Toward Incremental Type Checking for Java

Edlira Kuci Sebastian Erdweg Mira Mezini
TU Darmstadt, Germany

Abstract

A type system is a set of type rules and with respect to these type rules a type checker has an important role to ensure that programs exhibit a desired behavior. We consider Java type rules and extend the co-contextual formulation of type rules introduced in [1] to enable it for Java. Regarding the extension type rules result is a type, a set of *context requirements* and a set of *class requirements*. Since *context* and *class requirements* are propagated bottom-up and while traversing the syntax tree bottom-up and are merged from independent subexpression, this enables the type system to be incremental therefore the performance is increased.

Categories and Subject Descriptors F.3.1 [*Specifying and Verifying and Reasoning about Programs*]; F.3.2 [*Semantics of Programming Languages*]: Program analysis

Keywords type checking; incremental; co-contextual; constraints; class table; Java

1. Introduction

Type systems build context information with a top-down traversal of the syntax tree from a program. The top-down propagation of typing contexts has some problems. As the type checker traverses the syntax tree top-down, typing context is extended and it coordinates type checking of independent subexpressions. As a consequence it hampers incrementalization of type checking, because typing subexpression depends on typing parent expressions and vice versa.

We consider Featherweight Java as a core calculus of Java [2] to construct incremental Java type system from our proposed method. In addition to the typing context of simple type systems, Featherweight Java uses a class table during type checking as part of the type rules. The class table contains all the necessary information about the defined classes.

SPLASH Companion'15, October 25–30, 2015, Pittsburgh, PA, USA
ACM. 978-1-4503-3722-9/15/10
http://dx.doi.org/10.1145/2814189.2817272

To enable type checkers with incrementalization, dependencies that occur between sub-expression, imposed from the usage of the typing context and the class table, should be removed. In our proposed method, we eliminate typing context and the class table. We propose bottom-up propagated *context requirements* as a dual concept of the typing context. Instead of looking up the variable in the context or the field in the class table we generate a fresh class name (annotated with metavariable U) as a placeholder for their actual types. The same idea is used for class requirements, which are the dual concept of a class table. Instead of having a field (method) lookup in the class table, we generate a new requirement stating that the class need to have the given field (method) of a freshly generated class name. We call this approach co-contextual type checking. Co-contextual type checking moves bottom-up (from the leaves up to the root). As a consequence the information starts to collected from the leaves. The type checker refines the types of the sub-expressions and merges the context requirements and adds class requirements constantly until we reach the root of the tree.

Let us consider a simple example in java and delineate how standard and co-contextual type checking would apply to it.

```
class Pair{
        Int first;
        Int second;
        }
(new Pair(1,2)).first
```

In Featherweight Java type checking a field access *(new Pair(1,2)).first* is just a lookup in the class table, where the type of *first* is Int. On the other hand, co-contextual type checking does not have that information since the class table does not exist. As a result, we do not know the type of the field *first*, therefore we bind *first* to a fresh class name U. This generates a new class requirement of the form, *CR = Pair hasField first: U*.

In the following we present the typing rule field access. First we show the typing judgment. We suppose the original typing judgment has the form $CT, \Gamma \vdash e : T \mid C$, where CT is the class table, Γ is the typing context, e is the expression under analysis, and T is the type of e if all type constraints in set C hold. For reference, we use the constraint-based contextual type rule of Featherweight Java for field access.

$$\text{T-FIELD}\ \frac{e_0 : T_0 \mid C \mid R \mid CR_0 \qquad U_0 \textit{ is fresh} \qquad add(CR_0, (T_0\ hasField\ f : U_0)) = CR|_{C_r}}{e_0.f : U_0 \mid C \cup C_r \mid R \mid CR}$$

Figure 1. A co-contextual constraint-based formulation of the field invocation.

$$\text{T-FIELD}\ \frac{CT, \Gamma \vdash e_0 : T_0 \mid C \qquad fields(CT, T_0) = \bar{T}\bar{f}}{CT, \Gamma \vdash e_0.f_i : T_i \mid C}$$

Co-contextual type rules use judgment of the form $e : T \mid C \mid R \mid CR$, where e is the expression under analysis and T is the type of e if all type constraints in set C hold and all context requirements in set R and class requirements in set CR are satisfied.

To co-contextualize field access type rule we have to define it without the context and the class table. As a consequence we have no information for the type of the field f, which in the standard type rule is a class table lookup of f, yielding class T_i. In lack of a class table, we introduce fresh class names as mentioned above and associate them with variables or fields. Later on when more information for variables or fields is available we refine their types retroactively using type constraints and unification. As stated previously we do not have a class table, therefore we do not use the usual functions of standard type checking of field lookup (*fields(CT,T)*) and method lookup. Instead we have the dual operations for field lookup and method lookup, i.e., adding a new requirement to the class requirements. Figure 1 shows the type rule for field access in the co-contextual Featherweight Java type system. For the constraints we have union $\cup$ of them. *fields(CT, T)* is translated to new requirement *(T_0 hasField f: U_0)*, which is added to the rest of the class requirements obtained from the already type checked expression e_0. All context/class requirements collected from different sub-expressions are propagated up to the syntax tree and meanwhile merged/added if possible. The result of the merging/adding is a new set of requirements and new constraints.

The context requirements need to assure that all variables, fields and methods get assigned to the same type. Therefore we have the auxiliary function $merge$ to identify the overlapping requirements in the context requirement set. The $merge$ function is the same as the one presented in [1]. On the other hand adding of class requirements gets more complicated. Class requirements are mappings from classes C to its corresponding class declaration cld. Where a class declaration is a triple of super class, fields and methods $cld = (superClass, List[Field], list[Method])$. To realize adding of the class requirements we first find the class declarations for matching classes with the same name, then perform adding internally to every part of the triple. To summarize we have:

- Adding of super classes
- Adding for all equally-named fields
- Adding for equally-named methods
  - Adding of the return type
  - Adding for all equally-named parameters
  - Adding of the body requirements

Requirements when reaching the root of the syntax tree should be satisfied to have a well-typed program. Satisfaction of requirements is related when the actual type of variables or fields is available. For instance, we get the actual type for a variable or a field (the user defined type), all fresh class names assigned to them are unified with the actual type and the requirements corresponding to them are removed. That is, when in standard type system the context is extended with a new binding for the field or the variable, as a dual operation to it we have a removal from the context requirement set, meaning that the requirement is satisfied.

We can systematically construct co-contextual type rules for Featherweight Java from constraint-based contextual type rules. A co-contextual type-system, by decoupling the dependencies between sub-expressions, enables incrementalization of the type checker, which can improve the performance during type checking significantly.

References

[1] S. Erdeweg, O. Bračevac, E. Kuci, M. Krebs, and M. 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. to appear.

[2] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight java: A minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001.