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

Marcus Ermler

University of Bremen, Department for Mathematics and Computer Science

WFLP 2013, September 11, 2013

Motivation

Aim: Application of graph transformation for the verification of Haskell programs via structural induction.

Questions:

1. Is graph transformation a useful approach in this context?
2. How to tackle the nondeterminism of function equation application in automatic verification?

Answers:

1. graph transformation has been successfully applied to term rewriting (term graph rewriting/CLEAN/SPARKLE)
2. heuristics, exhaustive search, parallelization, SAT solving

⇒ We use graph transformation units and SAT solving to verify Haskell programs.

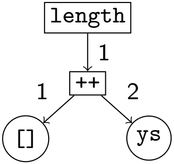
Considering a small subset of Haskell

- predefined data types like Int, Char, String, Lists
- functions defined by functions equations without guards or local definitions
- higher order functions
- in preparation: lambda abstractions, control structures, self-defined data types

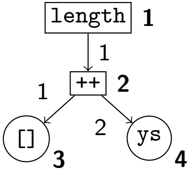
Translating Haskell programs into Trees

- edge labeled directed graphs without multiple edges and with a finite set of typed node. For a finite set Σ of labels and a set $\mathcal{T}$ of types: $G = (V, E, t)$ where $V = \{1, \ldots, n\} = [n]$, $E \subseteq V \times \Sigma \times V$, and $t \colon V \to \mathcal{T}$.
- rectangles for function names; outermost function name is the root; circles for constants and variables (leafs)
- outgoing edges are labeled with argument positions

Example: `length ([] ++ ys)` is expressed via



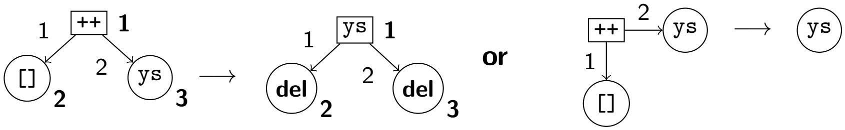
or



Graph transformational rules

- rule $r = (L \to R)$: left-hand side L and right-hand side R
- translation of Haskell function equations $l = r$ into rules: $tree(l) \to tree(r)$
- for technical reasons: only edge addition and deletion, node addition and deletion is realized via a simple trick
  $\Rightarrow$ in drawings: node labels instead of labeled loops

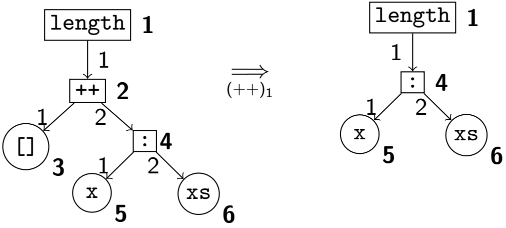
Example: The function equation `[] ++ ys = ys` (denoted by $(++)_1$) is translated into



Rule application and derivations

- injective graph morphisms for matching of subgraphs (structure-, label-, and type-preserving morphisms)
- application of a rule: find a match of $g(L)$ in a graph G, delete the edges of $g(L)$, and add the edges of $g(R)$

Example: mapping $g = \{1 \mapsto 2, 2 \mapsto 3, 3 \mapsto 4\}$ for $(++)_1$



Graph transformation units

- graph transformation units: $gtu = (I, P, C, T)$ where I and T are graph class expressions, R is a set of rules, and C is a control condition
- graph class expressions: for example, the class of all undirected graphs, also single graphs allowed
- control conditions: guide the rule application, restrict the nondeterminism of units; we use regular expressions
- Semantics of $gtu = (I, P, C, T)$: all derivations from initial to terminal graphs that are allowed by the control condition $\Rightarrow$ such derivations are called *successful*

From graphs to SAT

- graphs in derivation steps are represented via variables for their edges: $E(n,m) = \{edge(v,a,v',k) \mid (v,a,v') \in [n] \times \Sigma \times [n], k \in [m]\}$ where n is the graph size and m the maximum derivation step
- single graph in the kth derivation step expressed via edges that are in the graph and edges that are not in the graph

$$\text{graph}(\mathrm{G}, \mathrm{k}) = \bigwedge_{(v,a,v') \in E_G} edge(v,a,v',k) \quad \wedge \quad \bigwedge_{(v,a,v') \in ([n] \times \Sigma \times [n]) - E_G} \neg edge(v,a,v',k).$$

From graph rewriting to SAT

- rule application is expressed via five formulas: morph, rem, add, keep, and apply
- The matching of a rule r in a graph G_{k-1} with respect to a mapping g is expressed via:

$$\mathrm{morph}(\mathrm{r}, \mathrm{g}, \mathrm{k}) = \mathit{morph}(r, g, k) \leftrightarrow \bigwedge_{(v,a,v') \in E_L} \mathit{edge}(g(v), a, g(v'), k-1),$$

- further formulas for derivation steps, single derivations, and all derivations up to a certain bound
- morph, rem, add, keep, and apply can be converted in at most quadratic time to CNF, all other formulas are already in CNF

Proving properties via graph transformation (1)

Hypothesis is given as list property: $p(\texttt{xs}) = (\texttt{l(xs)=r(xs)})$

Definition

Let $p(\texttt{xs}) = (\texttt{l(xs)=r(xs)})$ be a list property with induction variable xs and let P be a set of graph transformational rules representing Haskell function equations. Then the base case unit and the inductive step unit are defined as follows.

- $base(p(\texttt{[]})) = (tree(\texttt{l([])}), P, P^*, tree(\texttt{r([])}))$
- $step(p(\texttt{x:xs})) = (tree(\texttt{l(x:xs)}), P_{step}, C_{step}, tree(\texttt{r(x:xs)}))$ where

$$\begin{aligned} hyp_1 &= (tree(\texttt{l(xs)}) \to tree(\texttt{r(xs)})),\\ hyp_2 &= (tree(\texttt{r(xs)}) \to tree(\texttt{l(xs)})),\\ P_{step} &= P \cup \{hyp_1, hyp_2\}, \text{ and}\\ C_{step} &= P^* \ ; \ (hyp_1 \mid hyp_2) \ ; \ P^*. \end{aligned}$$

Proving properties via graph transformation (2)

Theorem

Let p(xs) be a property, base(p([])) be a base case unit, and step(p(x:xs)) be an inductive step unit. If there is a successful derivation in base(p([])) as well as in step(p(x:xs)), then the property holds.

Example: a length property

Hypothesis: `length (xs ++ ys) = length xs + length ys` for all lists `xs` and `ys`.

Base case unit

<u>Base case:</u> `length ([] ++ ys) = length [] + length ys`

base

initial: $\mathit{tree}(\texttt{length ([] ++ ys)})$

rules:

$\mathit{tree}(\texttt{[] ++ xs}) \rightarrow \mathit{tree}(\texttt{xs})$ $[(++)_1]$

$\mathit{tree}(\texttt{0}) \rightarrow \mathit{tree}(\texttt{length []})$ $[\mathit{length}'_1]$

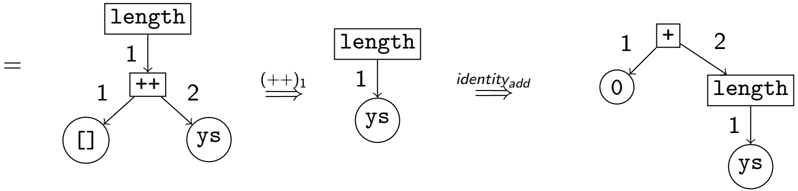
$\mathit{tree}(\texttt{x}) \rightarrow \mathit{tree}(\texttt{0 + x})$ $[\mathit{identity}_{add}]$

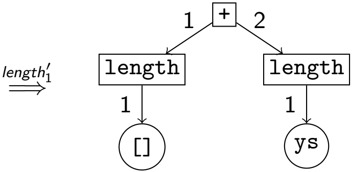
cond.: $\left((++)_1 \mid \mathit{length}'_1 \mid \mathit{identity}_{add}\right)^*$

terminal: $\mathit{tree}(\texttt{length [] + length ys})$

A sample derivation for proving the base case

$I_{base} = tree($`length ([] ++ ys)`$)$





$= tree($`length [] + length ys)`$) = T_{base}$

Inductive step unit

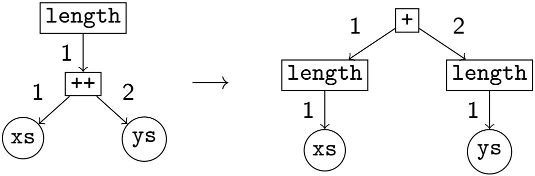
Inductive step: `length (x:xs ++ ys) = length (x:xs) + length ys`, `x::a`

initial:	*tree*(`length (x:xs ++ ys)`)	
rules:	*tree*(`x:xs ++ ys`) → *tree*(`x:(xs++ys)`)	[$(++)_2$]
	tree(`length (x:xs)`) $\overset{\leftarrow}{\rightarrow}$ *tree*(`1 + length xs`)	[$length_2$ + $length'_2$]
	tree(`x + (y + z)`) → *tree*(`(x + y) + z`)	[$assoc_{add}$]
	hypothesis	
cond.:	$((++)_2 \mid length_2 \mid length'_2 \mid assoc_{add})^*$; *hypothesis* ; $((++)_2 \mid length_2 \mid length'_2 \mid assoc_{add})^*$	
terminal:	*tree*(`length x:xs + length ys`)	

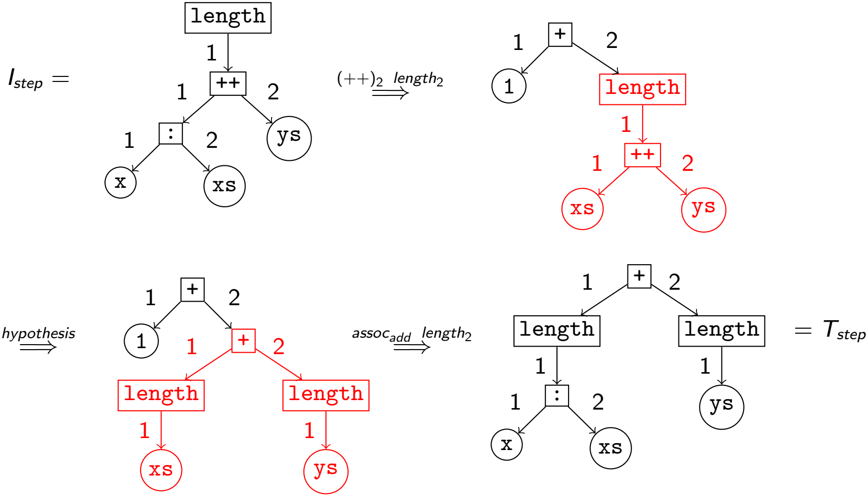
Hypothesis

Hypothesis: `length (xs ++ ys) = length xs + length ys`
for all lists `xs` and `ys`.

⇒ the *hypothesis*-rule:



A sample derivation for the inductive step



Experiments

Lemmata[1]:

1. `length (xs ++ ys) = length xs + length ys`
2. `xs ++ (ys ++ zs) = (xs ++ ys) ++ zs`
3. `xs ++ [] = xs`
4. `[] ++ (xs ++ []) = xs`

Lemma	Strategy	Base case	Inductive step
1.	induction	8 sec	90 sec
2.	induction	0.3 sec	17 sec
3.	induction	1 sec	1 sec
4.	direct proof	0 sec	

[1] tested under Ubuntu 10.04 LTS on an AMD 2.0 GHz with 4GB RAM where lemma 4 is proven by a direct proof via lemma 3.

Summary

- graph transformational approach for structural induction proofs
- experiments nurture the hope that our approach can be employed for verification proofs

Outlook:

- more Haskell features step-by-step or via preprocessing described in Giesl et al., *Automated termination proofs for Haskell by term rewriting*, 2011
- automatic translation of functional programs into rules and units (in preparation)

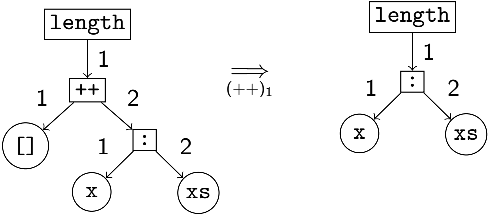
Questions?

Previous work

- translation of graph transformational derivation process into propositional formulas (presented on ICGT 2010)
- introducing SATaGraT (**SAT** solver **a**ssists **Gra**ph **T**ransformation Engine) on AGTIVE 2011
- using SAT solving to find a successful derivation
- applied on NP-complete graph problems
- bottleneck: conversion into conjunctive normal form

Simple graph transformation unit

Very simple example:



- $I = \mathit{tree}($`length ([] ++ x:xs)`$)$
- $P = \{(++)_1\}$
- $C = (++)_1$
- $T = \mathit{tree}($`length (x:xs)`$)$

From graph rewriting to SAT (1)

The application of a rule r to a graph G_{k-1} with respect to a mapping g is expressed via

- matching:

$$\mathrm{morph(r,g,k)} = \mathit{morph}(r,g,k) \leftrightarrow \bigwedge_{(v,a,v')\in E_L} \mathit{edge}(g(v),a,g(v'),k-1),$$

- edge deletion:

$$\mathrm{rem(r,g,k)} = \mathit{rem}(r,g,k) \leftrightarrow \bigwedge_{(v,a,v')\in E_L - E_R} \neg \mathit{edge}(g(v),a,g(v'),k),$$

- edge addition:

$$\mathrm{add(r,g,k)} = \mathit{add}(r,g,k) \leftrightarrow \bigwedge_{(v,a,v')\in E_R} \mathit{edge}(g(v),a,g(v'),k),$$

- kept edges:

$$\mathrm{keep(r,g,k)} = \mathit{keep}(r,g,k) \leftrightarrow \Big(\bigwedge_{(v,a,v')\notin g(E_L \cup E_R)} \big(\mathit{edge}(v,a,v',k-1) \leftrightarrow \mathit{edge}(v,a,v',k)\big)\Big)$$

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

From graph rewriting to SAT (2)

- whole application of a rule r to G_{k-1} with respect to graph morphism g is described by

$$\mathsf{apply}(\mathsf{r},\mathsf{g},\mathsf{k}) = apply(r,g,k) \leftrightarrow \Big(morph(r,g,k) \wedge rem(r,g,k) \wedge add(r,g,k) \wedge keep(r,g,k)\Big)$$

Theorem

$G_{k-1} \underset{r,g}{\Longrightarrow} G_k$ if and only if it there is a satisfying assignment to $\mathsf{graph}(\mathsf{G}_{\mathsf{k}-1},\mathsf{k}-1) \wedge \mathsf{apply}(\mathsf{r},\mathsf{g},\mathsf{k}) \wedge \mathsf{graph}(\mathsf{G}_{\mathsf{k}},\mathsf{k})$.

- further formulas for derivation steps, single derivations, and all derivations up to a certain bound
- morph, rem, add, keep, and apply can be converted in at most quadratic time to CNF, all other formulas are already in CNF