements (Section 3) in output positions to the right replace typing contexts and class tables in input positions to the left. Additionally, co-contextual FJ propagates constraint sets S in output positions. Note that the program typing judgment does not change, because programs are closed, requiring neither typing context nor class table inputs. Correspondingly, neither context nor class table requirements need to be propagated as outputs.

Figure 10 shows the co-contextual FJ typing rules (the reader may want to compare against contextual FJ in Figure 3). In what follows, we will discuss the rules for each kind of judgment.

4.1 Expression Typing

Typing rule TC-Var is dual to the standard variable lookup rule T-Var. It marks a distinct *occurrence* of x (or the self reference **this**) by assigning a fresh class variable U. Furthermore, it introduces a new context requirement $\{x : U\}$, as the dual operation of context lookup for variables x ($\Gamma(x) = C$) in T-Var. Since the latter does not access the class table, dually, TC-Var outputs empty class table requirements.

Typing rule TC-Field is dual to T-Field for field accesses. The latter requires a field name lookup (field), which, dually, translates to a new class requirement for the field f_i, i.e., $(T_e.f_i : U, \emptyset)$ (cf. Section 3). Here, T_e is the class type of the receiver e. U is a fresh class variable, marking a distinct occurrence of field name f_i, which is the class type of the entire expression. Furthermore, we merge the new field requirement with the class table requirements CR_e propagated from e. The result of merging is a new set of requirements CR and a new set of constraints S_{cr}. Just as the context Γ is passed into the subexpression e in T-Field, we propagate the context requirements for e for the entire expression. Finally,

$$\textsc{TC-Var}\ \frac{U \text{ is fresh}}{x : U \mid \emptyset \mid x : U \mid \emptyset}$$

$$\textsc{TC-Field}\ \frac{e : T_e \mid S_e \mid R_e \mid CR_e \qquad CR|_{S_f} = \text{merge}_{CR}(CR_e, (T_e.f_i : U, \emptyset)) \qquad U \text{ is fresh}}{e.f_i : U \mid S_e \cup S_f \mid R_e \mid CR}$$

$$\textsc{TC-Invk}\ \frac{\begin{array}{c} e : T_e \mid S_e \mid R_e \mid CR_e \qquad \overline{e : T \mid S \mid R \mid CR} \\ CR_m = (T_e.m : \overline{U} \to U', \emptyset) \qquad \overline{S_s = \{T <: U\}} \qquad U',\ \overline{U} \text{ are fresh} \\ R'|_{S_r} = \text{merge}_R(R_e, \overline{R}) \qquad CR'|_{S_{cr}} = \text{merge}_{CR}(CR_e, CR_m, \overline{CR}) \end{array}}{e.m(\overline{e}) : U' \mid \overline{S} \cup S_e \cup \overline{S_s} \cup S_r \cup S_{cr} \mid R' \mid CR'}$$

$$\textsc{TC-New}\ \frac{\begin{array}{c} \overline{e : T \mid S \mid R \mid CR} \qquad CR_f = (C.\textbf{init}(\overline{U}), \emptyset) \qquad \overline{S_s = \{T <: U\}} \\ \overline{U} \text{ is fresh} \qquad R'|_{S_r} = \text{merge}_R(\overline{R}) \qquad CR'|_{S_{cr}} = \text{merge}_{CR}(CR_f, \overline{CR}) \end{array}}{\textbf{new}\ C(\overline{e}) : C \mid \overline{S} \cup \overline{S_s} \cup S_r \cup S_{cr} \mid R' \mid CR'}$$

$$\textsc{TC-UCast}\ \frac{e : T_e \mid S_e \mid R_e \mid CR_e \qquad S_s = \{T_e <: C\}}{(C)e : C \mid S_e \cup S_s \mid R_e \mid CR_e}$$

$$\textsc{TC-DCast}\ \frac{e : T_e \mid S_e \mid R_e \mid CR_e \qquad S_s = \{C <: T_e\} \qquad S_n = \{C \neq T_e\}}{(C)e : C \mid S_e \cup S_s \cup S_n \mid R_e \mid CR_e}$$

$$\textsc{TC-SCast}\ \frac{e : T_e \mid S_e \mid R_e \mid CR_e \qquad S_s = \{C \not<: T_e\} \qquad S'_s = \{T_e \not<: C\}}{(C)e : C \mid S_e \cup S_s \cup S'_s \mid R_e \mid CR_e}$$

$$\textsc{TC-Method}\ \frac{\begin{array}{c} e : T_e \mid S_e \mid R_e \mid CR_e \qquad \overline{S_x = \{C = R_e(x) \mid x \in \text{dom}(R_e)\}} \\ S_c = \{U_c = R_e(\textbf{this}) \mid \textbf{this} \in \text{dom}(R_e)\} \qquad S_s = \{T_e <: C_0\} \\ R_e - \textbf{this} - \overline{x} = \emptyset \qquad U_c,\ U_d \text{ are fresh} \\ CR|_{S_{cr}} = \text{merge}_{CR}(CR_e, (U_c\ .\textbf{extends}: U_d, \emptyset), (U_d.m : \overline{C} \to C_0, \emptyset)_{opt}) \end{array}}{C_0\ m(\overline{C}\ \overline{x})\ \{\textbf{return}\ e\}\ \text{OK} \mid S_e \cup S_s \cup S_c \cup S_{cr} \cup \overline{S}_x \mid U_c \mid CR}$$

$$\textsc{TC-Class}\ \frac{\begin{array}{c} K = C(\overline{D}'\ \overline{g}, \overline{C}'\ \overline{f})\{\textbf{super}(\overline{g}); \textbf{this}.\overline{f} = \overline{f}\} \qquad \overline{M\ \text{OK} \mid S \mid U \mid CR} \\ CR'|_{S_{cr}} = \text{merge}_{CR}((D.\textbf{init}(\overline{D}'), \emptyset), \overline{CR}) \qquad \overline{S_{eq} = \{U = C\}} \end{array}}{\textbf{class}\ C\ \textbf{extends}\ D\{\overline{C}\ \overline{f};\ K\ \overline{M}\}\ \text{OK} \mid \overline{S} \cup \overline{S}_{eq} \cup S_{cr} \mid CR'}$$

$$\textsc{TC-Program}\ \frac{\begin{array}{c} \overline{L\ \text{OK} \mid S \mid CR} \qquad \text{merge}_{CR}(\overline{CR}) = CR'|_{S'} \\ \biguplus_{L' \in \overline{L}}(\text{removeMs}(CR', L') \uplus \text{removeFs}(CR', L') \uplus \text{removeCtor}(CR', L') \\ \uplus\, \text{removeExt}(CR', L')) = \emptyset|_S \end{array}}{\overline{L}\ \text{OK} \mid \overline{S} \cup S' \cup S}$$

Figure 10 A co-contextual formulation of the type system of Featherweight Java.

we propagate both the constraints S_e for e and the merge constraints S_f as the resulting output constraints.

Typing rule TC-Invk is dual to T-Invk for method invocations. Similarly to field access, the dual of method lookup is introducing a requirement for the method m and merge it

with the requirements from the premises. Again, we choose fresh class variables for the method signature $\overline{U} \to U'$, marking a distinct occurrence of m. We type check the list $\overline{e}$ of parameters, adding a subtype constraint $\overline{T} <: \overline{U}$, corresponding to the subtype check in T-Invk. Finally, we merge all context and class table requirements propagated from the receiver e and the parameters $\overline{e}$, and all the constraints.

Typing rule TC-New is dual to T-New for object creation. We add a new class requirement $C.\mathbf{init}(\overline{U})$ for the constructor of class C, corresponding to the $fields$ operation in FJ. We cannot look up the fields of C in the class table, therefore we assign fresh class variables $\overline{U}$ for the constructor signature. We add the subtyping constraint $\overline{T} <: \overline{U}$ for the parameters, analogous to the subtype check in T-New. As in the other rules, we propagate a collective merge of the propagated requirement structures/constraints from the subexpressions with the newly created requirements/constraints.

Typing rules for casts, i.e., TC-UCast, TC-DCast and TC-SCast are straightforward adaptions of their contextual counterparts following the same principles. These three type rules do overlap. We do not distinguish them in the formalization, but to have an algorithmic formulation, we implement different node names for each of them. That is, typing rules for casts are syntactically distinguished.

4.2 Method Typing

The typing rule TC-Method is dual to T-Method for checking method declarations. For checking the method body, the contextual version extends the empty typing context with entries for the method parameters $\overline{x}$ and the self-reference **this**, which is implicitly in scope. Dually, we remove the requirements on the parameters and self-reference in R_e propagated from the method body. Corresponding to extending an empty context, the removal should leave no remaining requirements on the method body. Furthermore, the equality constraints $\overline{S_x}$ ensure that the annotated class types for the parameters agree with the class types in R_e.[2] This corresponds to binding the parameters to the annotated classes in a typing context. Analogously, the constraints S_c deal with the self-reference. For the latter, we need to know the associated class type, which in the absence of the class table is at this point unknown. Hence, we assign a fresh class variable U_c for the yet to be determined class containing the method declaration. The contextual rule T-Method further checks if the method declaration correctly overrides another method declaration in the superclass, that is, if it exists in the superclass must have the same type. We choose another fresh class variable U_d for the yet to be determined superclass of U_c and add appropriate supertype and optional method override requirements. We assign to the optional method requirement $U_d.m$ the type of m declared in U_c. If later is known that there exists a declaration for m in the actual type of U_d, the optional requirement is considered and equality constraints are generated. These constraints ensure that the required type of m in the optional requirement is the same as the provided type of m in the actual superclass of U_c. Otherwise this optional method requirement is invalidated and not considered. By doing so, we enable the feature of subtype polymorphism for co-contextual FJ. Finally, we add the subtype constraint ensuring that the method body's type is conforming to the annotated return type.

[2] Note that a parameter x occurs in the method body if and only if there is a requirement for x in R_e (i.e., $x \in \mathrm{dom}(R_e)$), which is due to the bottom-up propagation. The same holds for the self-reference **this**.

4.3 Class Typing

Typing rule TC-Class is used for checking class declarations. A declaration of a given class C provides definite information on the identity of its superclass D, constructor, fields, and methods signatures. Dual to the fields lookup for superclass D in T-Class, we add the constructor requirement $D.\mathbf{init}(\overline{D}')$. We merge this requirement with all requirements generated from type checking C's method declarations $\overline{M}$. Recall that typing of method m yields a distinct class variable U for the enclosing class type, because we type check each method declaration independently. Therefore, we add the constraints $\overline{\{U = C\}}$, effectively completing the method declarations with their enclosing class C.

4.4 Program Typing

Type rule TC-Program checks a list of class declarations $\overline{L}$. Class declarations of all classes provide a definite information on the identity of their super classes, constructor, fields, methods signatures. Dual to adding clauses in the class table by constructing it, we remove requirements with respect to the provided information from the declarations. Hence, dually to class table being fully extended with clauses from all class declarations, requirements are empty. The result of removing different clauses is a new set of requirement and a set of constraints. Hence, we use notation $\uplus$ to express the union of the returned tuples (requirements and constraints), i.e., $CR|_S \uplus CR'|_{S'} = CR \cup CR'|_{S \cup S'}$ After applying remove to the set of requirements, the set should be empty at this point. A class requirement is discharged from the set, either when performing remove operation (Section 3), or when it is satisfied (all conditions hold).

As shown, we can systematically derive co-contextual typing rules for Featherweight Java through duality.

5 Typing Equivalence

In this section, we prove the typing equivalence of expressions, methods, classes, and programs between FJ and co-contextual FJ. That is, (1) we want to convey that an expression, method, class and program is type checked in FJ if and only if it is type checked in co-contextual FJ, and (2) that there is a correspondence relation between typing constructs for each typing judgment.

We use σ to represent substitution, which is a set of bindings from class variables to class types ($\{U \mapsto C\}$). projExt(CT) is a function that given a class table CT returns the immediate subclass relation Σ of classes in CT. That is, $\Sigma := \{(\mathsf{C}_1, \mathsf{C}_2) \mid (\mathsf{C}_1\ \mathbf{extends}\ \mathsf{C}_2) \in CT\}$. Given a set of constraints S and a relation between class types Σ, where $\text{projExt}(CT) = \Sigma$, then the solution to that set of constraints is a substitution, i.e., $\text{solve}(S, \Sigma) = \sigma$. Also we assume that every element of the *class table*, i.e., super types, constructors, fields and methods types are class type, namely ground types. Ground types are types that cannot be substituted.

Initially, we prove equivalence for expressions. Let us first delineate the *correspondence relation*. Correspondence states that *a*) the types of expressions are the same in both formulations, *b*) provided variables in context are more than required ones in context requirements and *c*) provided class members are more than required ones. Intuitively, an expression to be well-typed in co-contextual FJ should have all requirements satisfied. Context requirements are satisfied when for all required variables, we find the corresponding bindings in context. Class table requirements are satisfied, when for all valid requirements, i.e., all

conditions of a requirement hold, we can find a corresponding declaration in a class of the same type as the required one, or in its superclasses. The relation between the class table and class requirements is formally defined in our technical report [10].

▶ **Definition 1** (Correspondence relation for expressions)**.** Given judgments $\Gamma; CT \vdash e : C$, $e : T \mid S \mid R \mid CR$, and $\text{solve}(\Sigma, S) = \sigma$, where $\text{projExt}(CT) = \Sigma$. The correspondence relation between Γ and R, CT and CR, written $(C, \Gamma, CT) \rhd \sigma(T, R, CR)$, is defined as:
(a) $C = \sigma(T)$
(b) $\Gamma \supseteq \sigma(R)$
(c) $CT\ satisfies\ \sigma(CR)$

We stipulate two different theorems to state both directions of equivalence for expressions.

▶ **Theorem 2** (Equivalence of expressions: $\Rightarrow$)**.** *Given* $e,\ C,\ \Gamma,\ CT$, *if* $\Gamma; CT \vdash e : C$, *then there exists* $T,\ S,\ R,\ CR,\ \Sigma,\ \sigma$, *where* $\text{projExt}(CT) = \Sigma$ *and* $\text{solve}(\Sigma, S) = \sigma$, *such that* $e : T \mid S \mid R \mid CR$ *holds,* σ *is a ground solution and* $(C, \Gamma, CT) \rhd \sigma(T, R, CR)$ *holds.*

▶ **Theorem 3** (Equivalence of expressions: $\Leftarrow$)**.** *Given* $e,\ T,\ S,\ R,\ CR,\ \Sigma$, *if* $e : T \mid S \mid R \mid CR$, $\text{solve}(\Sigma, S) = \sigma$, *and* σ *is a ground solution, then there exists* $C,\ \Gamma,\ CT$, *such that* $\Gamma; CT \vdash e : C$, $(C, \Gamma, CT) \rhd \sigma(T, R, CR)$ *and* $\text{projExt}(CT) = \Sigma$.

Theorems 2 and 3 are proved by induction on the typing judgment of expressions. The most challenging aspect consists in proving the relation between the class table and class table requirements. In Theorem 2, the class table is given and the requirements are a collective merge of the propagated requirement from the subexpressions with the newly created requirements. In Theorem 3, the class table is not given,therefore we construct it through the information retrieved from *ground class requirements*. We ensure class table correctness and completeness with respect to the given requirements. First, we ensure that the class table we construct is correct, i.e., types of **extends**, fields, and methods clauses we add in the class table are equal to the types of the same **extends**, fields, and methods if they already exist in the class table. Second, we ensure that the class table we construct is complete, i.e., the constructed class table satisfies all given requirements.

Next, we present the theorem of equivalence for methods. The difference from expressions is that there is no context, therefore no relation between context and context requirements is required. Instead, the fresh class variable introduced in co-contextual FJ as a placeholder for the actual class, where the method under scrutiny is type checked in, after substitution should be the same as the class where the method is type checked in FJ.

▶ **Theorem 4** (Equivalence of methods: $\Rightarrow$)**.**
Given $m,\ C,\ CT$, *if* $C; CT \vdash C_0\ m(\overline{C}\ \overline{x})\{return\ e\}$
OK, *then there exists* $S,\ T,\ CR,\ \Sigma,\ \sigma$, *where* $\text{projExt}(CT) = \Sigma$ *and* $\text{solve}(\Sigma, S) = \sigma$, *such that*
$C_0\ m(\overline{C}\ \overline{x})\ \{return\ e_0\}\ OK \mid S \mid T \mid CR$ *holds,* σ *is a ground solution and*
$(C, CT) \rhd_m \sigma(T, CR)$ *holds.*

▶ **Theorem 5** (Equivalence of methods: $\Leftarrow$)**.**
Given $m,\ T,\ S,\ CR,\ \Sigma$, *if* $C_0\ m(\overline{C}\ \overline{x})\ \{return\ e_0\}$
$OK \mid S \mid T \mid CR$, $\text{solve}(\Sigma, S) = \sigma$, *and* σ *is a ground solution, then there exists* $C,\ CT$, *such that* $C; CT \vdash C_0\ m(\overline{C}\ \overline{x})\{return\ e\}\ OK$ *holds,* $(C, CT) \rhd_m \sigma(T, CR)$ *and* $\text{projExt}(CT) = \Sigma$.

Theorems 5 and 6 are proved by induction on the typing judgment. The difficulty increases in proving equivalence for methods because we have to consider the optional requirement,

as introduced in the previous sections. It requires a different strategy to prove the relation between the class table and optional requirements; we accomplish the proof by using case distinction. We have a detailed proof for method declaration, and also how this affects class table construction, and we prove a correct and complete construction of it.

Lastly, we present the theorem of equivalence for classes and programs.

▶ **Theorem 6** (Equivalence of classes: $\Rightarrow$). *Given* C, CT, *if* $CT \vdash$ *class* C *extends* D $\{\overline{C}\ \overline{f}; K\ \overline{M}\}$ OK, *then there exists* S, CR, Σ, σ, *where* $\text{projExt}(CT) = \Sigma$ *and* $\text{solve}(\Sigma, S) = \sigma$, *such that class* C *extends* $D\{\overline{C}\ \overline{f};\ K\ \overline{M}\}$ $OK \mid S \mid CR$ *holds,* σ *is a ground solution and* $(CT) \rhd_c \sigma(CR)$ *holds.*

▶ **Theorem 7** (Equivalence of classes: $\Leftarrow$). *Given* C, CR, Σ, *if class* C *extends* $D\{\overline{C}\ \overline{f};\ K\ \overline{M}\}$ $OK \mid S \mid CR$, $\text{solve}(\Sigma, S) = \sigma$, *and* σ *is a ground solution, then there exists* CT, *such that* $CT \vdash$ *class* C *extends* D $\{\overline{C}\ \overline{f}; K\ \overline{M}\}$ OK *holds,* $(CT) \rhd_c \sigma(CR)$ *holds and* $\text{projExt}(CT) = \Sigma$.

Theorems 8 and 9 are proved by induction on the typing judgment. Class declaration requires to prove only the relation between the class table and class table requirements since there is no context.

Typing rule for programs does not have as inputs context and class table, therefore there is no relation between context, class table and requirements. The equivalence theorem describes that a program in FJ and co-contextual FJ is well-typed.

▶ **Theorem 8** (Equivalence for programs: $\Rightarrow$). *Given* $\overline{L}$, *if* $\overline{L}$ OK, *then there exists* S, Σ, σ, *where* $\text{projExt}(\overline{L}) = \Sigma$ *and* $\text{solve}(\Sigma, S) = \sigma$, *such that* $\overline{L}$ $OK \mid S$ *holds and* σ *ground solution.*

▶ **Theorem 9** (Equivalence for programs: $\Leftarrow$). *Given* $\overline{L}$, *if* $\overline{L}$ $OK \mid S$, $\text{solve}(\Sigma, S) = \sigma$, *where* $\text{projExt}(\overline{L}) = \Sigma$, *and* σ *is a ground solution, then* $\overline{L}$ OK *holds.*

Theorems 10 and 11 are proved by induction on the typing judgment. In here, we prove that a class table containing all clauses provided from the given class declarations is dual to empty class table requirements in the inductive step.
We refer to our technical report [10] for omitted definitions, lemmas, and proofs.

6 Efficient Incremental FJ Type Checking

The co-contextual FJ model from Section 3 and 4 was designed such that it closely resembles the formulation of the original FJ type system, where all differences are motivated by dually replacing contextual operations with co-contextual ones. As such, this model served as a good basis for the equivalence proof from the previous section. However, to obtain a type checker implementation for co-contextual FJ that is amenable to efficient incrementalization, we have to employ a number of behavior-preserving optimizations. In the present section, we describe these optimization and the resulting *incremental* type checker implementation for co-contextual FJ. The source code is available online at `https://github.com/seba--/incremental`.

Condition normalization. In our formal model from Section 3 and 4, we represent context requirements as a set of conditional class requirements $CR \subset \mathit{Creq} \times \mathit{cond}$. Throughout type checking, we add new class requirements using function merge, but we only discharge class requirements in rule TC-Program at the very end of type checking. Since merge generates

$3 * m * n$ conditional requirements for inputs with m and n requirements respectively, requirements quickly become intractable even for small programs.

The first optimization we conduct is to eagerly normalize conditions of class requirements. Instead of representing conditions as a list of type equations and inequations, we map receiver types to the following condition representation (shown as Scala code):

```
case class Condition(notGround: Set[CName], notVar: Set[UCName],
                     sameVar: Set[UCName], sameGroundAlternatives: Set[CName]).
```

A condition is true if the receiver type is different from all ground types (CName) and unification variables (UCName) in notGround and notVar, if the receiver type is equal to all unification variables in sameVar, and if sameGroundAlternatives is either empty or the receiver type occurs in it. That is, if sameGroundAlternatives is non-empty, then it stores a set of alternative ground types, one of which the receiver type must be equal to.

When adding an equation or inequation to the condition over a receiver type, we check whether the condition becomes unsatisfiable. For example, when equating the receiver type to the ground type C and notGround.contains(C), we mark the resulting condition to be unsatisfiable. Recognizing unsatisfiable conditions has the immediate benefit of allowing us to discard the corresponding class requirements right away. Unsatisfiable conditions occur quite frequently because merge generates both equations and inequations for all receiver types that occur in the two merged requirement sets.

If a condition is not unsatisfiable, we normalize it such that the following assertions are satisfied: (i) the receiver type does not occur in any of the sets, (ii) sameGroundAlternatives.isEmpty || notGround.isEmpty, and (iii) notVar.intersect(sameVar).isEmpty. Since normalized conditions are more compact, this optimization saves memory and time required for memory management. Moreover, it makes it easy to identify irrefutable conditions, which is the case exactly when all four sets are empty, meaning that there are no further requisites on the receiver type. Such knowledge is useful when merge generates conditional constraints, because irrefutable conditions can be ignored. Finally, condition normalization is a prerequisite for the subsequent optimization.

In-depth merging of conditional class requirements. In the work on co-contextual PCF [6], the number of requirements of an expression was bound by the number of free variables that occur in that expression. To this end, the merge operation used for co-contextual PCF identifies subexpression requirements on the same free variable and merges them into a single requirement. For example, the expression $x + x$ has only one requirement $\{x : U_1\}|_{\{U_1=U_2\}}$, even though the two subexpressions propagate two requirements $\{x : U_1\}$ and $\{x : U_2\}$, respectively.

Unfortunately, the merge operation of co-contextual FJ given in Section 3.2 does not enjoy this property. Instead of merging requirements, it merely collects them and updates their conditions. A more in-depth merge of requirements is possible whenever two code fragments require the same member from the same receiver type. For example, the expression $\mathbf{this}.x + \mathbf{this}.x$ needs only one requirement $\{U_1.x() : U_2\}|_{\{U_1=U_3, U_2=U_4\}}$, even though the two subexpressions propagate two requirements $\{U_1.x() : U_2\}$ and $\{U_3.x() : U_4\}$, respectively. Note that $U_1 = U_3$ because of the use of **this** in both subexpressions, but $U_2 = U_4$ because of the in-depth merge.

However, conditions complicate the in-depth merging of class requirements: We may only merge two requirements if we can also merge their conditions. That is, for conditional requirements $(creq_1, cond_1)$ and $(creq_2, cond_2)$ with the same receiver type, the merged requirement

must have the condition $cond_1 \vee cond_2$. In general, we cannot express $cond_1 \vee cond_2$ using our Condition representation from above because all fields except sameGroundAlternatives represent conjunctive prerequisites, whereas sameGroundAlternatives represents disjunctive prerequisites. Therefore, we only support in-depth merging when the conditions are identical up to sameGroundAlternatives and we use the union operator to combine their sameGroundAlternatives fields.

This optimization may seem a bit overly specific to certain use cases, but it turns out it is generally applicable. The reason is that function removeExt creates requirements of the form $(D.f : T', cond \cup (T = C_i))$ transitively for all subclasses C_i of D where no class between C_i and D defines field f. Our optimization combines these requirements into a single one, roughly of the form $(D.f : T', cond \cup (T = \bigvee_i C_i))$. Basically, this requirement concisely states that D must provide a field f of type T' if the original receiver type T corresponds to any of the subclasses C_i of D.

Incrementalization and continuous constraint solving. We adopt the general incrementalization strategy from co-contextual PCF [6]: Initially, type check the full program bottom-up and memoize the typing output for each node (including class requirements and constraint system). Then, upon a change to the program, recheck each node from the change to the root of the program, reusing the memoized results from unchanged subtrees. This way, incremental type checking asymptotically requires only $\log n$ steps for a program with n nodes.

In our formal model of co-contextual FJ, we collect constraints during type checking and solve them at the end to yield a substitution for the unification variables. As was discussed by Erdweg et al. for co-contextual PCF [6], this strategy is inadequate for incremental type checking, because we would memoize unsolved constraints and thus only obtain an incremental constraint generator, but even a small change would entail that all constraints had to be solved from scratch. In our implementation, we follow Erdweg et al.'s strategy of continuously solving constraints as soon as they are generated, memoizing the resulting partial constraint solutions. In particular, equality constraints that result from merge and remove operations can be solved immediately to yield a substitution, while subtype constraints often have to be deferred until more information about the inheritance hierarchy is available. In the context of FJ with its nominal types, continuous constraint solving has the added benefit of enabling additional requirement merging, for example, because two method requirements share the same receiver type after substitution.

Tree balancing. Even with continuous constraint solving, co-contextual FJ as defined in Section 4 still does not yield satisfactory incremental performance. The reason is that the syntax tree is deformed due to the root node, which consists of a sequence of *all* class declarations in the program. Thus, the root node has a branching factor only bound by the number of classes in the program, whereas the rest of the tree has a relative small branching factor bound by the number of arguments to a method. Since incremental type checking recomputes each step from the changed node to the root node, the type checker would have to repeat discharging class requirements at the root node after every code change, which would seriously impair incremental performance.

To counter this effect, we apply tree balancing as our final optimization. Specifically, instead of storing the class declarations as a sequence in the root node, we allow sequences of class declarations to occur as inner nodes of the syntax tree:

$$L \quad ::= \quad \overline{L} \;\mid\; \textbf{class}\ C\ \textbf{extends}\ D\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\}$$

This allows us to layout a program's class declarations structurally as in $((((C_1\ C_2)\ C_3)\ (C_4\ C_5))\ (C_6\ C_7))$, thus reducing the costs for rechecking any path from a changed node to the root node. As part of this optimization, to satisfy requirements of classes that occur in different tree nodes such as C_1 and C_6, we also neeed to propagate *class facts* such as actual method signatures upwards. As consequence, we can now link classes in any order without changing the type checking result.

We have implemented an incremental co-contextual FJ type checker in Scala using the optimizations described here. In the following section, we present our run-time performance evaluation.

7 Performance Evaluation

We have benchmarked the initial and incremental run-time performance of co-contextual FJ implementation. However, this evaluation makes no claim to be complete, but rather is intended to confirm the feasibility and potential of co-contextual FJ for incremental type checking.

7.1 Evaluation on synthesized FJ programs

Input data. We synthesized FJ programs with 40 root classes that inherit from Object. Each root class starts a binary tree in the inheritance hierarchy of height 5. Thus, each root-class hierarchy contains 31 FJ class declarations. In total, our synthesized programs have $31 * 40 + 3 = 1243$ class declarations, since we always require classes for natural numbers Nat, Zero, and Succ.

Each class has at least a field of type Nat and each class has a single method that takes no arguments and returns a Nat. We generated the method body according to one of three schemes:

- *AccumSuper*: The method adds the field's value of this class to the result of calling the method of the super class.
- *AccumPrev*: Each class in root hierarchy $k > 1$ has an additional field that has the type of the class at the same position in the previous root hierarchy $k - 1$. The method adds the field's value of this class to the result of calling the method of the class at the same position in the previous root hierarchy $k - 1$, using the additional field as receiver object.
- *AccumPrevSuper*: Combines the other two schemes; the method adds all three numbers.

We also varied the names used for the generated fields and methods:

- *Unique*: Every name is unique.
- *Mirrored*: Root hierarchies use the same names in the same classes, but names within a single root hierarchy are unique.
- *Override*: Root hierarchies use different names, but all classes within a single root hierarchy use the same names for the same members.
- *Mir+Over*: Combines the previous two schemes, that is, all classes in all root hierarchies use the same names for the same members.

For evaluating the incremental performance, we invalidate the memoized results for the three Nat classes. This is a critical case because all other classes depend on the Nat classes and a change is traditionally hard to incrementalize.

Table 1 Performance measurement results with $k = 40$ root classes in **Milliseconds**. Numbers in parentheses indicate speedup relative to (javac/contextual) base lines.

Super	javac / contextual	co-contextual init	co-contextual inc
unique	70.00 / 93.99	3117.73 (0.02x / 0.03x)	23.44 (2.9x / 4x)
mirrored	68.03 / 88.73	1860.18 (0.04x / 0.05x)	15.17 (4.5x / 6x)
override	73.18 / 107.83	513.44 (0.14x / 0.21x)	16.92 (4.3x / 6x)
mir+over	72.64 / 132.09	481.07 (0.15x / 0.27x)	16.60 (4.4x / 8x)
Prev	javac / contextual	co-contextual init	co-contextual inc
unique	82.16 / 87.66	3402.28 (0.02x / 0.02x)	23.43 (3.5x / 4x)
mirrored	81.19 / 84.94	2136.42 (0.04x / 0.04x)	15.46 (5.3x / 5x)
override	81.51 / 120.60	840.14 (0.09x / 0.14x)	17.37 (4.7x / 7x)
mir+over	79.71 / 120.46	816.16 (0.09x / 0.15x)	16.61 (4.8x / 7x)
PrevSuper	javac / contextual	co-contextual init	co-contextual inc
unique	93.12 / 104.03	6318.69 (0.01x / 0.02x)	26.26 (3.5x / 4x)
mirrored	95.41 / 100.00	5014.12 (0.02x / 0.02x)	15.71 (6.1x / 6x)
override	92.88 / 130.01	3601.44 (0.03x / 0.04x)	17.35 (5.4x / 7x)
mir+over	93.37 / 126.57	3579.90 (0.03x / 0.04x)	16.61 (5.6x / 8x)

Experimental setup. First, we measured the wall-clock time for the initial check of each program using our co-contextual FJ implementation. Second, we measured the wall-clock time for the incremental reanalysis after invalidating the memoized results of the three `Nat` classes. Third, we measured the wall-clock time of checking the synthesized programs on javac and on a straightforward implementation of contextual FJ for comparison. Contextual FJ is the standard FJ described in Section 2, that uses contexts and class tables during type checking. Our implementation of contextual FJ is up to 2-times slower than javac, because it is not production quality. We used ScalaMeter[3] to take care of JIT warm-up, garbage-collection noise, etc. All measurements were conducted on a 3.1GHz duo-core MacBook Pro with 16GB memory running the Java HotSpot 64-Bit Server VM build 25.111-b14 with 4GB maximum heap space. We confirmed that confidence intervals were small.

Results. We show the measurement results in table 1. All numbers are in milliseconds. We also show the speedups of initial and incremental run of co-contextual type checking relative to both javac and contextual type checking.

As this data shows, the initial performance of co-contextual FJ is subpar: The initial type check takes up to 68-times and 61-times longer than using javac and a standard contextual checker respectively.

However, co-contextual FJ consistently yields high speedups for incremental checks. In fact, it only takes between 3 and 21 code changes until co-contextual type checking is faster overall. In an interactive code editing session where every keystroke or word could be considered a code change, incremental co-contextual type checking will quickly break even and outperform a contextual type checker or javac.

The reason that the initial run of co-contextual FJ induces such high slowdowns is because the occurrence of class requirements is far removed from the occurrence of the

[3] `https://scalameter.github.io/`

corresponding class facts. This is true for the Nat classes that we merge with the synthesized code at the top-most node as well as for dependencies from one root hierarchy to another one. Therefore, the type checker has to propagate and merge class requirements for a long time until finally discovering class facts that discharge them. We conducted an informal exploratory experiment that revealed that the performance of the initial run can be greatly reduced by bringing requirements and corresponding class facts closer together. On the other hand, incremental performance is best when the changed code occurs as close to the root node as possible, such that a change entails fewer rechecks. In future work, when scaling our approach to full Java, we will explore different layouts for class declarations (e.g., following the inheritance hierarchy or the package structure) and for reshuffling the layout of class declarations during incremental type checking in order to keep frequently changing classes as close to the root as possible.

7.2 Evaluation on real Java program

Input data. We conduct an evaluation for our co-contextual type checking on realistic FJ programs. We wrote about 500 SLOCs in Java, implementing purely functional data structures for binary search trees and red black trees. In the Java code, we only used features supported by FJ and mechanically translated the Java code to FJ. For evaluating the incremental performance, we invalidate the memoized results for the three Nat classes as in the experiment above.

Experimental setup. Same as above.

Results. We show the measurements in milliseconds for the 500 lines of Java code.

javac / contextual	co-contextual init	co-contextual inc
14.88 / 3.74	48.07 (0.31x / 0.08x)	9.41 (1.6x / 0.39x)

Our own non-incremental contextual type checker is surprisingly fast compared to javac, and not even our incremental co-contextual checker gets close to that performance. When comparing javac and the co-contextual type checker, we observe that the initial performance of the co-contextual type checker improved compared to the previous experiment, whereas the incremental performance degraded. While the exact cause of this effect is unclear, one explanation might be that the small input size in this experiment reduces the relative performance loss of the initial co-contextual check, but also reduces the relative performance gain of the incremental co-contextual check.

8 Related work

The work presented in this paper on co-contextual type checking for OO programming languages, specifically for Featherweight Java, is inspired by the work on co-contextual type checking for PCF [6]. OO languages and FJ impose features like nominal typing, subtype polymorphism, and inheritance that are not covered in the work for co-contextual PCF [6]. In particular, here we developed a solution for merging and removing requirements in presence of nominal typing.

Introducing type variables as placeholders for the actual types of variables, classes, fields, methods is a known technique in type inference [11, 12]. The difference is that we introduce a fresh class variable for each occurrence of a method m or fields in different branches of the

typing derivation. Since fresh class variables are generated independently, no coordination is required while moving up the derivation tree, ensuring context and class table independence. Type inference uses the context to coordinate type checking of m in different branches, by using the same type variable. In contrast to type inference where context and class table are available, we remove them (no actual context and class table). Hence, in type inference inheritance relation between classes and members of the classes are given, whereas in co-contextual FJ we establish these relations through requirements. That is, classes are required to have certain members with unknown types and unknown inheritance relation, dictated from the surrounding program.
Also, in contrast to bidirectional type checking [4, 5] that uses two sets of typing rules one for inference and one for checking, we use one set of co-contextual type rules, and the direction of type checking is all oriented bottom-up; types and requirements flow up. As in type inference, bidirectional type checking uses context to look up variables. Whereas co-contextual FJ has no context or class table, it uses requirements as a structure to record the required information on fields, methods, such that it enables resolving class variables of the required fields, methods to their actual types.

Co-contextual formulation of type rules for FJ is related to the work on principal typing [9, 17], and especially to principal typing for Java-like languages [2]. A principal typing [2] of each fragment (e.g., modules, packages, or classes) is associated with a set of type constraints on classes, which represents all other possible typings and can be used to check compatibility in all possible contexts. That is, principle typing finds the strongest type of a source fragment in the weakest context. This is used for type inference and separate compilation in FJ. They can deduce exact type constraints using a type inference algorithm. We generalize this and do not only infer requirements on classes but also on method parameters and the current class. Moreover, we developed a duality relation between the class table and class requirements that enables the systematic development of co-contextual type systems for OO languages beyond FJ.

Related to our co-contextual FJ is the formulations used in the context of compositional compilation [1] (continuation of the work on principal typing [2]) and the compositional explanation of type errors [3]. This type system [1] partially eliminate the class table, namely only inside a fragment, and does not eliminate the context. Hence, type checking of parameters and **this** is coordinated and subexpressions are coupled through dependencies on the usage of context. In our work, we eliminate both class table (not only partially) and context, therefore all dependencies are removed. By doing so we can enable compositional compilation for individual methods. To resolve the type constraints on classes, compositional compilation [2] needs a linker in addition to an inference algorithm (to deduce exact type constraints), whereas, we use a constraint system and requirements. We use duality to derive a co-contextual type system for FJ and we also ensure that both formulations are equivalent (5). That is, we ensure that an expression, method, class, or program is well-typed in FJ if and only if it is well-typed in co-contextual FJ, and that all requirements are fulfilled. In contrast, compositional compilation rules do not check whether the inferred collection of constraints on classes is satisfiable; they actually allow to derive judgments for any fragment, even for those that are not statically correct.

Refactoring for generalization using type constraints [16, 15] is a technique Tip et al. used to manipulate types and class hierarchies to enable refactoring. That work uses variable type constraints as placeholders for changeable declarations. They use the constraints to restrict when a refactoring can be performed. Tip et al. are interested to find a way to represent the actual class hierarchy and to use constraints to have a safe refactoring and a

well-typed program after refactoring. The constraint system used by Tip et al. is specialized to refactoring, because different variable constraints and solving techniques are needed In contrast, in our work, we use class variables as placeholders for the actual type of required extends, constructors, fields, and methods of a class, in the lack of the class table. We want to gradually learn the information about the class hierarchy. We are interested in the type checking technique and how to co-contextualize it and use constraints for type refinement.

Adapton [7] is a programming language where the runtime system traces memory reads and writes and selectively replays dependent computations when a memory change occurs. In principle, this can be used to define an "incremental" contextual type checker. However, due to the top-down threading of the context, most of the computation will be sensitive to context changes and will have to be replayed, thus yielding unsatisfactory incremental performance. Given a co-contextual formulation as developed in our paper, it might be possible to define an efficient implementation in Adapton.

The works on smart/est recompilation [13, 14] have a different purpose from ours, namely to achieve separate compilation they need algorithms for the inference and also the linking phase specific to SML. In contrast, we use duality as a guiding principle to enable the translation from FJ to co-contextual FJ. This technique allows us to do perform a systematic (but yet not mechanical) translation from a given type system to the co-contextual one. Our type system facilitates incremental type checking because we decouple the dependencies between subexpressions and the smallest unit of compilation is any node in the syntax tree. Moreover, we have investigated optimizations for facilitating the early solving of requirements and constraints.

9 Conclusion and Future Work

In this paper, we presented a co-contextual type system for FJ by transforming the typing rules in the traditional formulation into a form that replaces top-down propagated contexts and class tables with bottom-up propagated *context and class table requirements*. We used duality as a technique to derive co-contextual FJ's typing rules from FJ's typing rules. To make the correspondence between class table and requirements, we presented class tables that are gradually extended with information from the class declarations, and how to map operations on contexts and class tables to their dual operations on context and class table requirements. To cover the OO features of nominal typing, subtype polymorphism, and implementation inheritance, co-contextual FJ uses conditional requirements, inequality conditions, and conditional constraints. Also, it changes the set of requirements by adding requirements with the different receiver from the ones defined by the surrounding program, in the process of merging and removing requirements as the type checker moves upwards and discovers class declarations. We proved the typing equivalence of expressions, methods, classes, and programs between FJ and co-contextual FJ.

The co-contextual formulation of FJ typing rules enables incremental type checking because it removes dependencies between subexpressions. We implemented an incremental co-contextual FJ type checker. Also, we evaluated its performance on synthesized programs up to 1243 FJ classes and 500 SLOCs of java programs.

There are several interesting directions for future work. In short term, we want to explore parallel co-contextual type checking for FJ. A next step would be to develop a co-contextual type system for full Java. Another interesting direction is to investigate co-contextual formulation for gradual type systems.

References

1 Davide Ancona, Ferruccio Damiani, Sophia Drossopoulou, and Elena Zucca. Polymorphic bytecode: Compositional compilation for Java-like languages. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, 2005. doi:10.1145/1040305.1040308.

2 Davide Ancona and Elena Zucca. Principal typings for Java-like languages. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, 2004. doi:10.1145/964001.964027.

3 Olaf Chitil. Compositional explanation of types and algorithmic debugging of type errors. In *Proceedings of International Conference on Functional Programming (ICFP)*, 2001. doi:10.1145/507635.507659.

4 David Raymond Christiansen. Bidirectional typing rules: A tutorial, 2013.

5 Joshua Dunfield and Neelakantan R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *Proceedings of International Conference on Functional Programming (ICFP)*, 2013. doi:10.1145/2500365.2500582.

6 Sebastian Erdweg, Oliver Bračevac, Edlira Kuci, Matthias Krebs, and Mira Mezini. A co-contextual formulation of type rules and its application to incremental type checking. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2015. doi:10.1145/2814270.2814277.

7 Matthew A. Hammer, Khoo Yit Phang, Michael Hicks, and Jeffrey S. Foster. Adapton: Composable, demand-driven incremental computation. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*, 2014. doi:10.1145/2666356.2594324.

8 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *Transactions on Programming Languages and Systems (TOPLAS)*, 2001. doi:10.1145/503502.503505.

9 Trevor Jim. What are principal typings and what are they good for? In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, 1996. doi:10.1145/237721.237728.

10 Edlira Kuci, Sebastian Erdweg, Oliver Bračevac, Andi Bejleri, and Mira Mezini. A co-contextual type checker for Featherweight Java (incl. proofs). *CoRR*, abs/1705.05828, 2017.

11 Benjamin C. Pierce. *Types and programming languages*. MIT press, 2002.

12 Benjamin C. Pierce and David N. Turner. Local type inference. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, 1998. doi:10.1145/268946.268967.

13 Zhong Shao and Andrew W. Appel. Smartest recompilation. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, 1993. doi:10.1145/158511.158702.

14 Walter F. Tichy. Smart recompilation. *Transactions on Programming Languages and Systems (TOPLAS)*, 1986. doi:10.1145/5956.5959.

15 Frank Tip, Robert M. Fuhrer, Adam Kieżun, Michael D. Ernst, Ittai Balaban, and Bjorn De Sutter. Refactoring using type constraints. *Transactions on Programming Languages and Systems (TOPLAS)*, 2011. doi:10.1145/1961204.1961205.

16 Frank Tip, Adam Kiezun, and Dirk Bäumer. Refactoring for generalization using type constraints. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2003. doi:10.1145/949305.949308.

17 J. B. Wells. The essence of principal typings. In *Proceedings of International Colloquium on Automata, Languages and Programming (ICALP)*, 2002. doi:10.1007/3-540-45465-9_78.