Science of Computer Programming 100 (2015) 61–83

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Model checking recursive programs interacting via the heap

I.M. Asăvoae [b,*], F. de Boer [c,a], M.M. Bonsangue [a,c], D. Lucanu [b], J. Rot [a,c,**,1]

[a] Leiden Institute of Advanced Computer Science (LIACS), The Netherlands
[b] Faculty of Computer Science, Alexandru Ioan Cuza University, Romania
[c] Centrum Wiskunde en Informatica (CWI), The Netherlands

ARTICLE INFO

Article history:
Received 14 January 2013
Received in revised form 7 May 2014
Accepted 22 September 2014
Available online 2 October 2014

Keywords:
Heap manipulation
Pushdown system
Object-oriented program semantics
Model checking

ABSTRACT

Almost all modern imperative programming languages include operations for dynamically manipulating the heap, for example by allocating and deallocating objects, and by updating reference fields. In the presence of recursive procedures and local variables, the interactions of a program with the heap can become rather complex, as an unbounded number of objects can be allocated either on the call stack using local variables, or, anonymously, on the heap using reference fields. As such, a static analysis for recursive programs with dynamic manipulation of the heap is, in general, undecidable.

In this paper we study the verification of recursive programs with unbounded allocation of objects, in a simple imperative language with heap manipulation. We present a semantics for this language which is improved w.r.t. heap allocation, using an abstraction that is precise (i.e., bisimilar with the standard/concrete semantics). For any program with a bounded visible heap, meaning that the number of objects reachable from variables at any point of execution is bounded, this abstraction is a finitary representation of its behaviour, even though an unbounded number of objects can appear in the state. As a consequence, for such programs model checking is decidable. Finally, we introduce a specification language for heap-properties, and we discuss model checking of heap invariant properties against heap-manipulating programs.

1. Introduction

One of the major problems in model checking recursive programs which manipulate dynamic linked structures is that the state space is infinite, since programs may allocate an unbounded number of objects during execution by updating reference fields (pointers). Indeed model checking and reachability for such programs are undecidable, in general. Consequently, to allow a restricted form of model checking we need to impose either some syntactic restrictions on the program [13] or some suitable bounds on its model. A natural bound for model checking programs without necessarily restricting their capability of allocating an unbounded number of objects is to impose constraints on the size of the *visible* heap [7]. The visible heap consists of those objects which are reachable from the variables in the scope of the currently executed procedure. Such a bound still allows for storage of an unbounded number of objects onto the call-stack, using local variables.

* Corresponding author.
** Corresponding author at: Leiden Institute of Advanced Computer Science (LIACS), The Netherlands.
E-mail addresses: mariuca.asavoae@info.uaic.ro (I.M. Asăvoae), frb@cwi.nl (F. de Boer), marcello@liacs.nl (M.M. Bonsangue), dlucanu@info.uaic.ro (D. Lucanu), jrot@liacs.nl (J. Rot).
[1] The research of this author has been funded by the Netherlands Organisation for Scientific Research (NWO), CoRE project, dossier number: 612.063.920.

http://dx.doi.org/10.1016/j.scico.2014.09.009
0167-6423/

In this paper we introduce a method for model checking sequential imperative programs with pointers and recursive procedure calls. In order to allow implementation of model checking of unbounded object allocation in the context of a bounded visible heap, we introduce a new mechanism for the generation of fresh object identities which allows for the *reuse* of object identities and which includes a *renaming* scheme to resolve possible resulting name clashes. We introduce a formal operational semantics based on this mechanism for an abstract programming language, called Shylock [32].

Our renaming mechanism allows a different kind of reuse of object identities than usual garbage collection techniques. A garbage collector typically reuses object identities from the heap and considers objects on the call stack as still in use. In contrast, our technique is more tailored towards model checking, and as such, we need to reuse as much object identities as possible to guarantee a representation of program behaviour in terms of a finite pushdown system with a finite stack alphabet. In fact, our mechanism allows to reuse objects allocated in the call stack, that may become active when procedures return.

The semantics of Shylock is expressed in terms of a pushdown system specification, a conditional rewriting system specifying the rules of an ordinary pushdown system. We adapt an existing model checking algorithm for ordinary pushdown systems to work with pushdown system specifications, so to generate the rules of the pushdown system on-the-fly, when needed by the model checker. The method is novel and allows for a more compact representation of infinite systems, i.e., their semantics representation, paving the way to applications of model checking for recursive programs defined by means of structural operational semantics.

Subsequently, we introduce a logic for reasoning about properties of the heap, where we use atomic propositions defined as regular expressions in what is basically a Kleene algebra with tests [23]. Namely, the global and local variables of a program are used as *nominals*, whereas the pointers (reference fields) constitute the set of basic actions. We give a decision procedure for checking if a heap satisfies a regular expression by a coinductive mean, using derivatives. The procedure is then applied in the model checking algorithm to check invariant properties.

Structure of the paper In the next paragraph we briefly discuss related work. We introduce Shylock and its formal semantics in Section 2. In Section 3 the abstraction of this semantics is introduced, together with a proof of its correctness. Then in Section 4 we define a logic for expressing temporal properties of heaps, and finally in Section 5 we conclude.

Related work This paper is an extended and revised version of [32]. In the present paper we have added an elaborate treatment of model checking invariants for pushdown systems, and apply this general theory to the case of Shylock. Moreover we have added a number of examples throughout the paper to illustrate the theory. In [32] a novel technique for resolving name clashes in the context of reuse of object identities is introduced. It is based on the concept of *cut points* as originally introduced in [30] to support static analysis via abstract interpretation techniques. Cut points are objects in the heap that are referred to from both local and global variables, and as such are subject to modifications during a procedure call. Recording cut points in extra logical variables allows for a precise abstract execution of the program, which in case of a bound on the visible heap can be represented by a finitary structure, namely that of a *finite pushdown system*. Again, by *precise* abstract execution we understand here an execution bisimilar with the concrete execution.

In [7] a language is studied with the same features as our Shylock programs extended with a bounded form of concurrency. We have decided not to incorporate concurrency in our Shylock language because this is an orthogonal dimension to the bi-dimensional growing of the state space, namely: the vertical growing of the number of objects due to recursion and the horizontal growing due to anonymous field update. In fact, bounded concurrency could easily be handled with a technique similar to the one used in [7]. The novelty of our work is not in the decidability result, that is indeed similar to the one obtained in [7], but in the technique we used to obtain it. While [7] uses finite graphs and graph isomorphisms to represent heaps and avoid name clashes, respectively, our approach is purely symbolic, and, therefore, directly usable for model checking temporal properties of heaps. We discuss the relationship with [7] in more detail in the concluding Section 5.

Currently there are several model checkers for object oriented languages. Java Path Finder [21] is basically a Java Virtual Machine that executes a Java program not just once but in all possible ways, using backtracking and restoring the state during the state-space exploration. Even if Java Path Finder is capable of checking every Java program, the number of states stored during the exploration is a limit on what can be effectively checked. As with JCAT [15], Java source code can be translated into Promela, the input language of SPIN. Since Promela does not support dynamic data structures, fixed-size heaps and stacks have to be allocated. By comparison, we can handle unbounded stacks and we only need the guarantee that the size of the heaps is bounded – guarantee that we try to enforce in our abstract semantics.

Bandera [11] is an integrated collection of tools for model checking concurrent Java software using state-of-the-art abstractions, partial order reductions and slicing techniques to reduce the state space. It compiles Java source code into a reduced program model expressed in the input language of other existing verification tools. For example, it can be combined with the SAL (Symbolic Analysis Laboratory) model checker [28] that uses unbounded arrays whose sizes vary dynamically to store objects. In order to explore all reachable states model checking is restricted to Java programs with a bounded (but not fixed a priori) number of *created* objects. Obviously, this project is quite developed however it is slightly aside the current focus of our work. Namely, Bandera is rather a front-end for model checking tools while Shylock is more like a back-end for model checking. We plan to further develop Shylock with a front-end for real programming languages.