ISSN 1512-1232

UDC 004.4

Usage of Logic for Parallel Verification of Haskel Programs

Natela Archvadze[1], Merab Pkhovelishvili[2], Lia Shetsiruli[3], Otar Ioseliani[4]

[1] Department of Computer Sciences Faculty of Exact and Natural Sciences I. Javakhishvili Tbilisi State University 2, University st., 0143, Tbilisi, GEORGIA, Natela.Archvadze@tsu.ge
[2] Department of Programming N.Muskhelishvili Computing Mathematic Institute of Georgian Technical University, 7, Akuri st., 0193, Tbilisi, GEORGIA, merab5@list.ru
[3] Department of Mathematics and Computer Science Shota Rustaveli State University, 35, Ninoshvili st., 6010, Batumi, Georgia, likalika77u@yahoo.com
[4] Georgian-American University, 8 Merab Aleksidze str., 0160 Tbilisi, Georgia
otari.ioseliani@gmail.com

Abstract

Haskell was created as a functional programming language and extended later with possibilities to create additional functions for high level parallel programming. Kripke schemes are traditionally used in verification systems which use model checking with temporal (time) logic to describe the state of the program. Temporal logic is a good solution for processing Kripke schemes for linear programs verification but for verification of functional parallel programs it is inconvenient. P-logic used for parallel programing the Plover system is more attractive however Kripke schemes are more convenient for describing states in parallel programs.

In this research a tree of programs states is presented with Kripke schemes and is processed according to wave principle. Tree traversal is performed according to P - logic principals instead of temporal logics used in model checking.

Keywords: *Haskell, Verification, Functional Programming, Parallel calculations.*

1. Introduction

Model checking gives an opportunity to verify whether the given model of a parallel system with a finite number of states satisfies its formal specifications defined in a language of formal logic.

Hardware and software specifications are usually developed in temporal logic, which is a special language to describe the behavior of a system in time.

Nowadays, model checking for (temporal) logics is commonly accepted as one of the key methods in verification. A major limitation to the usefulness of model checking for verification purposes is the state space explosion problem. To solve this problem one employs symbolic methods [3] that work on process descriptions directly an usually achieves better performance there.

The system which has been reviewed in [1] is using the modal, fixpoint logic. It's being performed elaboration of symbolic model checking and it's being discussed how this algorithms should be parallelized however, for this reason is being used Glasgow Parallel Haskell (GpH) and its performance on a cluster of workstations.

The need for efficiency together with facing huge state spaces in relevant applications often leads to the use of logics with little expressive power for model checking tasks. This bears an obvious disadvantage: what if a desired correctness property is not expressible in this logic? This justifies the search for (as efficient as possible) decision procedures for logics with higher expressive power.

ISSN 1512-1232

A great step on the expressivity ladder regarding temporal logics was made with the introduction of FLC, a fixpoint logic that extends the modal μ- calculus with an operator for sequential composition. This gives FLC the power to express non-regular properties like "on every path the number of a's so far never exceeds the number of b's" or "something holds on all paths at the same time".

P- logic [4] was used particularly for verification of Haskell language, It's being used by system of "Plover" which is an automated property-verifier for Haskell programs [5]. This tool, called Plover, is being developed as part of the Programmatic project, whose objective is to explore means of providing scientifically based certification of formally specified properties of computer programs.

A Programatic certificate is a electronic document which is structured and provides tangible, auditable evidence that a source-code module has a specified property.

Certificates are associated with program modules by encrypted links that resist forgery. Many forms of evidence can be accommodated. These can include the testimony of expert reviewers, results of testing, formal proofs of properties and software model checking. Different forms of evidence are supported by a variety of certificate servers, and may evoke varying degrees of trust in the certification provided.

Plover is one of the Programatic certificate servers. It is intended to provide a degree of assurance based upon the soundness of automated reasoning in a formal logic. Ultimately, the quality of this reasoning depends upon the correctness of the Plover tool itself, thus may be less convincing than if it supplied a formal proof tree. On the other hand, the reasoning conducted by Plover is fully automated, and is thus obtained with far less user expertise and expenditure of effort than is required for proof construction with the aid of a theorem-proving assistant. Furthermore, Plover specifically implements reasoning in P-logic, which is the verification logic of Haskell 98, whereas few other available proof assistants directly support a verification logic so closely tied to a wide-spectrum programming language.

From annotation [6] a new strategy called "strength induction" was described to support automatic checking of assumptions.

Strength induction has been implemented in Plover, an automated property-verifier for Haskell programs that has been under development for the past three years as a component of the Programmatic project. In Programmatic, predicate definitions and property assertions written in P-logic, a programming logic for Haskell, can be embedded in the text of a Haskell program module. Properties refine the type system of Haskell but cannot be verified by type-checking alone because a more powerful logical verifier is required.

Plover codes the proof rules of P-logic, and additionally, embeds strategies and decision procedures for their application and discharge. It integrates a reduction system that implements a rewriting semantics for Haskell terms with a congruence-closure algorithm that supports reasoning with equality.

Besides P-logic other examples of language-specific verification logics are ACL2 [7], a verification logic for Common Lisp, and Sparkle [8], a verifier for Clean 2.0. When assertions are formulated in a language-specific verification logic it is unnecessary to translate expressions and their asserted properties into another logical formalism, which may have a different type system, and with the attendant risk that errors may be introduced in the translation.

The model checking is used not only for verification, but may be used for debugging Concurrent Haskell programs [9].

Today, almost any larger software application is no longer a sequential program but an entire concurrent system made of a varying number of processes. These processes often share resources which must be adequately protected against concurrent access. Usually this is achieved by concentrating these actions in critical sections which are protected by semaphores. If a semaphore uses a count, another process waiting for access is suspended until the semaphore is released. Combining two or more semaphores can easily lead to situations where a deadlock might occur.

ISSN 1512-1232

The complex of temporal qualified - free formulas represents temporal logic LTL. LTL formulas specifying assertions or other properties are verified at runtime. It is possible to dynamically add formulas at runtime, giving a degree of flexibility which is not available in static verification of source code. If the properties which falsifies a formula is detected at runtime, the debugger emits a warning and records the path leading to the violation. It is possible to dynamically add formulas at runtime, giving a degree of flexibility which is not available in static verification of source code.

Sometimes for verification is being used several methods like testing, interactive proof, model checking at the same time [10].

Testing and model checking are used for debugging programs and specifications before a costly interactive proof attempt. During proof development, testing and model checking quickly eliminate false conjectures and generate counterexamples which help to correct them. With an interactive theorem prover it can be ensured that the correctness of the reduction of a top level problem to subproblems that can be tested or proved. The method can be demonstrated using our random testing tool and binary decision diagrams-based (BDDs) tautology checker, which are added to the Agda/Alfa interactive proof assistant for dependent type theory. In particular it can be applied techniques to the verification of Haskell programs. The first example verifies the BDD checker itself by testing its components. The second uses the tautology checker to verify bitonic sort together with a proof that the reduction of the problem to the checked form is correct.

The modern realization of verification as VeriFast [11] is considering industrial use of software. In case on non existence of unlawful access to memory the orientation can occur, for example dividing on 0 or checking memory leakages. It’s being performed for Java and and C languages and according to this the industrial application is possible.

Conclusion: For the verification of Haskell programs should be used the scheme of Kripke and P-logic instead of temporal logics, because the temporal logics doesn't work for competitive programming verification and it should be performed the transformation of model – checking.

2. The issues of verification of verification Haskell’s parallel programs

1. P - logic instead of temporal logic

Strength induction has been implemented in Plover, an automated property-verifier for Haskell programs that has been under development for the past three years as a component of the Programatic project. In Programatic, predicate definitions and property assertions written in P-logic, a programming logic for Haskell, can be embedded in the text of a Haskell program module. Properties refine the type system of Haskell but cannot be verified by type-checking alone; a more powerful logical verifier is required.

Plover codes the proof rules of P-logic, and additionally, embeds strategies and decision procedures for their application and discharge. It integrates a reduction system that implements a rewriting semantics for Haskell terms with a congruence-closure algorithm that supports reasoning with equality.

2. Presentation of parallel programs by the Kripke schemes

The parallel programs consist of several ordered processes (modules). Each of this processes works independently but in case of necessity of solving unified issue they are interacting with other processes by using common variables or by exchanging messages in the connection channels.

For example, if we have two processes PR1 and PR2 which are performing in parallel and independently the simple operations of assignment:

PR1::

```
A:=5;
A:=B;
```

ISSN 1512-1232

```
END
    PR2::
   B:=7;
   B:=A:
    END
```

Sounds question, for which values of A and B variables the execution of program will be stopped, if the initial values of A and B variables are equal to 0. The attention should be paid that PR1 and PR2 have common variables.

To answer this question is being used the principals of Model Checking [12].

Model Checking is one of most perspectives and nowadays widely used approach, the method for automatically adjustment and checking correctness of program. The stages of Model Checking can be described like:

The stage of modeling - for the system which will be projected it's necessary to create the abstract model of it (for example: the full system of transactions), for which it will be acceptable the instrumental verification solutions for programs model.

The stage of specification - this issue consist from creating properties which should be included in the projecting system. Generally the specifications are being represented on the formal logic language. As a rule for hardware and software is being used dynamic logics, temporal logics and and their examples using the fixed points.

3. Parallelism in Haskell

The parallelism in Haskell represents the natural an reliable usage of calculating cores with following properties:

• The parallel programing id determined. This means that it's possible to repair parallel program in parallel without execution.

• The parallel program is a multilevel and declared, the have no direct connection with such a mechanisms as synchronization which is message.

As more abstract the program is as it's a simple to execute it on the parallel software. However it should be taken in account the quality of specification and dependence on data.

The model of parallel programing and the strategy of calculations

Lazy evaluations is a mechanism, which is being used for calculating expressions. The idea is that the calculations are being performed when there is a necessity. more precisely the calculation of arguments is being performed only in way and in time when it will be a strong necessity to reach a results. For example: after choosing first element of list the other part of the list is not necessary and it gives an opportunity to avoid in HEAD (1 : ones) impression the next calculation of ones endless list. Generally we have the following property: in case of using lazy calculations the expressions are being evaluated according to that context in which they are being used.

How to represent the "map" function, using the "Idea" which has following definition:

```
map: :( a->b)->[a]->[b]
map f []    =[]
map f (x:xs)=f x : map f xs
```

The lazy data structure which is been created for "map" function and in which is evident two "ideas" can be written:

```
map:: ( a->b)->[a]->[b]
map f []       =[]
map f (x:xs)=let
```

ISSN 1512-1232

```
    x'  = f x
    xs' = map  f  xs
  in
    x': xs'
```

The templates for the defined tail recursion code Haskell functions

The templates for the defined tail recursion code can be represented as [13-16]:

```
f [ ] = g1 [ ]
f ( x : xs ) = g2 ( g3 x ) ( g4 ( f ( g5 xs ) ) )
```

`g1, g2, g3, g4, g5` functions are depended on the program's conditions:

`g1` – is the function, to process empty list.

`g2` – is the function, which combines the tail and the top of the list.

`g3` – the function, which processes the top of the list.

`g4` – is the function, which processes the recursion call for not empty list's.

`g5` – is the function, which is processing the tail of not empty list for recursion call.

It's possible to represent for example function "last" which returns the last element from list with following example:

```
last  :: [a] -> a
last [x]         =  x
last (_:xs)      =  last xs
gl _ = error
g2 a b = b
g3 x = x
g4 x = x
g5 x = x
```

The template of the list is being represented using the "idea":

```
ListTemplate  [ ] = g1 [ ]
ListTemplate( x : xs ) = g2 ( g3 x ) ( g4 (ListTemplate( g5 xs ) )
)
ListTemplate :: [a]->b
ListTemplate []       =   []
ListTemplate (x:xs)= let
                          x'  = g3  x
                          x''=g5 xs
                          x'''= ListTemplate ( x'')
                          xs' =g4 (x''')
                          in
                          g2( x': xs')
```

3. Conclusion

Nowadays the most important is the issue of creation such as program code which can be processed in parallel on several cores of one processor or on several computers. Nowadays the computers with several cores are available for everyone, but creation such an applied program which will effectively use several streams is still a big issue.

Functional programming gives an opportunity noticeably simplify parallel programing. It's happening because in functional program are memory areas which are being used by several

ISSN 1512-1232

streams at the same time. Each function works with data which has been received from it's input. In spite of said before, the issue of effective dividing of calculations in different streams still exist.

In this research are given the main ideas which according to our opinion are the best for parallel programing in Haskell. Particularly we will use the schemes of Kripke an the P-logic. At the same time it's being shown the representing of the templates of Haskell functions according to form which makes it possible to execute it on several cores.

References

[1] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 1020 states and beyond. Information and Computation, 98(2):142–170, June 1992.

[2] Lange, M., Loidl, H.W. Parallel and symbolic model checking for fixpoint logic with chop. Electronic Notes in Theoretical Computer Science. Volume 128, Issue 3, 19 April 2005, Pages 125-138.

[3] M. M¨uller-Olm. A modal fixpoint logic with chop. In C. Meinel and S. Tison, editors, Proc.16th Symp. on Theoretical Aspects of Computer Science, STACS'99, volume 1563 of LNCS, pages 510–520, Trier, Germany, 1999. Springer.

[4] Richard B. Kieburtz. P-logic: Property verification for Haskell programs. ftp://ftp.cse.ogi.edu/pub/pacsoft/papers/Plogic.pdf, 2002.

[5] Richard B. Kieburtz. Programmed Strategies for Program Verification. Electronic Notes in Theoretical Computer Science 174 (2007), pages 3–38.

[6] Richard B. Kieburtz. Strength Induction in a Haskell Program Verifier. Electronic Notes in Theoretical Computer Science 193 (2007), pages 61–79.

[7] Maarten de Mol, Marko van Eekelen, and Rinus Plasmeijer. Theorem proving for functional programmers—SPARKLE: A functional theorem prover. In Proc. of 13th Internat. Workshop on Implementation of Functional Languages (IFL'01), volume 2312 of LNCS, pages 99–118. Springer Verlag, 2001.

[8] Matt Kaufmann, Panagiotis Manolios, and J. Strother Moore. Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers, June 2000.

[9] Volker Stolz. Runtime Verification of Concurrent Haskell Programs. Electronic Notes in Theoretical Computer Science 113 (2005), pages 201–216.

[10] P. Dybjer, Q. Haiyan, M.Takeyama. Verifying Haskell programs by combining testing, model checking and interactive theorem proving. Information and Software Technology Volume 46, Issue 15, 1 December 2004, Pages 1011-1025

[11] P. Philippaerts , J.T. Mühlberg, W. Penninckx, J.Smans, B. Jacobs, F. Piessens. Software verification with VeriFast: Industrial case studies. Science of Computer Programming. Volume 82, 1 March 2014, Pages 77–97.

[12] Birth of Model Checking. 25 Years of Model checking: Lecture notes in Computer Science, vol. 5000, 2008.

[13] N.Archvadze, M.Pkhovelishvili, L.Shetsiruli. The complexity of program synthesis from examples. Proceedings of the Eleventh International Conference Pattern Recognition and Informaton Processing (PRIP'2011). ISNB 978-985-448-772-7. pp. 275-279. http://lsi.bas-net.by/conferences/prip2011/ . 2011.

[14] N. Archvadze,M. Pkhovelishvili, L. Shetsiruli, M. Nizharadze. Program Recursive Forms and Programming Automatization for Functional Languages. WSEAS TRANSACTIONS on COMPUTERS. Volume 8, pp. 1256-1265, ISSN: 1109-2750

[15] http://www.wseas.us/e-library/transactions/computers/2009/29-531.pdf. 2009

[16] N. Archvadze, M. Pkhovelishvili, L.Shetsiruli. Construction of the Generalized Recursive Forms for Functional Languages and their Application Verification of. Electronic Scientific Journal: ―Computer Sciences and Telecommunications‖. No. 3(26), pp. 133-141. ISSN 1512-1232. http://gesj.internet-academy.org.ge . 2010.

ISSN 1512-1232

[17] N.Archvadze, M.Pkhovelishvili. POSSIBILITY OF FUNCTIONAL PROGRAMS VERIFICATION THROUGH APPLICATION OF MODEL CHECKING. Electronic Scientific Journal: ―Computer Sciences and Telecommunications‖. ISSN 1512-1232. 2013|No.4(40) [2013.12.31]. pp. 51-58.

Article received: 2016-10-19