# Typing Coroutines

Konrad Anton Peter Thiemann

A coroutine is a programming construct between function and thread. It can be invoked like a function, but before it returns a value (if ever) it may suspend itself arbitrarily often to return intermediate results and then be resumed with new inputs. Unlike with preemptive threading, a coroutine does not run concurrently with the rest of the program, but rather takes control until it voluntarily suspends to either return control to its caller or to pass control to another coroutine. Coroutines are close to cooperative threading, but they add value because they are capable of passing values into and out of the coroutine and they permit explicit switching of control.

The main uses of coroutines are the implementation of compositions of state machines and the implementation of generators. The latter use has lead to renewed interest in coroutines and to their inclusion in mainstream languages like C#, albeit in restricted form as generators.

Despite the renewed interest in the programming construct per se, the typing aspects of coroutines have not received much attention. Indeed, the supporting languages are either untyped (e.g., Lua, Scheme, Python), the typing for coroutines is trivalized, or coroutines are restricted so that a very simple typing is sufficient. For instance, in Modula-2, coroutines are created from parameterless procedures so that all communication between coroutines must take place through global variables. Also, for describing generators, a simple function type seems sufficient.

We propose a static type system for coroutines where coroutines are first-class values, coroutine operations can be performed within nested function calls, and both asymmetric (generator-style) and symmetric (task-switching-style) coroutine operators are available. Moreover, we permit passing arguments to a coroutine at each start and resume operations and we permit returning results on each suspend and on termination of the coroutine (and we distinguish these two events). Our type system is based on the simply-typed lambda calculus extended with effects that describe the way the coroutine operations are used. We present a small-step operational semantics for the language and prove type soundness.