

On Improving Termination Preservability of Transformations from Procedural Programs into Rewrite Systems by Using Loop Invariants

Naoki Nishida and Takumi Kataoka

Graduate School of Information Science, Nagoya University
Furo-cho, Chikusa-ku, 4648603 Nagoya, Japan
nishida@is.nagoya-u.ac.jp
kataoka@trs.cm.is.nagoya-u.ac.jp

Abstract

Recently, to analyze procedural programs by using techniques in the field of term rewriting, several transformations of a program into a rewrite system have been developed. Such transformations are basically complete in the sense of computation, and e.g., termination of the rewrite system ensures termination of the program. However, in general, termination of the program is not preserved by the transformations and, thus, the preservation of termination is a common interesting problem. In this paper, we discuss the improvement of a transformation from a simple procedural program over integers into a constrained term rewriting system by appending loop invariants to loop conditions of “while” statements so as to preserve termination as much as possible.

1 Introduction

Recently, program verification methods for procedural programs using term rewriting techniques have been developed e.g., [8, 12, 16, 5]. In these methods, to verify a property (e.g., termination), a program is transformed into a rewrite system, and then the rewrite system is verified instead of the program. A common problem in these methods is that in general they do not preserve termination of the program when transforming it into a rewrite system. The transformed rewrite system covers all computations of the program, but can reduce a term which does not correspond to any intermediate state for computation of the program.

This problem is crucial in verifying the equivalence of functions by using *inductive theorem proving* methods [15, 6], which are based on *rewriting induction* [13]. Rewriting induction requires a rewrite system to be terminating. For this reason, when termination of a program is not preserved in transforming it, we cannot apply the methods based on rewriting induction to the verification of the transformed rewrite system.

In this paper, we aim at improving transformations of simple procedural programs over integers, which are based on “while” programs, into rewrite systems so as to preserve termination as much as possible. To this end, We first introduce the transformation in [8] by using an example, for which the transformation does not preserve termination. Then, we show that appending a loop invariant to the loop condition of a “while” statement in the program makes the transformation preserve termination of the program. Finally, we briefly introduce how to automatically obtain the loop invariant by using an existing technique. Note that a preliminary version of this work is [10].

We deal with constrained term rewriting systems [8, 2, 14], but the idea in this paper is applicable to other frameworks (transformations and their resulting rewrite systems), e.g., *integer rewriting systems* [7, 16] and *logically constrained term rewriting systems* [11]. The improvement is useful to prove termination of a procedural program.

One may think that we can prove correctness of programs if we find loop invariants which are useful to preserve termination in applying the transformation. However, this would not be sufficient for the purpose of proving equivalence of two functions defined by programs.

This paper is organized as follows. Section 2 briefly introduces constrained TRSs. Section 3 introduces the transformation in [8] by an example. Section 4 shows an idea for improving the transformation by using loop invariants. Section 5 concludes this paper.

2 Preliminaries

In this section, we briefly recall some basic notions and notations of *constrained TRSs* [8, 2, 14]. We assume familiarity of readers with the basic notions and notations of term rewriting (see [1]). Throughout this paper, we use $\mathcal{V}$ as a countably infinite set of variables.

We restrict constrained TRSs to be over a fixed interpretable signature for integers. We use function symbols 0 (a constant), s (unary), p (unary), and $plus$ (binary), which are interpreted to in the usual way: 0 means 0, $s(x)$ means $x + 1$, $p(x)$ means $x - 1$, and $plus(x, y)$ means $x + y$. In the following, we write $t_1 + t_2$ instead of $plus(t_1, t_2)$. Moreover, we use the usual comparison operators (e.g., $=$, $\leq$, $<$) as predicates with the usual semantics, and use quantifier-free formulas over the interpreted symbols, the predicates, and variables, as *constraints*.

Let $\mathcal{F}$ be a signature which is basically specified by users. A *constrained rewrite rule* is a triple (l, r, ϕ) , written as $l \rightarrow r \llbracket \phi \rrbracket$, such that l and r are terms over $\mathcal{F}$ and the interpreted symbols, l is not a variable, ϕ is a satisfiable constraint, and $\mathcal{V}ar(l) \supseteq \mathcal{V}ar(r) \cup \mathcal{V}ar(\phi)$. A *constrained term rewriting system* (constrained TRS) is a finite set $\mathcal{R}$ of constrained rewrite rules. We omit the following rules for interpreted symbols from constrained TRSs shown in this paper:

$$\{ s(p(x)) \rightarrow x, \quad p(s(x)) \rightarrow x, \quad 0 + y \rightarrow y, \quad s(x) + y \rightarrow s(x + y), \quad p(x) + y \rightarrow p(x + y) \}$$

The rewrite relation $\rightarrow_{\mathcal{R}}$ of $\mathcal{R}$ is defined as follows: $\rightarrow_{\mathcal{R}} = \{(C[l\sigma], C[r\sigma]) \mid l \rightarrow r \llbracket \phi \rrbracket \in \mathcal{R}, \phi\sigma \text{ has no symbol in } \mathcal{F}, \phi\sigma \text{ is valid}\}$. A term t is called *terminating* if there is no infinite rewrite sequence of $\rightarrow_{\mathcal{R}}$ which is starting from t . $\mathcal{R}$ is called *terminating* if every term is terminating.

3 Transforming C Programs over Integers into Constrained TRSs

In this section, we briefly introduce a transformation of simple procedural programs over integers into constrained TRSs [8].

We deal with programs of the following form:

$$\text{int } f(\text{int } x_1, \dots, \text{int } x_n) \{ D; S; \text{return } e; \}$$

where D is a sequence of declarations for local variables with initialization, S is a so-called “while” program (having assignments, “while”, and “if” statements), e is an expression, and the code is compiled successfully as a C program.

► **Example 1.** The program `sum` illustrated in Listing 1 is the one we deal with.

The transformation in [8] (cf. [5, 16]) converts programs in a natural way: constrained rewrite rules are generated so as to represent control flows of programs, and loop or branch conditions of “while” and “if” statements are added to rewrite rules as constraints. The transformation preserves termination of a program w.r.t. terms corresponding to calls of functions defined in the program.

■ **Listing 1** A program computing the sum from 0 to x

```
int sum(int x){
  int i=0, z=0;
  if(x > 0)
    while(i != x){
      i = i+1;
      z = z+i;
    }
  return z;
}
```

► **Example 2.** Consider the program `sum` in Listing 1 again. This program is transformed into the following constrained TRS:

$$\mathcal{R}_{\text{sum}} = \left\{ \begin{array}{ll} \text{sum}(x) \rightarrow u_1(x, 0, 0), & \\ u_1(x, i, z) \rightarrow u_2(x, i, z) \llbracket x > 0 \rrbracket, & u_2(x, i, z) \rightarrow u_2(x, s(i), z + s(i)) \llbracket i \neq x \rrbracket, \\ u_1(x, i, z) \rightarrow z \llbracket \neg x > 0 \rrbracket, & u_2(x, i, z) \rightarrow z \llbracket \neg i \neq x \rrbracket \end{array} \right\}$$

The function symbols u_1 and u_2 represent the “if” and “while” statements in the program `sum`. All the terms of the form `sum`(n) with an interpreted term n are terminating, but $\mathcal{R}_{\text{sum}}$ is not terminating. For example, $u_2(0, s(0), 0)$ which is not reachable from any term of the form `sum`(n) is not terminating.

4 Improving Preservation of Termination

In this section, we improve termination preservability of the transformation by appending loop invariants to loop conditions of “while” statements.

Loop invariants are constraints which are satisfied whenever the bodies of “while” statements start to be evaluated. Thanks to this feature of loop invariants, appending a loop invariant to the corresponding loop condition does not change the computation of programs at all.

► **Example 3.** Consider the program `sum` in Listing 1 again. By using the method in [3] with the SMT solver Z3 [4], we obtain the constraint “ $x > i$ ” as a loop invariant of the “while” statement in the program. Thus, by appending the loop invariant to the loop condition “ $i \neq x$ ” of the “while” statement, we obtain the equivalent program illustrated in Listing 2. The modified program is transformed into the following constrained TRS:

$$\mathcal{R}'_{\text{sum}} = \left\{ \begin{array}{ll} \text{sum}(x) \rightarrow u_1(x, 0, 0), & \\ u_1(x, i, z) \rightarrow u_2(x, i, z) \llbracket x > 0 \rrbracket, & u_2(x, i, z) \rightarrow u_2(x, s(i), z + s(i)) \llbracket i \neq x \wedge x > i \rrbracket, \\ u_1(x, i, z) \rightarrow z \llbracket \neg x > 0 \rrbracket, & u_2(x, i, z) \rightarrow z \llbracket \neg(i \neq x \wedge x > i) \rrbracket \end{array} \right\}$$

In this case, $\mathcal{R}'_{\text{sum}}$ is terminating, and thus, termination of the program `sum` in Listing 2 is preserved by the transformation. Note that the termination was proved by the implementation of the technique proposed by Sakata et al [14]. Moreover, the termination prover AProVE 1.2 [9] succeeded in proving termination of the following *integer term rewriting system* [7] which corresponds to $\mathcal{R}'_{\text{sum}}$:

$$\left\{ \begin{array}{ll} \text{sum}(x) \rightarrow u_1(x, 0, 0, x > 0), & \\ u_1(x, i, z, \text{true}) \rightarrow u_2(x, i, z, x \neq i \wedge x > i), & u_1(x, i, z, \text{false}) \rightarrow z, \\ u_2(x, i, z, \text{true}) \rightarrow u_2(x, i + 1, z + i + 1, x \neq i + 1 \wedge x > i + 1), & u_2(x, i, z, \text{false}) \rightarrow z \end{array} \right\}$$

■ **Listing 2** A variant of the program `sum` where the loop invariant is introduced

```
int sum(int x){
  int i=0, z=0;
  if(x > 0)
    while(i != x && x > i){
      i = i+1;
      z = z+i;
    }
  return z;
}
```

We summarize the idea explained above as follows:

1. we generate a loop invariant for each “while” statement in the program;
 2. we append the loop invariant to the corresponding loop condition;
 3. we apply the transformation to the modified program, getting a constrained TRS.
- Note that not for all terminating programs, the transformed rewrite system is terminating.

5 Conclusion

In this paper, by using an example, we showed the idea for improving preservation of termination in applying transformations from simple procedural programs over integers into constrained TRSs. However, this idea does not always work as we expected. For example, `true` is a trivial loop invariant which does not improve anything. For this reason, the improvement relies on the power of methods to generate loop invariants.

One may think that we do not have to append a loop invariant to the negation of the corresponding loop condition, which is coupled with rewrite rules controlling the computation flow of the “while” statements. For example, for the computation of `sum`, we do not need to append `x > i` to $u_2(x, i, z) \rightarrow z \llbracket \neg i \neq x \rrbracket$, i.e., $u_2(x, i, z) \rightarrow z \llbracket \neg i \neq x \rrbracket$ is enough. However, if `x > i` is not appended to $\neg i \neq x$, then the rewrite system is not *sufficiently complete w.r.t. interpretable terms*. This can be said of the case that we replace the rule by $u_2(x, i, z) \rightarrow z \llbracket (\neg i \neq x) \wedge x > i \rrbracket$ — since `x > i` is a loop invariant, the invariant holds when leaving out of the “while” loop, and thus, to make the reduction more precise, we may append `x > i` to $\neg i \neq x$. In many methods based on rewriting induction, however, sufficient completeness of a rewrite system is required, and thus, the transformed rewrite system is expected to be sufficient complete. For this reason, we do not employ the more strict rule $u_2(x, i, z) \rightarrow z \llbracket (\neg i \neq x) \wedge x > i \rrbracket$.

As future work, we will make an experiment in order to evaluate how successful the improvement is, and will further improve the preservation of termination.

Acknowledgement We thank the anonymous referees for their useful comments to improve the paper.

References

- 1 F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
- 2 A. Bouhoula and F. Jacquemard. Automated induction with constrained tree automata. In *Proceeding of the 4th International Joint Conference on Automated Reasoning*, volume 5195 of *Lecture Notes in Computer Science*, pp. 539–554, Springer, 2008.

- 3 M. Colón, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In *Proceedings of the 15th International Conference on Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pp. 420–432, Springer, 2003.
- 4 L. M. de Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pp. 337–340, Springer, 2008.
- 5 S. Falke and D. Kapur. A term rewriting approach to the automated termination analysis of imperative programs. In *Proceedings of the 22nd International Conference on Automated Deduction*, volume 5663 of *Lecture Notes in Computer Science*, pp. 277–293, Springer, 2009.
- 6 S. Falke and D. Kapur. Rewriting induction + linear arithmetic = decision procedure. In *Proceedings of the 6th International Joint Conference on Automated Reasoning*, volume 7364 of *Lecture Notes in Computer Science*, pp. 241–255, Springer, 2012.
- 7 C. Fuhs, J. Giesl, M. Plücker, P. Schneider-Kamp, and S. Falke. Proving termination of integer term rewriting. In *Proceedings of the 20th International Conference on Rewriting Techniques and Applications*, volume 5595 of *Lecture Notes in Computer Science*, pp. 32–47, Springer, 2009.
- 8 Y. Furuichi, N. Nishida, M. Sakai, K. Kusakari, and T. Sakabe. Approach to procedural-program verification based on implicit induction of constrained term rewriting systems. *IPSJ Transactions of Programming*, 1(2):100–121, 2008 (in Japanese).
- 9 J. Giesl, P. Schneider-Kamp, and R. Thiemann. Aprove 1.2: Automatic termination proofs in the dependency pair framework. In *Proceedings of the 3rd International Joint Conference on Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*, pp. 281–286, Springer, 2006.
- 10 T. Kataoka, N. Nishida, M. Sakai, T. Sakabe, and K. Kusakari. Use of loop invariants for improving termination preservability of transformations from procedural programs to rewriting systems. In *Record of 2013 Tokai-Section Joint Conference on Electrical and Related Engineering*, no. M2-6, 1 page, 2013 (in Japanese).
- 11 C. Kop and N. Nishida. Term rewriting with logical constraints. In *Proceedings of the 9th International Symposium on Frontiers of Combining Systems*, volume 8152 of *Lecture Notes in Computer Science*, pp. 343–358, Springer, 2013.
- 12 C. Otto, M. Brockschmidt, C. von Essen, and J. Giesl. Automated termination analysis of java bytecode by term rewriting. In *Proceedings of the 21st International Conference on Rewriting Techniques and Applications*, volume 6 of *LIPIcs*, pp. 259–276, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2010.
- 13 U. S. Reddy. Term rewriting induction. In *Proceedings of the 10th International Conference on Automated Deduction*, volume 449 of *Lecture Notes in Computer Science*, pp. 162–177, Springer, 1990.
- 14 T. Sakata, N. Nishida, and T. Sakabe. On proving termination of constrained term rewrite systems by eliminating edges from dependency graphs. In *Proceedings of the 20th International Workshop on Functional and Constraint Logic Programming*, volume 6816 of *Lecture Notes in Computer Science*, pp. 138–155, Springer, 2011.
- 15 T. Sakata, N. Nishida, T. Sakabe, M. Sakai, and K. Kusakari. Rewriting induction for constrained term rewriting systems. *IPSJ Transactions of Programming*, 2(2):88–96, 2009 (in Japanese).
- 16 G. Vidal. Closed symbolic execution for verifying program termination. In *Proceedings of the 12th IEEE International Working Conference on Source Code Analysis and Manipulation*, pp. 34–43, IEEE Computer Society, 2012.