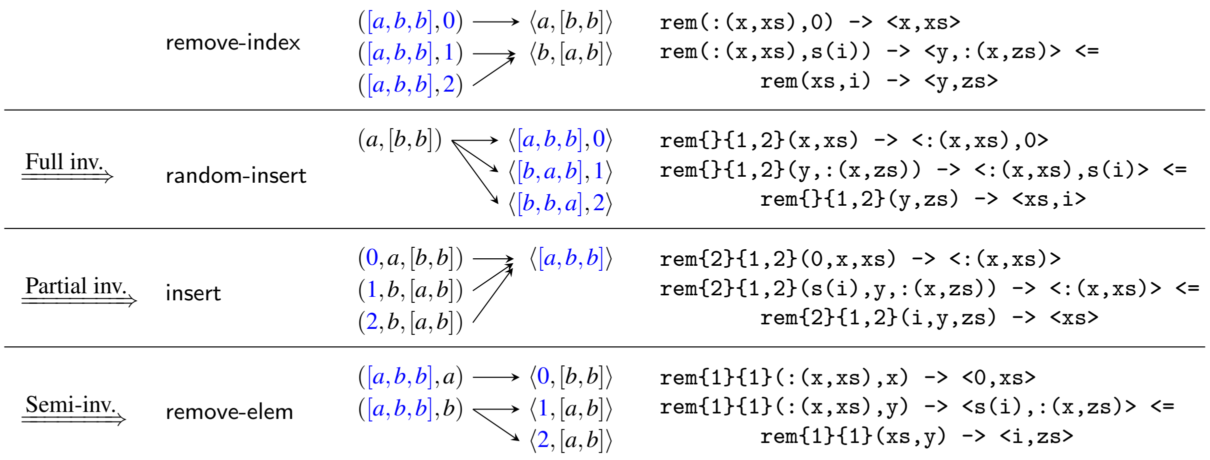
An Inversion Tool for Conditional Term Rewriting Systems

Maria Bendix Mikkelsen[*] Robert Glück
DIKU, University of Copenhagen, Denmark
mbm@di.ku.dk, glueck@acm.org

Maja H. Kirkeby
Roskilde University, Denmark
kirkebym@acm.org

In this extended abstract, we report on a tool implementation[1] of the local inversion framework [5] and on some experiments. The implementation includes four well-behaved rule-inverters ranging from trivial to full, partial and semi-inverters, several of which were studied in the literature [6, 8, 9]. The generic inversion algorithm used by the tool was proven to produce the correct result for all well-behaved rule-inverters [5]. 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 [11, 12]. His inversion algorithm, inspired by Turchin [13], inverts programs written in a Refal extension, Refal-R [12], which is a functional-logic language, whereas our tool uses a subclass of oriented conditional constructor term rewriting systems[2] (CCSs) [1, 10]. Conditional term rewriting systems are theoretically well founded and can model a wide range of language paradigms, e.g., reversible, functional, and declarative languages.


remove-index
$([a,b,b],0) \longrightarrow \langle a,[b,b]\rangle$
$([a,b,b],1) \longrightarrow \langle b,[a,b]\rangle$
$([a,b,b],2) \longrightarrow \langle b,[a,b]\rangle$
$\xrightarrow{\text{Full inv.}}$
random-insert
$(a,[b,b]) \longrightarrow \langle [a,b,b],0\rangle, \langle [b,a,b],1\rangle, \langle [b,b,a],2\rangle$
$\xrightarrow{\text{Partial inv.}}$
insert
$(0,a,[b,b]) \longrightarrow \langle [a,b,b]\rangle$
$(1,b,[a,b]) \longrightarrow \langle [a,b,b]\rangle$
$(2,b,[a,b]) \longrightarrow \langle [a,b,b]\rangle$
$\xrightarrow{\text{Semi-inv.}}$
remove-elem
$([a,b,b],a) \longrightarrow \langle 0,[b,b]\rangle$
$([a,b,b],b) \longrightarrow \langle 1,[a,b]\rangle, \langle 2,[a,b]\rangle$

```
rem(:(x,xs),0) -> <x,xs>
rem(:(x,xs),s(i)) -> <y,:(x,zs)> <=
        rem(xs,i) -> <y,zs>
```

```
rem{}{1,2}(x,xs) -> <:(x,xs),0>
rem{}{1,2}(y,:(x,zs)) -> <:(x,xs),s(i)> <=
        rem{}{1,2}(y,zs) -> <xs,i>
```

```
rem{2}{1,2}(0,x,xs) -> <:(x,xs)>
rem{2}{1,2}(s(i),y,:(x,zs)) -> <:(x,xs)> <=
        rem{2}{1,2}(i,y,zs) -> <xs>
```

```
rem{1}{1}(:(x,xs),x) -> <0,xs>
rem{1}{1}(:(x,xs),y) -> <s(i),:(x,zs)> <=
        rem{1}{1}(xs,y) -> <i,zs>
```

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

Let us illustrate our tool with three kinds of inversions of a simple remove-index function, `rem` (Figure 1). Given a list and a unary number n, it returns the nth element of the list and the list without the removed element. `rem` is defined by two rewrite rules: The first rule defines the base case where cons (`:`) is written in prefix notation and the two outputs are tupled (`<·>`). The second rule contains a so-called

[*]Thanks for support to Zonta International, Club of Copenhagen.
[1]https://github.com/mariabendix1/inversion-tool
[2]CCSs are also referred to as pure constructor CTRS [7].

```
ack(0,y)       -> <s(y)>
ack(s(x),0)    -> <z> <= ack(x,s(0)) -> <z>
ack(s(x),s(y)) -> <z> <= ack(s(x),y) -> <v>, ack(x,v) -> <z>
```

(a) Program ack implementing the Ackermann function $Ack(x,y)$.

```
ack{1}{1}(0,s(y)) -> <y>
ack{1}{1}(s(x),z) -> <0>    <= ack{1,2}{1}(x,s(0),z) -> <>
ack{1}{1}(s(x),z) -> <s(y)> <= ack{1}{1}(x,z)      -> <v>,
                               ack{1}{1}(s(x),v) -> <y>
ack{1,2}{1}(0,y,s(y))     -> <>
ack{1,2}{1}(s(x),0,z)     -> <> <= ack{1,2}{1}(x,s(0),z) -> <>
ack{1,2}{1}(s(x),s(y),z) -> <> <= ack{1}{1}(x,z) -> <v>,
                                  ack{1,2}{1}(s(x),y,v) -> <>
```

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

```
ack_2(0,s(y)) -> <y>
ack_2(s(x),z) -> <0>    <= ack_2(x,z) -> <s(0)>
ack_2(s(x),z) -> <s(y)> <= ack_2(x,z) -> <v>,
                           ack_2(s(x),v) -> <y>
```

(c) Romanenko's partial inverse Ack_2^{-1} [12, p.17] rewritten as a CCS.

	(a)	(b)	(c)
Number of functions	1	2	1
Number of rules	3	6	3
Functional	[x]	[ ]	[ ]
- Orthogonal	[x]	[ ]	[ ]
. Non-overlapping	[x]	[ ]	[ ]
. Left-linear	[x]	[ ]	[x]
- EV-Free	[x]	[x]	[x]
- Left-to-right determ.	[x]	[x]	[x]
Reversible	[ ]	[ ]	[ ]
- Functional	[x]	[ ]	[ ]
- Output-orthogonal	[ ]	[ ]	[ ]
. Non-output-overlap.	[ ]	[ ]	[ ]
. Right-linear	[x]	[x]	[x]
- Strictly non-erasing	[x]	[ ]	[x]
. Non-erasing	[x]	[ ]	[x]
. Weakly non-erasing	[x]	[x]	[x]

(d) Paradigm properties of programs (a-c).

Figure 2: A partial inversion of the Ackermann function and the paradigm properties of the CCSs.

condition after the separator (<=) that can be read as a recursion. The input-output relation specified by `rem` is exemplified by the list $[a,b,b]$ and the indices 0, 1, and 2 (inputs are marked in blue).

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 that 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}` that inserts an element into a list at a random position and returns the new list and the element's position. 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 function because of the rules with overlapping left-hand sides or extra variables, so that the inverse becomes a non-functional relation. 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{2}{1,2}` is a partial inverse of `rem` where the original list was 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-inversion*, the most general form of inversion, can swap any part of the input and output. `rem{1}{1}` is a semi-inversion 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.

Ackermann We illustrate the use of our tool by repeating three experiments [12], namely two partial inversions and a full inversion of the Ackermann function `ack` (Figure 2a). `ack` takes two unary numbers as inputs and returns one unary number as output. An io-set together with the `ack` program in Figure 2a are the input for our tool. The io-set for the first experiment is $I = \{1\}$ and $O = \{1\}$, specifying that the first input-term and the output-term of `ack` are the input 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 [5, Fig.6] is shown in Figure 2b. It is observed that the resulting program consists of two defined function symbols, namely, the desired partial inverse `ack{1}{1}` that depends on another partial inverse `ack{1,2}{1}`. The io-set of `ack{1,2}{1}` specifies that it takes both inputs and the output of `ack` as the input. As a consequence, all of its three rules return a nullary output tuple `<>`, implying that it is a semi-predicate. This illustrates that the 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 [12, p.18], but the program in Figure 2b is nondeterministic due to a single pair of overlapping rules, i.e. the 2nd and 3rd rule of `ack{1}{1}`. The same issue occurs for the partially inverted program Ack_2^{-1} (`ack_2`, Figure 2c) [12, p.17]. Comparison of the two programs shows that in `ack{1}{1}`'s second rule, our tool has moved the constant `s(0)` and thereby created a dependency on the more specific partial inversion `ack{1,2}{1}`. This kind of precision of the io-set propagation has a cost: in the worst case, all possible inversions of a function symbol are created–io-sets are never 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 [10]. 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 2c); this may be due to the generalization of the call graph [11].

Generally, program inversion does not respect language paradigms and therefore it is important that the tool also provides an analysis of the programs' paradigm characteristics [5, Fig.2]. The characteristics of the original and the two inverted programs are shown in Figure 2d. The first column shows that the original program `ack` is a functional program, i.e., the rules defining a function symbol are orthogonal and without unbound variables[3], and the conditions of the rules can be evaluated sequentially from left to right, such as nested let-expressions in a functional program (left-to-right determinism). In the second column, we can see that `ack{1}{1}` is not a functional implementation because it is not orthogonal. We have added the characteristics of `ack_2` in a third column, and here we see that it is not a functional implementation either because of the existence of overlapping rules. The tool also states the additional properties required for reversible programs and it is observed that none of the three programs satisfies these properties. For details and definitions of the characteristics in Figure 2d see [5].

The next experiment is the partial inversion `ack{2}{1}` and our tool correctly produces the inverse that defines four function symbols including `ack{1}{1}`, `ack{1,2}{1}` and, in addition, a full inverse `ack{}{1}`. This full inverse depends on the partial inverses `ack{1}{1}` and `ack{2}{1}` due to the precision of the io-set propagation in our tool. By contrast, Romanenko's partial inversion Ack_1^{-1} depends on itself and on *Ack*'s full inverse Ack^{-1}. This full inverse depends only on itself [12, p.17] instead of partial inverses that would have been possible if all known information was exploited.

In the third experiment, Romanenko used his full inverter [12, Sect.3.1] to invert `ack`, and our pure full inverter [5, Fig.6] produces exactly the same program. Using our tool, we also reproduce all of the examples in [5, 6] (included with the tool implementation).

[3]That is, without extra variables, in short EV-free rules as defined in [9].

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 [6]. It will be interesting to investigate Romanenko's inversion method [11] as well as related global approaches [2, 3, 4] 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 post-optimizations of the inverted programs represent another future line of investigation. We have observed two potential improvements: the first is the reduction of nondeterminism by determinization [4, 7], and the other is exploiting constants by partial evaluation, for example, the constant `s(0)` of the 2nd rule of Figure 2b. 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 a logic or functional-logic programming language, such as Prolog or Curry.

References

[1] Marc Bezem, Jan W. Klop & Roel de Vrijer (2003): *Term rewriting systems: Terese*. Cambridge University Press, United Kingdom.

[2] Robert Glück & Masahiko Kawabe (2005): *Revisiting an automatic program inverter for Lisp*. *SIGPLAN Notices* 40(5), pp. 8–17, doi:10.1145/1071221.1071222.

[3] Robert Glück & Valentin F. Turchin (1990): *Application of metasystem transition to function inversion and transformation*. In: *International Symposium on Symbolic and Algebraic Computation. Proceedings*, ACM, pp. 286–287, doi:10.1145/96877.96953.

[4] Masahiko Kawabe & Robert Glück (2005): *The program inverter LRinv and its structure*. In Manuel Hermenegildo & Daniel Cabeza, editors: *Practical Aspects of Declarative Languages. Proceedings*, LNCS 3350, Springer, pp. 219–234, doi:10.1007/978-3-540-30557-6_17.

[5] Maja H. Kirkeby & Robert Glück (2020): *Inversion framework: reasoning about inversion by conditional term rewriting systems*. In: *Principles and Practice of Declarative Programming. Proceedings*, ACM, p. Article 9, doi:10.1145/3414080.3414089.

[6] Maja H. Kirkeby & Robert Glück (2020): *Semi-inversion of conditional constructor term rewriting systems*. In Maurizio Gabbrielli, editor: *Logic-based Program Synthesis and Transformation. Proceedings*, LNCS 12042, Springer, pp. 243–259, doi:10.1007/978-3-030-45260-5_15.

[7] Masanori Nagashima, Masahiko Sakai & Toshiki Sakabe (2012): *Determinization of conditional term rewriting systems*. *TCS* 464, doi:10.1016/j.tcs.2012.09.005.

[8] Naoki Nishida (2004): *Transformational Approach to Inverse Computation in Term Rewriting*. Ph.D. thesis, Graduate School of Engineering, Nagoya University, Japan.

[9] Naoki Nishida, Masahiko Sakai & Toshiki Sakabe (2005): *Partial inversion of constructor term rewriting systems*. In Jürgen Giesl, editor: *Rewriting Techniques and Applications. Proceedings*, LNCS 3467, Springer, pp. 264–278, doi:10.1007/978-3-540-32033-3_20.

[10] Enno Ohlebusch (2002): *Advanced Topics in Term Rewriting*. Springer, New York.

[11] Alexander Y. Romanenko (1988): *The generation of inverse functions in Refal*. In Dines Bjørner, Andrei P. Ershov & Neil D. Jones, editors: *Partial Evaluation and Mixed Computation*, North-Holland, pp. 427–444.

[12] Alexander Y. Romanenko (1991): *Inversion and metacomputation*. In: *Partial Evaluation and Semantics-Based Program Manipulation. Proceedings*, ACM, pp. 12–22, doi:10.1145/115865.115868.

[13] Valentin F. Turchin (1986): *The concept of a supercompiler*. *ACM TOPLAS* 8(3), pp. 292–325, doi:10.1145/5956.5957.