日本ソフトウェア科学会第 40 回大会 (2023 年度) 講演論文集

Automatic Correctness Checking of Haskell's Rewrite Rules: Theory and Practice

Makoto Hamana

We present a theoretical basis for automatic correctness checking of Haskell 's rewrite rules. We also demonstrate a new tool which implements advanced rewriting techniques as a GHC plugin to ensure the correctness of GHC 's rewrite rules. Key to our method is ensuring local confluence and strong normalisation of rewrite rules automatically.

1 Introduction

The Glasgow Haskell Compiler (GHC) is an open source compiler for the Haskell functional language. It has the feature of *rewrite rules* to specify optimising transformations [9]. Rewrite rules have been widely used in many libraries.

But there has been no formal theory that formalises the rules and establishes their correctness. We present a variant of the polymorphic λ-calculus extended with rewrite rules and equations. By giving a rewriting semantics of total Haskell, we show that rewrite rules and the functional programming language based on the polymorphic λ-calculus are modelled uniformly in a single framework, which allows us to reason about the correctness of rewrite rules. We also demonstrate the tool ReCheck which automatically checks the correctness of Haskell rewrite rules by local confluence and termination checking.

The aim of this paper is to establish the foundations of the correctness problems of rewrite rules in GHC by giving a rewriting theoretical framework that models Haskell.

2 Rewrite rules in GHC

To illustrate a problem concerning the correctness of a rule, we consider a Haskell program with a rewrite rule as shown in Fig. 1. The code inside the {`-# ... #-`} is called a *pragma*. Inside the `RULES` pragma there is a rule called `"one"` which tells GHC to rewrite an expression "`f` t" to "1". But in the program the function f is defined to always return 0. What is the output of `main`?

Calling GHC with `-fenable-rewrite-rules` applies the rewrite rule before compilation, so the output is `1`. This shows that the rewrite rule inadvertently changes the meaning of the original program. Namely, ***the rule is wrong***. Since the rule feature was introduced as an optimisation tool, it should not change the meaning of the original program.

More than 20 years have passed since rewrite rules were first proposed for GHC [9]. Observations have shown that is useful in very many libraries[†1], whose efficiency often depends crucially on rewrite rules. For example, GHC's own base library contains more than 100 rules. However, GHC does not currently attempt to check the correctness of rules in a program. This is dangerous.

*Haskell における書換え規則の正当性自動検証、その理論と実践

This is a non-peer-reviewed paper, which is an extended abstract of a presentation at JSSST 20th Symposium. Copyrights belong to the Author.

浜名 誠, 群馬大学情報学部, Faculty of Informatics, Gunma University.

†1 https://www.aosabook.org/en/ghc.html

```
-- Diverge.hs
f :: Int -> Int
f x = 0

{-# RULES
"one" forall x. f x = 1
#-}
main = print (f 2)
```

Fig. 1 Is this rule correct?

```
-- Append.hs
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)

{-# RULES
"assoc" forall xs ys zs. (xs ++ ys) ++ zs = xs ++ (ys ++ zs)
#-}
```

Figure 2. Append program with the associativity rule

3 How to ensure the correctness of rules

Ideally, we expect the correctness of rules to be checked automatically before compilation. In the case of a simple program like the one in Figure 1, this may not be difficult. But in general, the problem gets complicated.

Consider another example of a well-known append function in Figure 2, with the associativity rule for append. This is a typical use of rewriting rules derived from natural properties of functions. To say that the rule `"assoc"` is correct is to ***prove*** that this equation holds for arbitrary lists. As is well known in the *Algebra of Programming* methodology [2] [3] [6], to prove this requires structural induction on lists. Proving by induction involves appropriate applications of the induction hypothesis, and sometimes requires non-trivial lemmas. A clever prover may be able to prove such properties automatically on inductive types. However, since Haskell is a higher-order polymorphic functional programming language, programs and rules can also involve higher-order polymorphic functions. To prove equations involving higher-order functions, one may need to use more sophisticated proof principles or denotational semantics. How to automate such proofs is not obvious and may be difficult.

In this paper, we address this problem by using novel rewriting techniques in the System F_{RE} framework. We provide a method for automatically proving the correctness of rules without the need for complex proof principles.

4 Equational Logic Approach

To discuss the correctness of a rewrite rule in a formal setting, we need a suitable framework for dealing with meaning and equality. One way is to use denotational semantics and insist that a rule $l \Rightarrow r$ preserves meaning as $[\![l]\!] = [\![r]\!]$. However, this is not suitable for automatic checking because it requires the formalisation of various mathematical structures, such as complete partial orders. Moreover, it is unclear how automatic proofs are possible for this purpose. Instead, we use a logical approach to reasoning about equations. We use equational logic [5] [10] with an appropriate higher-order and polymorphic typed extension. A rule can be thought of as an oriented equation. Therefore we formalise "a rule $l \Rightarrow r$ is correct w.r.t. a program $\mathcal{P}$" as "an equation $l = r$ is provable under the assumption $\mathcal{P}$".

For example, we consider again the example of append in Fig. 2. We regard the program as a set of equational axioms

$$\mathcal{P} = \left\{ \begin{array}{ll} \forall ys :: [a]. & [] +\!\!+ ys = ys \\ \forall x :: a.\ xs, ys :: [a]. & (x : xs) +\!\!+ ys = x : (xs +\!\!+ ys) \end{array} \right\}$$

Then we can formulate the correctness of an equation $l = r$ as the provability of it from $\mathcal{P}$.

The inference rules of the ordinary first-order many-sorted equational logic EL_1 consist of reflexivity, transitivity, symmetry, congruence, substitution, and axiom rules [5] [10] [1]. We denote by $\mathcal{P} \vdash_{\mathsf{EL}_1} s = t$ the deducibility of $s = t$ from the axioms of $\mathcal{P}$ by EL_1 and call $s = t$ an *equational theorem* of $\mathcal{P}$.

It is important to note that in EL_1 associativity

is not deducible from $\mathcal{P}$:

$$\mathcal{P} \not\vdash_{\mathsf{EL}_1} \forall xs, ys, zs :: [a].\, (xs +\!\!+ ys) +\!\!+ zs = xs +\!\!+ (ys +\!\!+ zs) \quad (1)$$

What kind of associativity is deducible from $\mathcal{P}$ in EL_1 are all the *ground instances*:

$$\mathcal{P} \quad \vdash_{\mathsf{EL}_1} \quad (xs\theta +\!\!+ ys\theta) +\!\!+ zs\theta = xs\theta +\!\!+ (ys\theta +\!\!+ zs\theta) \quad (2)$$

where θ is an arbitrary substitution that assigns a ground list (i.e. a list containing no variable) to each variable. The notation $xs\theta$ denotes a term obtained by the application of θ to xs.

To see why (2) is deducible in EL_1, consider the substitution $\theta : xs, ys, zs \mapsto []$. Then $([] +\!\!+ []) +\!\!+ [] = [] = [] +\!\!+ ([] +\!\!+ [])$ is formally provable (by constructing a proof tree) from $\mathcal{P}$ in EL_1 by using the axiom instantiation of the first equation in $\mathcal{P}$, congruence, symmetry, and transitivity.

For the case of substitution $\theta : xs \mapsto (n : ns)$, etc., where $n : ns$ is a ground list, the associativity is similarly proved using a previously deduced associativity for the list ns. This is nothing but a proof by structural induction on lists using an induction hypothesis. This kind of theorems (such as (2)) is known as *inductive theorems* [1][10] in theory of term rewriting and universal algebra. In summary, under the axioms $\mathcal{P}$, $\forall xs, ys, zs :: [a].\, (xs +\!\!+ ys) +\!\!+ zs = xs +\!\!+ (ys +\!\!+ zs)$ is not an equational theorem of $\mathcal{P}$, but an inductive theorem of $\mathcal{P}$. Evevy equational theorem is an inductive theorem, but not vice versa.

For this reason, we formulate the correctness of $l \Rightarrow r$ w.r.t. $\mathcal{P}$ as a property whether $l = r$ is an *inductive theorem* of $\mathcal{P}$.

5 Rewriting approach to inductive equational reasoning

For automatic rule checking, we need an algorithm to decide whether an equation is an inductive theorem or not. Ordinary equational theories can be decided if given axioms are expressed as terminating and confluent rules. This is a well-known rewriting approach, as Knuth and Bendix solved the word problem for groups by constructing such axioms using the completion technique [7]. But now we need to prove all the *ground instances of an equation* (such as (2)), which requires structural induction. This is different from proving a universally quantified equation (such as (1)) in EL_1. Structural induction could be taken as an additional inference rule of EL_1. But [8] proved that equational logic with structural induction is not complete. It follows that inductive theorems on natural numbers are not recursively enumerable [4]. In this paper we take a different approach, which establishes the following new theorem for deciding inductive theorems.

The Rule Correctness Theorem. Let $\mathcal{P}$ be a total Haskell program and $l \Rightarrow r$ a rewrite rule. Suppose that

(i) $\mathcal{P} \cup \{l \Rightarrow r\}$ is locally confluent, and
(ii) $\mathcal{P} \cup \{l \Rightarrow r\}$ is strongly normalising.

Then $l = r$ is an inductive theorem of $\mathcal{P}$, therefore, $l \Rightarrow r$ is correct with respect to $\mathcal{P}$.

This theorem provides a simple algorithm for deciding inductive theorems by checking (i)(ii). Namely, if we have automatic local confluence and termination (i.e. strong normalisation) checkers, then we can decide the correctness automatically. A remarkable point is that these conditions do not use structural induction. This theorem is simple in form, but powerful. In fact, we can use it to prove the correctness of the associativity of append This theorem requires checking the termination and local confluence of the union of a program and a rewrite rule. Therefore, we will develop methods to check local confluence and termination of rewrite systems for Haskell. The termination and local confluence checking are non-trivial problems because ordinary rewriting methods are restricted to first-order terms, not including λ-terms, polymorphism and inductive types. We develop an extension of the higher-order polymorphic λ-calculus System F_ω extended with rewrite rules and equations. We have also developed a syntactic criteria to decide local confluence and termination: critical pair analysis and the Polymorphism Termination Guarantee.

References

[1] Baader, F. and Nipkow, T.: *Term Rewriting and All That*, Cambridge University Press, 1998.
[2] Bird, R. and Wadler, P.: *An Introduction to Functional Programming*, Prentice Hall, 1988.
[3] Bird, R. and Moor, O. D.: *Algebra of Programming*, Prentice-Hall, 1996.
[4] Davis, M., Matijasevic, Y., and Robinson, J.: Hilbert's tenth problem. Diophantine equations: positive aspects of a negative solution, *Proc. Sym-*

posia in Pure Math, Vol. AMS 28, 1978, pp. 323–378.

[5] Goguen, J. and Meseguer, J.: Completeness of many-sorted equational logic, *Houston Journal of Mathematics*, Vol. 11, No. 3(1985), pp. 307–334.

[6] Hutton, G.: *Programming in Haskell*, Cambridge University Press, 2016.

[7] Knuth, D. and Bendix, P.: Simple Word Problems in Universal Algebras, *Computational Problem in abstract algebra*, Pergamon Press, Oxford, 1970, pp. 263–297.

[8] Nourani, F.: On induction for programming logic: syntax, semantics and inductive closure, *Bull. EATCS*, Vol. 13(1981), pp. 51–64.

[9] Peyton Jone, S., Tolmach, A., and Hoare, T.: Playing by the rules: rewriting as a practical optimisation technique in GHC, *Haskell Workshop 2001*, 2001.

[10] Wechler, W.: *Universal algebra for computer scientists*, Springer-Verlag, 1992.