Formal verification of Cloud Sisal programs

Victor N. Kasyanov, Elena V. Kasyanova, Dmitry A. Kondratyev
A.P. Ershov Institute of Informatics Systems SB RAS, Novosibirsk, 630090, Russia
kvn@iis.nsk.su

Abstract. A cloud parallel programming system CPPS which is under development at the A.P. Ershov Institute of Informatics Systems is aimed to support the development, verification and debugging of Cloud Sisal programs and their correct conversion into the efficient code of parallel computing systems for its execution in clouds. In this paper, the methods and subsystems of CPPS intended for formal verification of Cloud Sisal programs are described.

1. Introduction

The cloud parallel programming system CPPS is aimed to be an integrated cloud visual programming environment in the Cloud Sisal language which contains both an interpreter that supports the user interaction during creation and debugging of a functional program and an optimizing cross-compiler that builds a parallel program according to its functional specification [6 – 8].

The Cloud Sisal language carries on the traditions of previous versions of the Sisal language [4, 5] while remaining a functional data-flow language focused on writing large scientific programs and expanding their capabilities by supporting cloud computing. Its functional semantics guarantees deterministic results for parallel and sequential implementations – something that cannot be guaranteed for usual imperative languages like C or Fortran. Moreover, the implicit parallelism of the Cloud Sisal language removes the need to rewrite the source code when transferring it from one computer to another. It is guaranteed that any Cloud Sisal program correctly executed on a personal computer will be correctly executed on any high-speed parallel or distributed computer.

Using the CPPS system any user will be able to develop, verify and debug a Cloud Sisal program in clouds on his/her low-cost device in a visual style not taking into account the target supercomputers. Then it is possible to tune the debugged program to a supercomputer available via network in order to achieve high performance execution of the developed parallel program, as well as to transfer this program to the supercomputer to run it and receive its results.

To make it easier for the users to justify the correctness and reliability of the designed parallel programs, two deductive verification subsystems CSV1 and CSV2 are created for formal verification of the Cloud Sisal programs. The main difference between these subsystems is the following: the CSV2 subsystem uses the classical approach based on the axiomatic semantics of the programming language (Cloud Sisal in our case), and the CSV1 subsystem uses an intermediate verification language (C in our case). It should be noted that the second approach is also quite common [12]. For example, the following systems use the Boogie language [13] as an intermediate verification language: the HAVOC system for verification of C programs [1], the Spec# system for verification of C# programs [2], and the AutoProof system for verification of Eiffel programs [16].
It is assumed that a Cloud Sisal program and its specification in the form of a precondition and a postcondition are used as an input for the verification subsystems. If the result of the subsystem running is “yes”, then the program is partially correct, i.e. when the precondition is fulfilled and the program terminates, the postcondition is satisfied. And the result of the verification system may be “unknown”, also.

The rest of the paper is organized as follows: Sections 2 and 3 describe the CSV1 and CSV2 subsystems, and Section 4 concludes the paper and highlights the future research directions.

2. The CSV1 subsystem
The CSV1 subsystem reduces the problem of verifying the correctness of the annotated Cloud Sisal program to the problem of verifying the truth of the correctness conditions in the framework of the intermediate C-light language, which is a wide and expressive subset of the C language [10, 11]. The CSV1 subsystem supports verification of Cloud Sisal programs which consists of the following three steps. At the first step, the annotated Cloud Sisal program is translated into the annotated C-light program. At the second step, the C-lightVer system is used to generate the correctness conditions for the obtained C-light program. The third step is to prove the obtained correctness conditions.

Note that, at the first step, loop expressions of the Cloud Sisal language are translated into for loops of the C language. At the second step, to avoid generating loop invariants in the C-lightVer system, the symbolic method of verification of finite iterations is used. The loop body of such iteration is executed once for each element of the structure of finite dimension. In the symbolic method of verification of finite iterations, a special inference rule is used for such iterations. This rule is based on a special replacement operation (the so-called rep function) that expresses the action of the loop in a symbolic form. The function rep returns the values of the loop variables after the iterations, the number of which is specified by the first argument. The result of applying the rep function to the nth iteration is the result of applying the operations of the loop body to the value of the rep function at the (n-1)th iteration. Thus, the rep function is defined recursively. The C-lightVer system uses an algorithm of automatic generation of the definition of such a function.

The for loops generated by the translation of loop expressions of the Cloud Sisal language are finite iterations over data structures. Therefore, as a result of applying the symbolic method of verification of finite iterations, correctness conditions are generated containing the applications of the recursive function rep, the first argument of which is the number of iterations. The proof of these correctness conditions is by induction on the number of iterations.

But the popular satisfiability modulo theories (SMT) solvers insufficiently support proof by induction [14]. Our attempts to prove separately the basis of induction and the induction step also did not lead to success. As a result, it was decided to use the theorem prover ACL2 [9, 14] which supports several deductive mechanisms for proof structuring and permits the ACL2 user to extend a theory and to invoke an external tool during a proof attempt. Despite of the built-in support for proof by induction, ACL2 could not cope with the proof of correctness conditions which contained the rep function in the automatic mode. Therefore, an automated strategy was developed to prove such correctness conditions. This strategy is based on strengthening the correctness condition and it is applied at the third step of verification in the CSV1 system. It is a problem for the ACL2 system to compare the results of the function invocation with a constant or a variable. In this case, induction on a variable which is the argument of such an invocation may lead to a situation when the inductive hypothesis is not used by ACL2.

The basic idea of this strategy is to make the system ACL2 try to prove not the original condition of correctness but a strengthened one. Thus, the strategy consists in application (with possible reiterations) of transformations strengthening a formula. In addition, the following property should hold: the truth of the strengthened formula implies the truth of the original formula.

The main transformation for strengthening a formula is the replacement of constants and variables in conclusions of implications by function invocations. This replacement is performed using the properties of transitivity or non-transitivity of different types of comparison relations. If the variable...
on which induction is conducted appears as an argument of such functions, then it simplifies application of the induction hypothesis in the case of the system ACL2.

Since this strategy is heuristic, it is impossible to guarantee its completeness. The correctness of the CSV1 system depends on the termination and correctness of this strategy. It was proved that the strategy of strengthening the correctness condition always terminates producing either “formula is valid” for some partially correct program or “unknown”.

To prove correctness of the strategy of strengthening the correctness condition, we consider a set of transformations that are performed in a certain order and strengthen the correctness condition. The proof of correctness is reduced to checking the truth of the property that, for any possible sequence of application of these transformations, the resulting formula is a strengthening of the original one. To do this, it is enough to prove that each transformation either strengthens the formula or leaves it equivalent to its initial form.

The proof is based on induction on the transformation application number. It is proved that the transformation either strengthens the input formula or is an equivalent transformation. The proof is divided into cases, each of which corresponds to a step of the strategy. The main step of the strategy is a transformation that replaces variables and constants by function invocations. It has been proven that if such a replacement occurs, then the original function is strengthened. The proof is based on the use of the transitivity / non-transitivity property of various types of comparison relations.

The following two examples illustrate the applicability of the CSV1 subsystem. The first example is successful verification of the annotated Cloud Sisal program `Search_count` checking that the number of occurrences of a given key in a given array is not less than a given number. The second example is successful verification of the annotated Cloud Sisal program `Is_ordered` checking the orderliness of a given array by counting the number of consecutive ordered pairs of its elements.

3. The CSV2 subsystem
The CSV2 subsystem is based on the axiomatic semantics of a representative subset of the Cloud Sisal language and it will have property of completeness when the axiomatic semantics of the entire Cloud Sisal language will be used by the CSV2 subsystem.

Typically, for constructs of a programming language, one can propose different sets of axioms and inference rules that form different axiomatic semantics. Therefore, it is convenient to define semantics so that special output methods can be applied. To trace the weakest precondition, reverse tracking is used: we move from the end of the program to its beginning, removing the rightmost statement (at the top level), applying the corresponding derivation rule of the axiomatic semantics. Thus, axiomatic semantics oriented towards backtracking contain inference rules that eliminate the last program statement from its conclusion. The advantage of the weakest precondition method is the absence of quantifiers when eliminating the assignment statement. So, it was decided to develop the axiomatic semantics for a subset of the Cloud Sisal language with orientation to the weakest precondition method. A special term `result` was introduced into the specification language for using it in postconditions to model values of the Cloud Sisal expressions.

We strive to automate the operation of all components of our CSV2 subsystem and use the ACL2 system [9, 14] to prove the correctness conditions. So, in our CSV2 subsystem, before proving the correctness conditions source program specification should be converted into the ACL2 language which is an applicative dialect of the Common Lisp language. Therefore, as part of the CSV2 subsystem, a translator (`sisal_to_acl2`) of the Cloud Sisal expressions to the ACL2 language was developed. This translator is defined recursively: the translation of constants and variables is specified as the recursion basis, and the translation of a compound expression is defined through the translation of its subexpressions.

Special constructs for Cloud Sisal expressions have been defined in the ACL2 system. The theories containing theorems about these constructions have been developed for these constructions. These theories have been developed in order to simplify the proof of correctness conditions containing such constructions.
When translating expressions and generating correctness conditions in the ACL2 language, it is necessary to use information about type of each expression of the Cloud Sisal program. To solve this problem, we introduced the \textit{sisal_expr_type} function, which receives information from the Cloud Sisal compiler. Also, the mapping of the types of the Cloud Sisal subset introduced by us to the types of the ACL2 language is supported. Therefore, special constructs were introduced in the ACL2 system to support the types of the Cloud Sisal subset.

To support implicit parallelism, the so-called \textit{loop} expression is used in the Cloud Sisal language [6]. In heading of the loop expression for variables so-called \textit{triplets} are used; they are \textit{ranges}, the values from which these variables take during the execution of the loop. A repetitive execution of the \textit{loop} expression determines a sequence of all values of all variables of the loop, corresponding to all its iterations. The \textit{loop} expression has the co-called \textit{reduction} which is used to reduce all values of such sequence to one or more scalar values (or to gather them to one or more aggregate data structures) as the value of the \textit{loop} expression. For example, the \textit{value} reduction of a variable allows to get us the last element of such a sequence of its values, i.e. the value of this variable after execution of the loop.

To avoid the requirement for the user of the CSV2 system to specify the invariants of loops, we derive their weakest precondition using the symbolic method of verification of finite iterations. A special operation of replacement (the \textit{rep} function) expresses the action of loop in a symbolic form, is used. This method offers the special inference of terms of correctness rule for loops, based on the operation of replacement. The advantage of these inference rule is non-use of invariants.

The \textit{loop} expressions of the Cloud Sisal language are based on triplets, so we introduced them in the considered subset of the Cloud Sisal language. The \textit{sisal_to_acl2} translator transforms every triplet in the list of values of its range. To define such a list in ACL2, a special \textit{triplet} function and theorems for it have been introduced.

Different ranges for different variables can be used in the \textit{loop} expression header. To simulate the execution of such a loop, we use the Cartesian product of all ranges as one of the arguments to the \textit{rep} function. To specify the Cartesian product, the \textit{cartesian-product} function defined in ACL2 is used. Also, in the ACL2 system, a theory is given for this function containing theorems about it.

At present, we model the \textit{loop} expression using the \textit{rep} function and consider a subset of Cloud Sisal without the \textit{while} tests in the \textit{loop} expressions. It is difficult to define the replacement operation for a loop with a \textit{while} test, since it can interrupt the loop execution when a certain condition is met.

The axiomatic semantics of the described subset of the Cloud Sisal language was used in our experiment. To illustrate applicability of CSV2, verification of the Cloud Sisal program \textit{Sum_array} computing the sum of all elements of a given two-dimensional array was successfully performed.

4. Conclusion
This paper describes the verification subsystems of the CPPS system which is intended to provide means to write, debug and verify parallel programs regardless of target architectures on low-cost devices and then execute them in clouds on high-performance parallel computers without extensive rewriting and debugging. In the CPPS system, an input Cloud Sisal program plays the role of an executed specification of parallel programs. So, the CSV1 and CSV2 subsystems here described can be considered as formal verifiers for parallel programs. But in their current state, they are applicable only in experiments, and their capabilities should be enhanced. Future research is conducted in a several parallel and complementary directions including the following.

With the development of the CSV1 verification subsystem, it is planned to add new strategies to it, oriented towards the automation of the proof of the correctness conditions in the ACL2 subsystem. Such strategies will be based on other ways of strengthening the correctness formulas. We plan to prove the correctness of such strategies before their implementation in our verification subsystem.

Applicability of the verification subsystem CSV2 will be expanded. In the next version of the axiomatic semantics of the Cloud Sisal language within the CSV2 framework we are going to realize support of the following Cloud Sisal constructions: the \textit{loop} expressions guided by \textit{while} tests, the \textit{old-}
names, and the user reductions. We plan to complete this process when the axiomatic semantics of the entire Cloud Sisal language will be developed.

In addition, we are also working on the development of verification methods and subsystems in the direction of their generalization aimed at formal verification of correctness of the optimizing transformations performed by the Cloud Sisal compiler. There are two types of the optimizing transformations in the parallelizing compilers [6]: the so-called restructuring optimizations that transform programs into the same programming language (dead code elimination, loop invariant removal, and so on), and the so-called constructing optimizations that transform programs from one programming language to another. We consider the CSV1 subsystem as a basis for formal verification of constructing optimizations that transform the Cloud Sisal programs to the equivalent C programs. The CSV2 is considered by us as a basis for formal verification of the restructuring optimizations.

Acknowledgements
This work was supported by a grant of the Russian Science Foundation (project 18-11-00118).

References
[1] Ball T, Hackett B, Lahiri S K, et al 2010 Towards scalable modular checking of user-defined properties Lecture Notes in Computer Science vol 6217 Springer pp 1–24
[2] Barnett M, Leino K R M, Schulte W 2005 The Spec# programming system: An overview Lecture Notes in Computer Science vol. 3362 Springer pp 49–69
[3] Evstigneev V A, Kasyanov V N 1996 Optimizing transformations in paralleling compilers Programming and Computer Software 22 (6) pp 279–290
[4] Gaudiot J-L, DeBon T, Feo J, et al 2001 The Sisal project: real world functional programming. Lecture Notes in Computer Science vol 1808 Springer pp 45–72
[5] Kasyanov V 2013 Sisal 3.2: functional language for scientific parallel programming Enterprise Information Systems 7 (2) 227–236
[6] Kasyanov V, Kasyanova E 2018 A system of functional programming for supporting of cloud supercomputing WSEAS Transactions on Information Science and Applications 15 81–90.
[7] Kasyanov V N, Kasyanova E V 2019 Methods and system for cloud parallel programming Proceedings of the 21st International Conference on Enterprise Information Systems (ICEIS 2019) vol 1 SCITEPRESS pp 623–629
[8] Kasyanov V, Kasyanova E, Zolotuhin T 2018 Visualization of graph representations of data-flow programs WSEAS Transactions on Information Science and Applications 15 141–146.
[9] Kaufmann M, Moore J S 1997 An Industrial Strength Theorem Prover for a Logic Based on Common Lisp IEEE Transactions on Software Engineering 23 (4) 203–213
[10] Kondratyev D 2018 Implementing the symbolic method of verification in the C-light project Lecture Notes in Computer Science vol 10742 Springer pp 227–240
[11] Kondratyev D, Promsky A 2019 Proof strategy for automated Sisal program verification. In Lecture Notes in Computer Science vol 11771 Springer pp 113–120
[12] Leino K R M 2012 Program proving using intermediate verification languages (IVLs) like Boogie and Why3 Proceedings of the 2012 ACM conference on High integrity language technology ACM pp 25–26
[13] Leino K. R. M., Rümmer P A 2010 Polymorphic intermediate verification language: Design and Logical Encoding Lecture Notes in Computer Science vol 6015 Springer pp 312–327
[14] Moore J S 2019 Milestones from the pure Lisp theorem prover to ACL2 Formal Aspects of Computing 31 (6) pp 699–732
[15] Rustan K, Leino M 2012 Automating induction with an SMT solver Lecture Notes in Computer Science vol 7148 Springer pp 315–331
[16] Tschannen J, Furia C A, Nordio M, Meyer B 2012 Automatic verification of advanced object-oriented features: The AutoProof approach Lecture Notes in Computer Science vol 7682 Springer pp 133–155

5