Journal of Logical and Algebraic Methods in Programming 94 (2018) 128–149

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

Reversible computation in term rewriting ☆

Naoki Nishida [a], Adrián Palacios [b], Germán Vidal [b,*]

[a] Graduate School of Informatics, Nagoya University, Furo-cho, Chikusa-ku, 4648603 Nagoya, Japan
[b] MiST, DSIC, Universitat Politècnica de València, Camino de Vera, s/n, 46022 Valencia, Spain

ARTICLE INFO

Article history:
Received 7 December 2016
Received in revised form 12 June 2017
Accepted 8 October 2017
Available online 31 October 2017

Keywords:
Term rewriting
Reversible computation
Program transformation

ABSTRACT

Essentially, in a reversible programming language, for each forward computation from state S to state S', there exists a constructive method to go backwards from state S' to state S. Besides its theoretical interest, reversible computation is a fundamental concept which is relevant in many different areas like cellular automata, bidirectional program transformation, or quantum computing, to name a few.

In this work, we focus on term rewriting, a computation model that underlies most rule-based programming languages. In general, term rewriting is not reversible, even for injective functions; namely, given a rewrite step $t_1 \to t_2$, we do not always have a decidable method to get t_1 from t_2. Here, we introduce a conservative extension of term rewriting that becomes reversible. Furthermore, we also define two transformations, injectivization and inversion, to make a rewrite system reversible using standard term rewriting. We illustrate the usefulness of our transformations in the context of bidirectional program transformation.

1. Introduction

The notion of reversible computation can be traced back to Landauer's pioneering work [22]. Although Landauer was mainly concerned with the energy consumption of erasing data in irreversible computing, he also claimed that every computer can be made reversible by saving the *history* of the computation. However, as Landauer himself pointed out, this would only postpone the problem of erasing the tape of a reversible Turing machine before it could be reused. Bennett [6] improved the original proposal so that the computation now ends with a tape that only contains the output of a computation and the initial source, thus deleting all remaining "garbage" data, though it performs twice the usual computation steps. More recently, Bennett's result is extended in [9] to nondeterministic Turing machines, where it is also proved that transforming an irreversible Turing machine into a reversible one can be done with a quadratic loss of space. We refer the interested reader to, e.g., [7,14,40] for a high level account of the principles of reversible computation.

☆ This work has been partially supported by the EU (FEDER) and the Spanish *Ministerio de Economía y Competitividad* (MINECO) under grants TIN2013-44742-C4-1-R and TIN2016-76843-C4-1-R, by the *Generalitat Valenciana* under grant PROMETEO-II/2015/013 (SmartLogic), and by the COST Action IC1405 on Reversible Computation – extending horizons of computing. Adrián Palacios was partially supported by the EU (FEDER) and the Spanish *Ayudas para contratos predoctorales para la formación de doctores* and *Ayudas a la movilidad predoctoral para la realización de estancias breves en centros de I+D*, MINECO (SEIDI), under FPI grants BES-2014-069749 and EEBB-I-16-11469. Part of this research was done while the second and third authors were visiting Nagoya University; they gratefully acknowledge their hospitality.

* Corresponding author.
E-mail addresses: apalacios@dsic.upv.es (A. Palacios), gvidal@dsic.upv.es (G. Vidal).

https://doi.org/10.1016/j.jlamp.2017.10.003
2352-2208/

In the last decades, reversible computing and *reversibilization* (transforming an irreversible computation device into a reversible one) have been the subject of intense research, giving rise to successful applications in many different fields, e.g., cellular automata [28], where reversibility is an essential property, bidirectional program transformation [24], where reversibility helps to automate the generation of inverse functions (see Section 6), reversible debugging [17], where one can go both forward and backward when seeking the cause of an error, parallel discrete event simulation [34], where reversible computation is used to undo the effects of speculative computations made on a wrong assumption, quantum computing [39], where all computations should be reversible, and so forth. The interested reader can find detailed surveys in the *state of the art* reports of the different working groups of COST Action IC1405 on Reversible Computation [20].

In this work, we introduce reversibility in the context of *term rewriting* [4,36], a computation model that underlies most rule-based programming languages. In contrast to other, more *ad-hoc* approaches, we consider that term rewriting is an excellent framework to rigorously define reversible computation in a functional context and formally prove its main properties. We expect our work to be useful in different (sequential) contexts, like reversible debugging, parallel discrete event simulation or bidirectional program transformation, to name a few. In particular, Section 6 presents a first approach to formalize bidirectional program transformation in our setting.

To be more precise, we present a general and intuitive notion of *reversible* term rewriting by defining a Landauer embedding. Given a rewrite system $\mathcal{R}$ and its associated (standard) rewrite relation $\to_{\mathcal{R}}$, we define a reversible extension of rewriting with two components: a forward relation $\rightharpoonup_{\mathcal{R}}$ and a backward relation $\leftharpoondown_{\mathcal{R}}$, such that $\rightharpoonup_{\mathcal{R}}$ is a conservative extension of $\to_{\mathcal{R}}$ and, moreover, $(\rightharpoonup_{\mathcal{R}})^{-1}{=}\leftharpoondown_{\mathcal{R}}$. We note that the inverse rewrite relation, $(\to_{\mathcal{R}})^{-1}$, is not an appropriate basis for "reversible" rewriting since we aim at defining a technique to *undo* a particular reduction. In other words, given a rewriting reduction $s \to^*_{\mathcal{R}} t$, our reversible relation aims at computing the term s from t and $\mathcal{R}$ in a decidable and deterministic way, which is not possible using $(\to_{\mathcal{R}})^{-1}$ since it is generally non-deterministic, non-confluent, and non-terminating, even for systems defining injective functions (see Example 6). In contrast, our backward relation $\leftharpoondown_{\mathcal{R}}$ is deterministic (thus confluent) and terminating. Moreover, our relation proceeds backwards step by step, i.e., the number of reduction steps in $s \rightharpoonup^*_{\mathcal{R}} t$ and $t \leftharpoondown^*_{\mathcal{R}} s$ are the same.

In order to introduce a reversibilization transformation for rewrite systems, we use a *flattening* transformation so that the reduction at top positions of terms suffices to get a normal form in the transformed systems. For instance, given the following rewrite system:

$$\begin{aligned} \mathsf{add}(0, y) &\to y, \\ \mathsf{add}(\mathsf{s}(x), y) &\to \mathsf{s}(\mathsf{add}(x, y)) \end{aligned}$$

defining the addition on natural numbers built from constructors 0 and s( ), we produce the following *flattened* (conditional) system:

$$\begin{aligned} \mathcal{R} = \{ \quad & \mathsf{add}(0, y) \to y, \\ & \mathsf{add}(\mathsf{s}(x), y) \to \mathsf{s}(z) \Leftarrow \mathsf{add}(x, y) \twoheadrightarrow z \ \} \end{aligned}$$

(see Example 29 for more details). This allows us to provide an improved notion of reversible rewriting in which some information (namely, the positions where reduction takes place) is not required anymore. This opens the door to *compile* the reversible extension of rewriting into the system rules. Loosely speaking, given a system $\mathcal{R}$, we produce new systems $\mathcal{R}_f$ and $\mathcal{R}_b$ such that *standard* rewriting in $\mathcal{R}_f$, i.e., $\to_{\mathcal{R}_f}$, coincides with the forward reversible extension $\rightharpoonup_{\mathcal{R}}$ in the original system, and analogously $\to_{\mathcal{R}_b}$ is equivalent to $\leftharpoondown_{\mathcal{R}}$. E.g., for the system $\mathcal{R}$ above, we would produce

$$\begin{aligned} \mathcal{R}_f = \{ \quad & \mathsf{add}^{\mathrm{i}}(0, y) \to \langle y, \beta_1 \rangle, \\ & \mathsf{add}^{\mathrm{i}}(\mathsf{s}(x), y) \to \langle \mathsf{s}(z), \beta_2(w) \rangle \Leftarrow \mathsf{add}^{\mathrm{i}}(x, y) \twoheadrightarrow \langle z, w \rangle \ \} \end{aligned}$$

$$\begin{aligned} \mathcal{R}_b = \{ \quad & \mathsf{add}^{-1}(y, \beta_1) \to \langle 0, y \rangle, \\ & \mathsf{add}^{-1}(\mathsf{s}(z), \beta_2(w)) \to \langle \mathsf{s}(x), y \rangle \Leftarrow \mathsf{add}^{-1}(z, w) \to \langle x, y \rangle \quad \} \end{aligned}$$

where $\mathsf{add}^{\mathrm{i}}$ is an injective version of function add, add^{-1} is the inverse of $\mathsf{add}^{\mathrm{i}}$, and β_1, β_2 are fresh symbols introduced to label the rules of $\mathcal{R}$.

In this work, we will mostly consider *conditional* rewrite systems, not only to have a more general notion of reversible rewriting, but also to define a reversibilization technique for unconditional rewrite systems, since the application of *flattening* (cf. Section 4) may introduce conditions in a system that is originally unconditional, as illustrated above.

This paper is an extended version of [31]. In contrast to [31], our current paper includes the proofs of technical results, the reversible extension of term rewriting is introduced first in the unconditional case (which is simpler and more intuitive), and presents an improved injectivization transformation when the system includes injective functions. Furthermore, a prototype implementation of the reversibilization technique is publicly available from http://kaz.dsic.upv.es/rev-rewriting.html.

The paper is organized as follows. After introducing some preliminaries in Section 2, we present our approach to reversible term rewriting in Section 3. Section 4 introduces the class of *pure constructor* systems where all reductions take place at topmost positions, so that storing this information in reversible rewrite steps becomes unnecessary. Then, Section 5 presents injectivization and inversion transformations in order to make a rewrite system reversible with standard rewriting. Here, we also present an improvement of the transformation for injective functions. The usefulness of these transformations