Towards a Verification Framework for Haskell by Combining Graph Transformation Units and SAT Solving

Marcus Ermler

University of Bremen, Department of Computer Science
P.O.Box 33 04 40, 28334 Bremen, Germany
maermler@informatik.uni-bremen.de

Abstract

The development of correct software systems is of highest relevance in software engineering. Various methods have been applied to gain this goal like theorem provers, exhaustive tests, or algebraic specification techniques. In this paper, we propose a new approach for the automatic verification of Haskell programs by combining graph transformation units and SAT solving. Therefor, function equations, known properties, and the property to be proven are translated into graph transformation units for the base case and the inductive step. These units perform a structural induction to verify the property. In general, the automation of this process is highly nondeterministic because in each rewriting step severals rules could be applied, also those that may lead in the wrong direction. To tackle this deficiency we translate the derivation process of graph transformation into propositional formulas and yield, in this way, the whole state space up to a certain bound.

Contents

1 Introduction

Developing and implementing correct software systems is one of the important challenges in software engineering. It has a strong impact on security-critical environments like power plants, traffic control systems, or aviation. Various approaches have come into general use to guarantee the correctness of such systems. Famous examples are exhaustive tests, theorem provers, or algebraic specification methods.

In this paper, we propose a new approach for proving the correctness of software systems written in the functional programming language Haskell. We introduce a prototypical framework for verifying properties of Haskell functions via structural induction proofs based on graph transformation units [8] and their combination with SAT solving [1]. This seems to us worthwhile for three reasons:

(1) Graph rewriting techniques have proven successful in term rewriting (cf. e.g. [12, 10]) and functional programming (cf. [2, 7]).

(2) The translation to propositional formulas tackles the inherent nondeterminism of applying automatically function equations (or their graph transformation rule counterparts) because in each derivation step various equations could be applied but only some of them yield a desired and proper result. The translation of graph transformation to SAT has proven beneficial in the application to NP-complete graph problems [9, 6].

(3) Verification with model checkers and SAT solvers is a hot topic in general, but not much studied in graph transformation. Besides our SATaGraT tool, the GROOVE tool [11] is the only tool in the graph transformation area that directly combines graph transformation with model checking and verification.

A graph transformation unit consists of an initial and a terminal graph class expression describing the permitted in- and outputs, respectively, a set of graph transformation rules and a control condition for guiding the rule application. Haskell function equations that are converted into graph transformation rules and a property in equation form are translated into graph transformation units for the base case and the inductive step. In both cases, the left-hand side of the property is the input of the graph transformation unit and the right-hand side is their output. If both graph transformation units find a successful derivation, i.e. a derivation from the input to the output graph, then the property has been proven.

Furthermore, we devise a new translation of the derivation process of graph transformation units into propositional formulas yielding directly formulas in conjunctive normal form, the standard input format for today's SAT solver. The translation of graph transformation to SAT, the application of the SAT solver MiniSat [5], and the evaluation of the output is done fully automatically in our tool SATaGraT (***SAT*** *solving* ***a****ssists* ***Gra****ph* ***T****ransformation* Engine) [6]. We extend SATaGraT by our new translation to SAT and our proposed verification framework. Finally, we test the framework for a certain, well-known list property.

The paper is organized as follows. In Section 2 we recall the basic concepts of graph transformation and detail how derivations of graph transformation units are translated into propositional formulas. The main ideas behind our framework is described in Section 3. Section 4 contains the conclusion.

2 Translating Graph Transformation Units to SAT

Our prototypical framework has implemented *edge labeled directed graphs without multiple edges* and with a finite node set. For a finite set Σ of labels, such a graph is a pair $G = (V_G, E_G)$ where $V_G = \{1, \ldots, n\} = [n]$ for some natural number n is a finite set of *nodes*, numbered from 1 to n, and $E_G \subseteq V_G \times \Sigma \times V_G$ is a set of *labeled edges*. n is called the *size* of G. Nodes with labeled loops are drawn as labeled nodes. Figure 1 shows an example for such a graph. Furthermore, we use injective graph morphisms for the matching of subgraphs in graphs. Such a morphisms g from a graph G to a graph H is an injective mapping between the nodes and the edges of G and H, respectively.

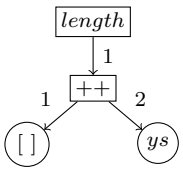

Figure 1: A graph

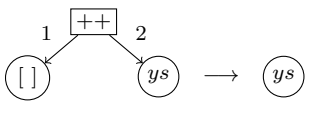

Figure 2: The rule $(++)_1$

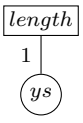

Figure 3: Result of applying $(++)_1$

For the graph transformation we use simplified DPO rules [4] which only add and delete edges. To handle the addition and deletion of nodes we use a simple trick: deleted and unused nodes are marked with a special label. By removing these labels, one can add the corresponding nodes[1]. Such a *rule* $r = (L \to R)$ consists of two graphs: the *left-hand side* L and the *right-hand side* R where the node set remains invariant, i.e. $V_L = V_R$. The *rule application* of r to a graph G works as follows: Search for a subgraph H that is isomorphic to L, i.e. find a match $g(L)$ in G. If such a subgraph is found, delete the edges of $g(L)$ and add the edges of $g(R)$. In Figure 2 one can find an example for a graph transformation rule which represents the first Haskell equation of concatenation (see Section 3). A rule application is also called a *direct derivation* and the iteration of direct derivations is called a *derivation*. Figure 3 shows the result of applying the rule $(++)_1$ to the graph in Figure 1.

A *graph transformation unit* [9] is a system $gtu = (I, P, C, T)$ where I and T are graph class expressions to specify the *initial* and *terminal* graphs, P is a set of rules, and C is a control condition. The graph transformation unit gtu specifies all derivations from initial to terminal graphs, which are allowed by the control condition. Single graphs can be used as initial or terminal graph class expressions, i.e. such graphs specify themselves. *Control conditions* guide the rule application and restrict their nondeterminism. We use regular expressions as control conditions where rules are the basic components. A rule set $P = \{r_1, \ldots, r_n\}$ abbreviates the expression $r_1 | \ldots | r_n$ where $|$ is the alternation operator. By omitting the control condition, we denote that each rule can be applied in every step.

For employing SAT solving, the derivation process of graph transformation units is described via propositional formulas where a satisfying assignment represents one of the derivations of the graph transformation unit from initial to terminal graphs. Every propositional formula with variable set $\{edge(e,k) \mid e \in [n] \times \Sigma \times [n], k \in [m]\}$ represents a sequence $G_1, \ldots, G_m$ of graphs for each variable assignment f satisfying the formula, i.e. the graph G_k contains the edge e if and only if $f(edge(e,k)) = TRUE$. Graph class expressions are also expressed via propositional formulas. For example, a single initial graph G can be described via the formula $\bigwedge_{(v,a,v') \in E} edge(v,a,v',0) \wedge \bigwedge_{(v,a,v') \in ([n] \times \Sigma \times [n]) - E} \neg edge(v,a,v',0)$.

The application of a rule $r = (L \to R)$ to a graph G_{k-1} with respect to a graph morphism $g \in \mathcal{M}(r,n)$[2] is then expressed by the following five formulas[3]

- $\mathsf{morph}(\mathsf{r},\mathsf{g},\mathsf{k}) = morph(r,g,k) \leftrightarrow \bigwedge_{(v,a,v') \in E_L} edge(g(v),a,g(v'),k-1)$,
- $\mathsf{rem}(\mathsf{r},\mathsf{g},\mathsf{k}) = rem(r,g,k) \leftrightarrow \bigwedge_{(v,a,v') \in E_L - E_R} \neg edge(g(v),a,g(v'),k)$,
- $\mathsf{add}(\mathsf{r},\mathsf{g},\mathsf{k}) = add(r,g,k) \leftrightarrow \bigwedge_{(v,a,v') \in E_R} edge(g(v),a,g(v'),k)$,
- $\mathsf{keep}(\mathsf{r},\mathsf{g},\mathsf{k}) = keep(r,g,k) \leftrightarrow (\bigwedge_{(v,a,v') \notin g(E_L \cup E_R)} edge(v,a,v',k-1) \leftrightarrow edge(v,a,v',k))$[4],
- $\mathsf{apply}(\mathsf{r},\mathsf{g},\mathsf{k}) = apply(r,g,k) \leftrightarrow (morph(r,g,k) \wedge rem(r,g,k) \wedge add(r,g,k) \wedge keep(r,g,k))$.

The formula morph describes that g is a graph morphism from L to G_{k-1}. The removal of the images of every edge of the left-hand side L from G_{k-1} is expressed by rem. The addition of edges of the right-hand side R is described by add. That edges that have been neither deleted nor added must be kept, corresponds to the formula keep. Finally, apply describes the whole application of r to G_{k-1} with respect to g. All formulas can be converted to conjunctive normal form in at most quadratic time.

[1] For reasons of readability, we allow in drawings the deletion of nodes instead of using a special label.

[2] For a rule $r = (L \to R)$ the set of injective graph morphisms from $[size(r)]$ to the set of nodes $[n]$ is denoted by $\mathcal{M}(r,n)$.

[3] Please note that in the following the expressions in bold typewriter font abbreviate propositional formulas whereas the italic font is used for variables.

[4] $g(E_L \cup E_R) = \{(g(v),a,g(v')) \mid (v,a,v'), \in E_L \cup E_R\}$

The semantics of graph transformation units can be expressed via a rather complex propositional formula. Let $gtu = (I, C, P, T)$ be a graph transformation unit. Then for all natural numbers m, n and for all $r_1 \cdots r_m \in L(C)(m)$[5] a derivation of length m from an initial graph of size n that follows $r_1 \cdots r_m$ is expressed via $\mathsf{f_{gtu}}(\mathsf{n}, \mathsf{r_1} \cdots \mathsf{r_m}) = I(n,0) \wedge \mathsf{derivation}(\mathsf{n}, \mathsf{r_1} \cdots \mathsf{r_m}) \wedge T(n,m)$ where $\mathsf{derivation}(\mathsf{n}, \mathsf{r_1} \cdots \mathsf{r_m}) = \bigwedge_{k=1}^{m} \bigwedge_{g \in M(r_k,n)} (\mathsf{apply}(\mathsf{r_k}, \mathsf{g}, \mathsf{k}) \wedge \mathsf{morph}(\mathsf{r_k}, \mathsf{g}, \mathsf{k}) \wedge \mathsf{rem}(\mathsf{r_k}, \mathsf{g}, \mathsf{k}) \wedge \mathsf{add}(\mathsf{r_k}, \mathsf{g}, \mathsf{k}) \wedge \mathsf{keep}(\mathsf{r_k}, \mathsf{g}, \mathsf{k})) \wedge \bigvee_{g \in M(r_k,n)} apply(r_k, g, k)$. The set of all derivations of length m from a graph of size n is described by $\mathsf{f_{all}}(\mathsf{n}, \mathsf{m}) = \bigvee_{r_1 \cdots r_m \in L(C)(m)} \mathsf{f_{gtu}}(\mathsf{n}, \mathsf{r_1} \cdots \mathsf{r_m})$ where each of the subformulas $\mathsf{f_{gtu}}$ can be generated and solved separately. Furthermore, the formula is bounded by a polynomial regarding the number of literals. Therefore, we can use a SAT solver because they are only feasible if the size of the input formula is bounded by a polynomial.

3 Structural Induction via Graph Transformation Units

We want to sketch the main ideas of how graph transformation can be employed for structural induction proofs by verifying a well-known list property in our framework. The corresponding implementation is currently in a prototypical stage and supports a small subset of Haskell types (Integers, lists) and functions (length, (++), reverse, (+), (:)). Further aspects of Haskell like higher-order functions or user-defined data structures will be supplemented in a coming version.

The function `(++)` given by the equations `(++) [] ys = ys` and `(++) (x:xs) ys = x : xs ++ ys` (abbreviated with $(++)_1$ and $(++)_2$, respectively) can be translated easily into graph transformational rules, where rectangles are used for function names and circles for constants and variables (see Figures 2 and 4). The outgoing edges of nodes with function names are labeled with the argument positions of the corresponding function arguments because otherwise the terms `xs ++ ys` and `ys ++ xs` would have the same graph representation. As one can see,

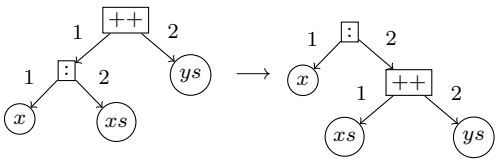

Figure 4: The rule $(++)_2$

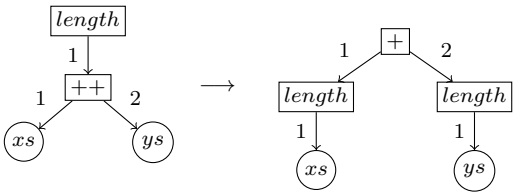

Figure 5: The rule *hypothesis*

Haskell terms are represented by trees where the outermost function name is the root and constant names and variables, respectively, are the leafs. In the following, we denote for each Haskell term t by $tree(t)$ the corresponding term tree. The function `length` has also a simple rule representation which can be generated in the same way as shown above. To simplify the example, we are using the following definition of `length` instead of the Prelude definition in Data.List.genericLength: `length [] = 0` and `length (x:xs) = 1 + length xs` (abbreviated with $length_1$ and $length_2$, respectively).

One interesting property to be proven by a structural induction proof could be, whether the concatenation of two lists preserves their separate lengths, i.e.

$$length\ (xs\ +\!+\ ys) = length\ xs\ +\ length\ ys \text{ for all lists } xs \text{ and } ys.$$

[5]For each control condition C and each natural number m, the finite sublanguage $L(C)(m)$ describes the set of all rule sequences in C with a length of exactly m.

This can be proven by using two graph transformation units: One unit for the base case and the other for the inductive step where we perform an induction over xs. The input of both units is the left-hand side of the property, i.e. $length\ ([]\ +\!+\ ys)$ and $length\ (x : xs\ +\!+\ ys)$, respectively, and the output is the right-hand side, i.e. $length\ []\ +\ length\ ys$ and $length\ x : xs + length\ ys$, respectively. All known function equations can be used as rules, in the inductive step the hypothesis is added as additional rule. If it is found in both cases a derivation from the initial to the terminal graph, the property has been proven. Why is that? Obviously, the rule arrows can be reversed, i.e. one can apply the rules also from right to left, and, thus, the derivation process can be reversed, i.e. it also exists a derivation from the terminal to the initial graph. Hence, both sides are equal and the property has been verified.

The unit for the base case has the following form[6]: $base = (I_{base}, P_{base}, T_{base})$ where $I_{base} = tree(length\ ([]\ +\!+\ ys))$, $P_{base} = \{(++)_1, length_1, identity_{add}\}$, and $T_{base} = tree(length\ []\ + length\ ys))$. $identity_{add}$ is the rule counterpart of $0 + x = x$. The unit for the inductive step has two additional rules, the hypothesis (see Figure 5) and a rule for the commutative property of addition: $step = (I_{step}, P_{step}, C_{step}, T_{step})$ where $I_{step} = tree(length\ (x : xs\ +\!+\ ys))$, $P_{step} = \{length_2, (++)_2, hypothesis, commutative_{add}\}$, $C_{step} = (P_{step} - \{hypothesis\})^*; hypothesis; (P_{step} - \{hypothesis\})^*$, and $T_{step} = tree(length\ x : xs\ +\ length\ ys)$. One successful derivation for the inductive step can be found in Figure 6. The first graph is the initial graph, the second graph is derived by applying the rules $(++)_2$ and $length_2$, the third graph is the result of the application of the $hypothesis$ rule, and, finally, the fourth graph is the terminal graph derived by applying the rules $commutative_{add}$ and $length_2$.

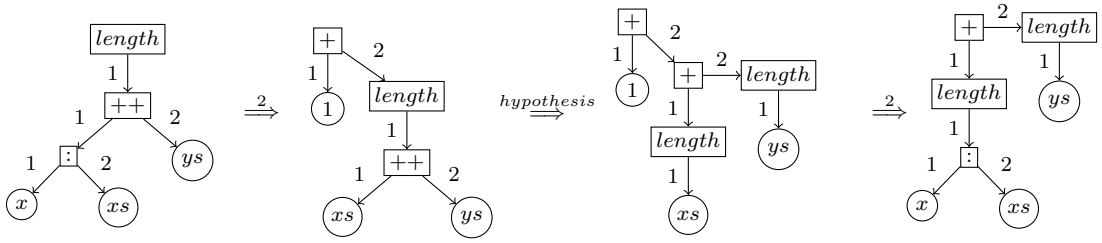

Figure 6: Inductive step performed by the graph transformation unit *step*

An open issue is why we use a translation of graph transformation to SAT. In each derivation step several rules and matches can be applied, but only some of them yield a proper result and the rest leads in the wrong direction. For example, it is possible that in the base case the rule $identity_{add}$ is applied infinitely often because of the automation of the verification process. So one would need a backtracking mechanism to continue at an other point of the derivation. By using a translation to SAT the whole state space up to a certain bound is generated and we need no backtracking mechanism. In the base case, the translation to SAT[7] generates in each separate solving step subformulas consisting of circa 5.000 variables and circa 450.000 clauses for the bound $\#P_{base}$. These subformulas are each solved in less than 1 sec. To find a successful derivation all possible derivations are checked leading to an overall computation time of less than 10 sec. These results make us optimistic that further investigations will lead to a convenient and useful verification framework for Haskell.

[6]Please note, that we choose for reasons of comprehensibility an adequate and simple rule set in both cases. In general, the rule set consists of much more function equations and properties.

[7]Our tests have been conducted under Ubuntu 10.04 LTS on an AMD 2.0 GHz with 4GB RAM.

4 Conclusion

In this paper, we have introduced an approach for verifying automatically Haskell programs by means of graph transformation units and SAT solving. Our first experiments with this technique nurture the hope that it can be employed for verification proofs in Haskell.

We are planning to undertake further investigations in different directions in future. In Section 3, we sketch our first ideas for the verification framework, in a next step we want to provide it with a formal foundation. Furthermore, we want to facilitate the input of properties to be proven by using QuickCheck [3]. This shall enable the user to work only with Haskell, i.e. to write his program and the properties in Haskell, whereas the translation to graph transformation and SAT and, finally, the verification are done fully automatically without user interaction. Moreover, we want to supplement term graph rewriting [12, 10] to our framework and want to compare it with the current proposal.

References

[1] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *TACAS*, LNCS, pages 193–207. Springer, 1999.

[2] T. H. Brus, Marko C. J. D. van Eekelen, M. O. van Leer, and Marinus J. Plasmeijer. Clean: A language for functional graph writing. In Gilles Kahn, editor, *FPCA*, volume 274 of *LNCS*, pages 364–384. Springer, 1987.

[3] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In Martin Odersky and Philip Wadler, editors, *ICFP*, pages 268–279. ACM, 2000.

[4] Andrea Corradini, Hartmut Ehrig, Reiko Heckel, Michael Löwe, Ugo Montanari, and Francesca Rossi. Algebraic approaches to graph transformation part I: Basic concepts and double pushout approach. In Grzegorz Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations*, pages 163–245. World Scientific, 1997.

[5] Niklas Eén and Niklas Sörensson. An extensible SAT solver. In *SAT 2003*, volume 2919 of *LNCS*, pages 502–518. Springer, 2004.

[6] Marcus Ermler, Hans-Jörg Kreowski, Sabine Kuske, and Caroline von Totth. From graph transformation units via MiniSat to GrGen.NET. In Andy Schürr, Dániel Varró, and Gergely Varró, editors, *AGTIVE 2011*, volume 7233 of *LNCS*, pages 153–168. Springer, 2012.

[7] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[8] Hans-Jörg Kreowski, Sabine Kuske, and Grzegorz Rozenberg. Graph transformation units – An overview. In Pierpaolo Degano, Rocco De Nicola, and José Meseguer, editors, *Concurrency, Graphs and Models*, volume 5065 of *LNCS*, pages 57–75. Springer, 2008.

[9] Hans-Jörg Kreowski, Sabine Kuske, and Robert Wille. Graph transformation units guided by a SAT solver. In Hartmut Ehrig, Arend Rensink, Grzegorz Rozenberg, and Andy Schürr, editors, *ICGT 2010*, volume 6372 of *LNCS*, pages 27–42. Springer, 2010.

[10] Detlef Plump. Term graph rewriting. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools*, pages 1–61. World Scientific, 1999.

[11] Arend Rensink. The GROOVE simulator: A tool for state space generation. In John L. Pfaltz, Manfred Nagl, and Boris Böhlen, editors, *AGTIVE 2003*, volume 3062 of *LNCS*, pages 479–485. Springer, 2004.

[12] M. Ronan Sleep, Rinus Plasmeijer, and Marko van Eekelen, editors. *Term Graph Rewriting. Theory and Practice*. Wiley & Sons, Chichester, 1993.