A Java-like calculus with user-defined coeffects

Riccardo Bianchini, Francesco Dagnino, Paola Giannini, Elena Zucca

Abstract

Modern applications are thought to be resource-aware, so it is very useful to focus on the concept of resource and to keep track of them. Coeffect systems provide a static control capable to guarantee interesting properties on the usage of tracked objects. Our goal is to develop a Java-like calculus where declared variables can be annotated by coeffects specifying constraints on their use, such as linearity or security levels. Such annotations are written in the language itself, as expressions of TypeCoef. A predefined class which can be extended by user-defined subclasses, modeling coeffects desired for a specific application. We will also show a simple example of coeffect system checking the linear use of a variable.

Coeffects

Coeffect systems are, in a sense, the dual of effect systems. The latter track what the program requires from the context of a computation. There are two kinds of coeffect systems:

• Structural coeffects. That is, coeffect annotations annotate each variable in the context independently

• Flat coeffects annotate the whole context

Here we consider structural coeffects.

Structure of coeffects

Coeffects are assumed to form a semiring, that is, a tuple (C, +, 0, ×, 1) such that

• (C, +, 0) is a commutative monoid
• (C, ×, 1) is a monoid
• Given c ∈ C
• 0 + c = c + 0 = 0

Coeffect annotations

Our Java-like calculus supports user-defined structural coeffects. That is, coeffect annotations are values of (subclasses of) a predefined class Coeffect, analogously to Java exceptions. Coeffects can be seen as constraints on the use of declared variables, so with user-defined coeffects we can impose user-defined constraints. Metavaraible e is used for used-defined coeffects, that is, values expected to be of a subclass of Coef.

Properties of user-defined coeffects

To guarantee preservation of coeffects during execution, operators determined by user-defined methods have to respect some equations, notably:

1. × e = e × e
2. × e = e × e
3. + 0 = 0
4. + 1 × 1 = 1 × 1
5. 0 + 0 × 1 = 0 × 1
6. × 0 × 1 = 0 × 1
7. × + 0 = 0 + 0 = 0
8. × + 0 = 0 + 0 = 0
9. + 0 = 0
10. + = + = +
11. × = × = ×
12. + = + = +
13. × = × = ×
14. × = × = ×
15. × = × = ×

An example: \( 1, 1, \omega \) Coeffects

In this simple example, coeffect new Zero() is meant to be assigned to unused variables, new Omega() to variables used linearly (exactly once), new Omega() to unrestricted variables. Class Linearity is used to define methods zero and one only.

Properties of user-defined coeffects

To guarantee preservation of coeffects during execution, operators determined by user-defined methods have to respect some equations, notably:

Given coeffects \( c_1, c_2, c_3 \):

1. \( c_1 + c_2 + c_3 \)
2. \( c_1 + (c_2 + c_3) = (c_1 + c_2) + c_3 \)
3. \( c_1 + 0 = c_1 \)
4. \( c_1 + 1 = 1 \times c_1 \)
5. \( 0 + 0 = 0 \times 0 = 0 \)
6. \( c_1 	imes (c_2 + c_3) = (c_1 	imes c_2) + (c_1 	imes c_3) \)
7. \( 1 	imes (c_2 + c_3) + c_1 = c_1 \times c_2 + c_1 \times c_3 \)
8. \( (c_1 + c_2) \times c_3 = c_1 \times c_3 + c_2 \times c_3 \)

Soundness result

Execution preserves types and coeffects assuming that user-defined coeffects guarantee conditions 1–15

Future goals

• Adding graded modal types
• Allowing a “global” annotation in a method's signature
• Allowing variables in coeffect annotations