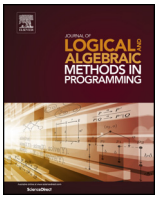
Journal of Logical and Algebraic Methods in Programming 100 (2018) 16–35

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp



A type checking algorithm for concurrent object protocols

Luca Padovani

Università di Torino, Dipartimento di Informatica, Corso Svizzera 185, 10149 Torino (TO), Italy

ARTICLE INFO

Article history:
Received 15 August 2017
Received in revised form 25 March 2018
Accepted 5 June 2018
Available online xxxx

Keywords:
Objective Join Calculus
Concurrent objects
Object protocols
Behavioral type checking
Type inference
Commutative Kleene Algebra

ABSTRACT

Concurrent objects can be accessed and possibly modified concurrently by several running processes. It is notoriously difficult to make sure that such objects are consistent with – and are used according to – their intended protocol. In this paper we detail a type checking algorithm for concurrent objects protocols that provides automated support for this verification task. We model concurrent objects in the Objective Join Calculus and specify protocols using terms of a Commutative Kleene Algebra. The presented results are an essential first step towards the application of this static analysis technique to real-world programs.

1. Introduction

The flourishing research on *behavioral types* [25] aims at developing static analysis techniques for ensuring that programmable resources are used according to a given *protocol* expressed as a type. Most of the current research in this field focuses on *session types* [22,23], a family of behavioral types specifically suited to the description of communication protocols over private communication channels called *sessions* which connect two or more processes. A cornerstone trait of virtually all session type systems, regardless of the number of interacting processes, is that the resources granting access to the session – called *session endpoints* – are meant to be used by a single process at any given time. This assumption is key for the soundness of the approach and is enforced by suitable forms of linearity embedded in the type system. In contrast to this setting, the present work considers *concurrent objects*, namely objects that are accessed and possibly modified concurrently by several processes. Instances of such objects are common in everyday concurrent programming and include locks, barriers, queues and future variables. In fact, every object may turn into a concurrent object depending on the context in which it is used. *Actors* [20,1] as implemented for example in Erlang [4] or Scala Akka [19] are another popular instance of concurrent objects. Sessions themselves can be considered concurrent objects: interacting processes concurrently access the session through one of its endpoints, each endpoint providing a distinct interface to the session [8,29,23].

In previous work, Crafa and Padovani [12] have proposed a behavioral type system ensuring that concurrent objects are consistent with, and are used according to, their protocol specified as a behavioral type. Specifically, the type system checks that, whenever a message is sent to a (concurrent) object, that message is allowed to be sent to (and possibly be processed by) the object taking its state into account. Typical examples of protocol violations are: releasing a lock that has not been acquired beforehand, removing an element from an empty queue, resolving a future variable twice. Not surprisingly, the type system of Crafa and Padovani is quite different from those based on session types. While session types are structured using choice and *sequential composition* [22,25], behavioral types for concurrent objects are structured using

E-mail address: luca.padovani@unito.it.

https://doi.org/10.1016/j.jlamp.2018.06.001
2352-2208/

Table 1
Syntax of the behaviorally typed Objective Join Calculus.

Process	P, Q	::=	`null`	(idle process)
		|	$u!\mathtt{m}(\overline{u})$	(message output)
		|	$P \,\&\, Q$	(process composition)
		|	`object` $a : t\ [C]\ P$	(object definition)
Pattern	J, K	::=	$\mathtt{m}(\overline{x})$	(message pattern)
		|	$J \,\&\, K$	(pattern composition)
Class	C, D	::=	$J \rhd P$	(reaction rule)
		|	$C \mid D$	(class composition)

choice, *concurrent composition* and *unlimited sharing*. Sequential composition is attained by means of explicit continuation passing, taking advantage of higher-order types. Crafa and Padovani [12] prove their type system sound but leave the definition of a corresponding type checking algorithm for future work. The contribution of this paper is an achievement of this pending goal. There are various factors that make the type system of Crafa and Padovani challenging to turn into a type checking algorithm. First, there is a loose correspondence between type connectives and code constructs, meaning that the typing rules rely on substantial amounts of guessing and non-local information for finding the "right" way of typing code. Second, the notion of subtyping (and consequently that of type equivalence) crucially embeds the laws of Commutative Kleene Algebra [10,24]. It follows that semantically related types can be syntactically very different. Third, the extensive use of explicit continuations leads to a proliferation of higher-order types, to the point that a certain amount of type inference is highly desirable if not outright necessary. The type checking algorithm we present in this paper addresses these challenges and is shown to be correct and complete with respect to Crafa and Padovani's type system.

Structure of the paper We begin overviewing the model of concurrent objects adopted by Crafa and Padovani [12] and the corresponding type system (Section 2). Then, we present an alternative but equivalent formulation of the typing rules that is more amenable to be realized as a type checking algorithm (Section 3). This reduces the type checking problem to resolving a system of *type constraints*, for which we give a resolution procedure (Section 4). Throughout the paper we use a simple but comprehensive example to illustrate all the phases of the type checking algorithm. A slightly more complex example is presented in Section 5. We conclude with an overview of related work (Section 6) and a few hints at future developments (Section 7). Relatively long proofs have been postponed in Appendix A so that they do not interrupt the flow of the main text. Cobalt**Blue** [30] is a Haskell implementation of the presented type checking algorithm. Its distribution includes the source code, a tutorial introduction to concurrent object protocols and several additional examples.

2. The behaviorally typed Objective Join Calculus

Our model of concurrent objects is the Objective Join Calculus [16], a mild extension of the Join Calculus [15] whose underlying computational model is the Chemical Abstract Machine [5]. As discussed by Crafa and Padovani [12] and briefly recalled in Section 6, the Objective Join Calculus is a natural core model of typestate-oriented programming (TSOP) in a concurrent setting. The syntax of the calculus is shown in Table 1 and makes use of infinite sets of *object names*, ranged over by a, b, c, of *variables*, ranged over by x, y, z, and of *message tags*, ranged over by $\mathtt{m}$. *Names*, ranged over by u, are either object names or variables. Hereafter we will make extensive use of the notation $\overline{e}$ for denoting finite, possibly empty sequences of various entities. For instance, we write $\overline{u}$ for denoting the sequence $u_1, \ldots, u_n$ of names where n may be 0 if the sequence is empty. We write $|\overline{e}|$ for the length of $\overline{e}$.

Processes in the Objective Join Calculus may have one of four possible forms. The term `null` represents the idle process that does nothing. The term $u!\mathtt{m}(\overline{u})$ represents the process that (asynchronously) sends the *message* $\mathtt{m}(\overline{u})$ to the object u. The message consists of a tag $\mathtt{m}$ and a possibly empty sequence of arguments $\overline{u}$. The *arity* of the message is the length of $\overline{u}$. The term $P \,\&\, Q$ represents the parallel composition of P and Q. Finally, the process `object` $a : t\ [C]\ P$ creates an object a with type t and scope P. Syntax and semantics of types will be given shortly. The behavior of the object is described by its *class* C, which is a non-empty, finite collection of *reaction rules* of the form $\mathtt{m}_1(\overline{x_1}) \,\&\, \cdots \,\&\, \mathtt{m}_k(\overline{x_k}) \rhd Q$. Whenever the messages $\mathtt{m}_1(\overline{c_1}), \ldots, \mathtt{m}_k(\overline{c_k})$ are targeted to a, they are atomically consumed and the process $Q\{\overline{c_1}/\overline{x_1}\} \cdots \{\overline{c_k}/\overline{x_k}\}$ is spawned, where $P\{c/x\}$ is the usual notation for the capture-avoiding substitution of the free occurrences of x with c in P and $P\{\overline{c}/\overline{x}\}$ its obvious extension to same-length sequences of names and variables.

We do not provide further details on the semantics of processes, which is irrelevant in this paper, and we omit other syntactic forms – most notably molecules and soups – that are needed only to describe their operational semantics. The interested reader may refer to Fournet et al. [16] and Crafa and Padovani [12]. The notions of free and bound names follow from the syntax as expected, noting that the binders are message patterns $\mathtt{m}(\overline{x})$, whose scope is the process on the right-hand side of the reaction in which they occur, and object definitions. The name of an object is visible in its own reactions. We identify processes modulo their bound names.