An inversion tool for conditional term rewriting systems - a case study of Ackermann inversion

Mikkelsen, Maria Bendix; Glück, Robert; Kirkeby, Maja Hanne

Published in:
Proceedings of the 9th International Workshop on Verification and Program Transformation

DOI:
10.4204/EPTCS.341.3

Publication date:
2021

Document version
Publisher's PDF, also known as Version of record

Document license:
CC BY-NC

Citation for published version (APA):
We report on an inversion tool for a class of oriented conditional constructor term rewriting systems. Four well-behaved rule inverters ranging from trivial to full, partial and semi-inverters are included. Conditional term rewriting systems are theoretically well founded and can model functional and non-functional rewrite relations. We illustrate the inversion by experiments with full and partial inversions of the Ackermann function. The case study demonstrates, among others, that polyvariant inversion and input-output set propagation can reduce the search space of the generated inverse systems.

Keywords program inversion, program transformation, term rewriting systems, case study

1 Introduction

Program inversion is one of the fundamental transformations that can be performed on programs [3]. Although function inversion is an important concept in mathematics, program inversion has received little attention in computer science. In this paper, we report on a tool implementation of an inversion framework [6] and on some computer experiments within the framework. The implementation includes four well-behaved rule inverters ranging from trivial to full, partial and semi-inverters, several of which have been studied in the literature [7, 12, 13]. The generic inversion algorithm used by the tool was proven to produce the correct result for all well-behaved rule inverters [6]. The tool reads the standard notation of the established confluence competition (COCO), making it compatible with other term rewriting tools. The Haskell implementation is designed as an open system for experimental and educational purposes that can be extended with further well-behaved rule inverters.

In particular, we illustrate the use of the tool by repeating A.Y. Romanenko’s three experiments with full and partial inversions of the Ackermann function [15, 16]. His inversion algorithm, inspired by Turchin [17], inverts programs written in a Refal extension, Refal-R [16], which is a functional-logic language, whereas our tool uses a subclass of oriented conditional constructor term rewriting systems (CCSs) [1, 14]. Conditional term rewriting systems are theoretically well founded and can model a wide range of language paradigms, e.g., reversible, functional, and declarative languages.

Let us illustrate our tool with three kinds of inversions of a simple remove-index function, \texttt{rem} (Figure 1). Given a list and a unary number \texttt{n}, it returns the \texttt{n}th element of the list and the list without the removed element. \texttt{rem} is defined by two rewrite rules: the first rule defines the base case where \texttt{cons} (\texttt{::}) is written in prefix notation and the two outputs are tupled (\texttt{<\cdot>}). The second rule contains a so-called condition after the separator (\texttt{<=}) that can be read as a recursion. The input-output relation specified by \texttt{rem} is exemplified by the list \([a, b, b]\) and the indices 0, 1, and 2 (inputs are marked in blue).
An Inversion Tool for Conditional Term Rewriting Systems – A Case Study of Ackermann Inversion

I sets (output. In our tool, the new directionality of the desired program is specified by input and output index O and an output index set rem{}{1,2} positions are the output. The name random-insert identifies the insertion of an element into a list at an arbitrary position. The updated lists and the corresponding generic inversion algorithm and the rule inverters, interested readers are therefore referred to [6].

The contribution of the work reported here is a complete implementation of the generic inversion algorithm, together with four well-behaved rule inverters. The system is available for experimental (source code) and educational purposes (web system). We then report on the case study of Ackermann inversion repeating three experiments by A.Y. Romanenko and compare the results and provide measures of the rewrite steps and function calls. We are not aware of other investigations of Romanenko’s experiments.

We begin by giving an overview of the tool (Section 2) followed by the case study of Ackermann inversion (Section 3). This paper, the tool implementation itself, and the paper on the inversion framework are intended to complement each other. They are intended to be used together. For more details on the generic inversion algorithm and the rule inverters, interested readers are therefore referred to [6].

Figure 1: Full, partial and semi-inversion of the remove-index function rem.

 Usually, we consider program inversion as full inversion that swaps a program’s entire input and output. In our tool, the new directionality of the desired program is specified by input and output index sets (io-sets). The user can select the input and output arguments that become the input arguments of the inverse program, so this technique is very general. Full inversion always has an empty input index set I and an output index set O containing all indices of the outputs.

Full inversion of rem yields a program rem{}{1,2} whose non-functional input-output relation specifies the insertion of an element into a list at an arbitrary position. The updated lists and the corresponding positions are the output. The name rem{}{1,2} indicates that none of rem’s inputs (\{\}) and all of rem’s outputs (\{1,2\}) are the new inputs.

The full inverse of a non-injective function specifies a non-functional relation. Thus, program inversion does not respect language paradigms, and this is one of the inherent difficulties when performing program inversion for a functional language. The inverted rules do not always define a functional relation because they may have overlapping left-hand sides or extra variables. The non-functional relation random-insert of rem{}{1,2} is induced by two overlapping rules.

Partial inversion swaps parts of the input and the entire output. rem{}{1,2} is a partial inverse of rem where the original list is swapped with the entire output. It defines the insertion of an element at a position n in a list, i.e., the functional relation insert. Semi-inverse, the most general form of inversion, can swap any part of the input and output. rem{}{1} is a semi-inverse of rem where the position and the element are swapped, i.e., the non-functional relation remove-elem. While we obtain two programs for the price of one by full inversion, we can obtain several programs by partial and semi-inversion.
2 Inversion Tool

The tool is implemented in Haskell\(^2\) and we provide both an online web-based version\(^3\) and the source code\(^4\). We demonstrate how to partially invert a function add, which defines the addition of two unary numbers, to obtain \(\text{add}\{1\}\{1\}\), which defines the subtraction of two unary numbers. As illustrated in Figure\(^2\) the Inversion Framework requires three inputs: (i) the original CCS rules with the add-rules, (ii) the Inversion task: add with io-set \(I = \{1\}\) and \(O = \{1\}\), and (iii) an indication of which well-behaved Rule inverter the tool should apply, here, the partial rule inverter. The inversion framework provides two outputs: (i) the Inverted CCS rules, containing the \(\text{add}\{1\}\{1\}\)-rules defining subtraction, and (ii) a Diagnostics table with an overview of the systems’ paradigm characteristics. Because the program inversion does not respect language paradigms, it is useful that the tool also provides an analysis of the programs’ paradigm characteristics; see \([6, \text{Fig} 2]\) for definitions and interrelations.

Whereas the source code provides a command line interface, which facilitates composition with other program transformations, the online web-based version provides a friendly clickable interaction; see Figure\(^3\) for a screenshot. In the following, we describe the most important content and features of the online tool. The tool web-site contains the following:

1. a navigation bar (in the top) with green action buttons and white settings buttons,
2. a white input window with a text field for the original CCS,
3. a gray output window (in the lower left corner) for the inverted systems, and
4. another gray output window (in the lower right corner) with program diagnostics.

The original CCS can either be entered into the input window or chosen from the predefined CCS examples available via the Examples button, e.g., choosing add. Using the Options button, one defines the inversion task, e.g., the partial inversion of add with \(I = \{1\}\) and \(O = \{1\}\) and selects one of the rule inverters, e.g., the partial rule inverter. To apply the inverter, we use the Invert button whereafter the tool

\(^2\)The Glorious Glasgow Haskell Compilation System, version 8.10.4
\(^3\)https://topps.di.ku.dk/pirc/inversion-tool
\(^4\)https://github.com/pirc-src/inversion-tool
Figure 3: The interface of the web-based inversion tool after partially inverting `add` to `add{1}{1}`.

creates (or updates) the gray output windows with the inversion, e.g., the partial inversion `add{1}{1}`, and the paradigm characteristics of both the original and the inverted program, e.g., column `ORIG` and column `PART`. For instance, we can see that both `add` and `add{1}{1}` are functional and that none of them is reversible. The `Diagnose` button provides a more detailed property analysis of the program in the white input text field. Another feature is the `Latex` button that translates the CCS in the main window into LATEX code that can be used when typesetting documents.

3 A Case Study of Ackermann Inversion

We illustrate the use of our tool by repeating three experiments [16], namely, two partial inversions and a full inversion of the Ackermann function `ack` (Figure 4a). `ack` takes two unary numbers as inputs and returns one unary number as output.

First Experiment An io-set together with the `ack` program in Figure 4a are the tool inputs. The io-set for this experiment is `I = {1}` and `O = {1}`, specifying that the first input term and the output term of `ack` are the inputs for the partially inverted program `ack{1}{1}`. Then, our tool propagates the io-set through the entire program and transforms the rules locally using the selected well-behaved rule inverter.

The result of the pure partial inversion [6, Fig.6] is shown in Figure 4b. The resulting program
ack(0, y) \rightarrow <s(y)>
ack(s(x), 0) \rightarrow <2> \leftarrow \text{ack}(x, s(0)) \rightarrow <z>
ack(s(x), s(y)) \rightarrow <z> \leftarrow \text{ack}(s(x), y) \rightarrow <v>, \text{ack}(x, v) \rightarrow <z>

(a) Program ack implementing the Ackermann function $\text{Ack}(x, y)$.

\begin{align*}
\text{ack}(1\{1\}\{0, s(y)\}) & \rightarrow <y> \\
\text{ack}(1\{1\}\{s(x), z\}) & \rightarrow <0> \leftarrow \text{ack}(1, 2\{1\}\{x, s(0), z\}) \rightarrow <> \\
\text{ack}(1\{1\}\{s(x), z\}) & \rightarrow <s(y)> \leftarrow \text{ack}(1\{1\}\{x, z\}) \rightarrow <v>, \\
& \quad \text{ack}(1\{1\}\{s(x), v\}) \rightarrow <y> \\
\text{ack}(1, 2\{1\}\{0, y, s(y)\}) & \rightarrow <> \\
\text{ack}(1, 2\{1\}\{s(x), 0, z\}) & \rightarrow <> \leftarrow \text{ack}(1\{1\}\{x, s(0), z\}) \rightarrow <> \\
\text{ack}(1, 2\{1\}\{s(x), s(y), z\}) & \rightarrow <> \leftarrow \text{ack}(1\{1\}\{x, z\}) \rightarrow <v>, \\
& \quad \text{ack}(1, 2\{1\}\{s(x), y, v\}) \rightarrow <> \\
\end{align*}

(b) Partial inverse of ack with $I = \{1\}$ and $O = \{1\}$.

\begin{align*}
\text{ack}_2(0, s(y)) & \rightarrow <y> \\
\text{ack}_2(s(x), z) & \rightarrow <0> \leftarrow \text{ack}_2(x, z) \rightarrow <s(0)> \\
\text{ack}_2(s(x), z) & \rightarrow <s(y)> \leftarrow \text{ack}_2(x, z) \rightarrow <v>, \\
& \quad \text{ack}_2(s(x), v) \rightarrow <y> \\
\end{align*}

(c) Romanenko’s partial inverse $\text{Ack}_2^{-1}$ [16, p.17] rewritten as a CCS.

Figure 4: Partial inversions of the Ackermann function and the dependency graphs.

consists of two defined function symbols, namely, the desired partial inverse $\text{ack}\{1\}\{1\}$, which depends on another partial inverse $\text{ack}\{1, 2\}\{1\}$. The io-set of $\text{ack}\{1, 2\}\{1\}$ specifies that it takes both inputs and the output of $\text{ack}$ as the input. As a consequence, all of its three rules return a nullary output tuple $<>$. In the case of Romanenko’s $\text{ack}_2$ in Figure 4c, the second rule’s right-hand side of the condition is a constant $s(0)$. Since this output is a known constant, we can provide it as input to the left-hand side using our tool. This illustrates that our tool fully propagates the io-sets such that all known terms become the new input. This means that the algorithm is a polyvariant inverter in that it may produce several inversions of the same function symbol, namely, one for each input-output index set.

The relation specified by the partial inverse is functional [16, p.18], but the program in Figure 4b is nondeterministic due to a single pair of overlapping rules, i.e., the 2nd and 3rd rules of $\text{ack}\{1\}\{1\}$. The same issue occurs for the partially inverted program $\text{Ack}_2^{-1}$ ($\text{ack}_2$, Figure 4c) [16, p.17]. Comparison of the two programs shows that in $\text{ack}\{1\}\{1\}$’s second rule, our tool has moved the condition’s constant $s(0)$ to the input side and thereby created a dependency on the more specific partial inversion $\text{ack}\{1, 2\}\{1\}$ instead of $\text{ack}\{1\}\{1\}$.

The effect is to reduce the search space when rewriting using the inverted systems. In this experiment, we found a remarkable reduction of function calls and rewrite steps. The results and the speed-ups for $\text{ack}_2$ and $\text{ack}\{1\}\{1\}$ are reported in Table 1. For the tested inputs ranging from $(1, 2)$ to $(3, 509)$ the speed-up is up to 5.89 for rewrite steps and up to 3.92 for function calls when comparing $\text{ack}\{1\}\{1\}$ with Romanenko’s $\text{ack}_2$. We observe that there is a rewrite step speed-up for all inputs, while the two programs have equally many function calls when the first input is 1. One reason is that $\text{ack}\{1\}\{1\}$ tends
to use fewer rewriting steps because its call to \( \text{ack}\{1, 2\}\{1\} \) can fail using pattern matching, whereas \( \text{ack}_2 \) requires a rewriting and pattern match of the result to establish the same failure.

The number of required rewrite steps is used in the complexity of conditional term rewriting systems [8], and the number of function/predicate calls is used in the complexity analysis of functional and logic programs [9, 11]. Here, function calls correspond to the number of function-rooted terms that must be rewritten to reach normal form.

To confirm that these speed-ups manifest themselves in a functional-logic language, we implemented \( \text{ack}\{1\}\{1\} \) and \( \text{ack}_2 \) in Curry and measured their runtimes in CPU seconds on input \((3, 253)\) using two Curry systems\(^5\). The Haskell-based Kics2 terminated on the programs after 1295.7 s and 4674.6 s, respectively, and the Prolog-based Pakcs3 terminated after 8.5 s and 40.7 s, respectively. Thus, the speed-ups in Curry, which are 3.61 and 4.79, are comparable to the speed-ups for function calls and rewrite steps.

On the other hand, the polyvariant io-set propagation also has a cost with respect to the size of \( \text{ack}\{1\}\{1\} \): in the worst case, all possible inversions of a function symbol are created–io-sets are never

---

\(^5\)The programs were executed using the Docker images caups/pakcs3:3.3.0 and caups/kics2:2.3.0 on an Apple MacBook Pro (2.6 GHz 6-Core Intel Core i7 processor, 16 GB memory, Intel Graphics). The execution times are slower than if Curry were installed directly on the machine, but the relative program execution times are expected to hold in either case.
ack{2}{1}(y, s(y)) -> <0>
ack{2}{1}(0, z) -> <s(x)> <= ack{2}{1}(s(0), z) -> <x>
ack{2}{1}(s(y), z) -> <s(x)> <= ack{1}{1}(z) -> <x, v>,
ack{1,2}{1}(s(x), y, v) -> < >
ack{}{1}(s(y)) -> <0, y>
ack{}{1}(z) -> <s(x), 0> <= ack{}{1}(z) -> <x>
ack{}{1}(z) -> <s(x), s(y)> <= ack{}{1}(z) -> <x, v>,
ack{1}{1}(s(x), v) -> <y>

(a) Partial inverse of ack with \( I = \{2\} \) and \( O = \{1\} \) includes, in addition, the rules of Figure 4b.

ack_1(y, s(y)) -> <0>
ack_1(0, z) -> <s(x)> <= ack_1(s(0), z) -> <x>
ack_1(s(y), z) -> <s(x)> <= ack_0(z) -> <s(x), v>,
ack_1(s(y), v) -> <s(x)>
ack_0(s(y)) -> <0, y>
ack_0(z) -> <s(x), 0> <= ack_0(z) -> <x, s(0)>
ack_0(z) -> <s(x), s(y)> <= ack_0(z) -> <x, v>,
ack_0(v) -> <s(x), y>

(b) Romanenko’s partial inverse \( Ack^{-1}_{1} [16, p.17] \) rewritten as a CCS.

![Dependency graph](image)

(c) Dependency graphs of ack{2}{1} and ack_1.

Figure 5: A partial inversion of the Ackermann function and the dependency graphs.

generalized—thereby increasing the size of the generated program. Despite the full propagation of the io-sets, the tool always terminates due to their finite number for any program; this characteristic relates to mode analysis [14]. Romanenko’s method, which is potentially more powerful due to the global approach because it builds a configuration graph and uses generalization to make the unfolding of calls terminate, produces a *monovariant* partial inverse \( Ack_2 \) so that not all known local information is used (Figure 4c); this may be due to the generalization in the configuration graph [15].

**Second experiment** The next experiment is the partial inversion \( Ack_2 \) and our tool correctly produces the inverse that defines four function symbols including \( Ack_1 \) and \( Ack_2 \) and also a full inverse \( Ack_0 \). This full inverse depends on the partial inverses \( Ack_0 \) and \( Ack_2 \) due to the io-set propagation in our tool. By contrast, Romanenko’s partial inversion \( Ack_1 \) depends on itself and on \( Ack_1 \)’s full inverse \( Ack_0 \). His full inverse depends on itself [16, p.17] instead of partial inverses that would have been possible if all known information was exploited. Both systems \( Ack_2 \) and \( Ack_1 \) are shown in Figure 5a and 5b where \( Ack_2 \) depends on \( Ack_1 \) and \( Ack_0 \) in Figure 4b. The systems are illustrated by their dependency graphs in Figure 5c.

Romanenko’s \( Ack_1 \) and \( Ack_2 \) are nonterminating. The third rule of \( Ack_0 \) has \( Ack_0(z) \)
An Inversion Tool for Conditional Term Rewriting Systems – A Case Study of Ackermann Inversion

\begin{align*}
  \text{ack}\{1\}(s(y)) & \rightarrow <0, y> \\
  \text{ack}\{1\}(z) & \rightarrow <s(x), 0> \leftarrow \text{ack}\{1\}(z) \rightarrow <x, s(0)> \\
  \text{ack}\{1\}(z) & \rightarrow <s(x), s(y)> \leftarrow \text{ack}\{1\}(z) \rightarrow <x, v>, \\
  \text{ack}\{1\}(v) & \rightarrow <s(x), y> \\
\end{align*}

Figure 6: The full inverse of the Ackermann function.

as its left-hand side and also requires a rewriting of the same term \(\text{ack}_0(z)\) in its first condition, thus yielding an infinitely deep search tree. The third rule of \(\text{ack}\{1\}\) has a similar structure. Since both programs are nonterminating, no counts are provided. Nevertheless, when producing \(\text{ack}\{2\}\{1\}\), our tool discovers an improvement of the inverse system, e.g., the second condition of the third rule depends on the terminating \(\text{ack}\{1,2\}\{1\}\) whereas the same condition of the same rule of \(\text{ack}_1\) depends on the nonterminating \(\text{ack}_1\).

The cost of creating polyvariant inversions is evident in \(\text{ack}\{2\}\{1\}\), where the tool has created 4 different inversions of the 3 original rules, producing a system of 12 rules. In comparison, \(\text{ack}_1\) consists of two inversions of the same three original rules producing a smaller system of 6 rules; see the dependency graphs in Figure 5c.

Third experiment In the third experiment, Romanenko used his full inverter [16, Sect.3.1] to invert \(\text{ack}\) and our pure full inverter [6, Fig.6] produces exactly the same program, namely, \(\text{ack}\{1\}\), in Figure 6. Please note that this full inversion shares the same defined function symbol as the rules in Figure 5, but the rules are different. This is because they define the same input-output relation, namely, the full inversion of the original \(\text{ack}\). This system is nonterminating; thus, no count is provided. By exploiting the mathematical property of Ackermann that its output is larger than its input, it may be possible to create a terminating full inversion. It is beyond the tool to use extra mathematical properties to improve the inversions.

The fourth partial inversion that is possible is \(\text{ack}\{1,2\}\{1\}\), which is already included in Figure 4b. This means that with our tool, we produced all four possible partial inversions (including the special case of full inversion) of the Ackermann function in the course of the three experiments. Using our tool, we also reproduced all of the examples in [6,7].

4 Conclusion and Future Work

The goal of this work was to provide a design space for the experimental evaluation and comparison of different well-behaved rule inverters, including those using heuristic approaches [7]. It will be interesting to investigate Romanenko’s inversion method [15] as well as related global approaches [2,4,5] and program analyses such as mode and binding-time analyses. Using CCSs enabled us to focus on the essence of inversion without considering language-specific details, as demonstrated by the examples above. The examples demonstrate that polyvariant inversion can considerably reduce the search space of the inverted system. The post-optimizations of the inverted programs represent another future direction of investigation. We have observed two potential improvements: the first is the reduction of nondeterminism by determinization [5,10], and the other is exploiting constants by partial evaluation, for example, the constant \(s(0)\) of the 2nd rule of Figure 4b. We expect that this will further improve the efficiency of inverse systems. In future work, one can consider the translation of the resulting programs to logic or functional-logic programming languages, such as Prolog or Curry, and explore the relation to partial deduction in logic programming.
Acknowledgements  Thanks to Alberto Pettorossi and to the anonymous reviewers for their constructive feedback on an earlier version of this paper.

References


