Journal of Logical and Algebraic Methods in Programming 84 (2015) 341–358

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

Combining relation algebra and data refinement to develop rectangle-based functional programs for reflexive–transitive closures

Rudolf Berghammer [a,*], Sebastian Fischer

[a] Institut für Informatik, Christian-Albrechts-Universität zu Kiel, Olshausenstraße 40, 24098 Kiel, Germany

ARTICLE INFO

Article history:
Received 23 May 2013
Received in revised form 18 March 2014
Accepted 6 August 2014
Available online 6 September 2014

Keywords:
Directed graphs
Relation algebra
Reflexive-transitive closure
Rectangle
Functional programming
Haskell

ABSTRACT

We show how to systematically derive simple purely functional algorithms for computing the reflexive–transitive closure of directed graphs. Directed graphs can be represented as binary relations and we develop our algorithms based on a relation-algebraic description of reflexive–transitive closures. This description employs the notion of rectangles and instantiating the resulting algorithm with different kinds of rectangles leads to different algorithms for computing reflexive–transitive closures. Using data refinement, we then develop simple Haskell programs for two specific choices of rectangles and show that one of them has cubic running time like Warshall's standard algorithm. Finally, we apply our approach to other standard operations of relation algebra and present graph theoretic applications of our developments.

1. Introduction

The reflexive–transitive closure of a directed graph G is a directed graph with the same vertices as G that contains an edge from each vertex x to each vertex y if and only if y is reachable from x in G. With "directed graph" we mean in this paper a graph with directed edges but without parallel edges between two vertices; sometimes this kind of graphs is also called 1-graph. Therefore, we can represent any directed graph G as (binary) relation R on the vertex set of G that relates two vertices x and y if and only if there is an edge from x to y. The reflexive–transitive closure of a directed graph G then corresponds to the reflexive–transitive closure R^* of the binary relation R which represents G, i.e., to the least reflexive and transitive relation that contains R.

Usually, the task of computing R^* is solved by a variant of Warshall's algorithm which represents R as a Boolean matrix, where each entry that is 1 represents an edge of the graph. It first computes the transitive closure R^+ of R using Warshall's original method (presented in [43]) and then obtains R^* from R^+ by putting all entries of the main diagonal of R^+ to 1. Representing the directed graph as a 2-dimensional Boolean array leads to a simple and efficient algorithm with three nested loops for the computation of R^+ and a single loop for the subsequent treatment of the main diagonal, i.e., to a program of running time complexity in $O(n^3)$ with n as cardinality of the carrier set of R.

* Corresponding author. Tel.: +49 431 7272; fax: +49 431 7613.
E-mail address: rub@informatik.uni-kiel.de (R. Berghammer).

http://dx.doi.org/10.1016/j.jlamp.2014.08.003
2352-2208/

```
for i = 1 to n do
  for j = 1 to n do
    for k = 1 to n do
      R[j,k] := R[j,k] ∨ (R[j,i] ∧ R[i,k])
for i = 1 to n do
  R[i,i] := 1
```

However, in certain cases arrays are unfit for representing binary relations. In particular this holds if both R and R^* are of "medium density" or even sparse. Such relations appear, for instance, in computational linguistics when computing the so-called subtype relation in HPSG-type signatures (see, for example, [30,31]). They also appear in XML-query processing since XML structures are nothing else than (directed) trees with few additional reference links (see, e.g., [17]). Also in the context of ordered sets sparse relations appear if a partial order relation is computed from the cover relation (the Hasse diagram) and a lot of elements are non-comparable. In these applications a representation of relations by, say, successor lists, lists of pairs or even look-up tables (i.e., characteristic functions) is much more economic. But such a representation sacrifices the simplicity and efficiency of the above implementation. Moreover, the traditional method of imperatively updating an array representing the directed graph is alien to the purely functional programming paradigm which restricts the use of side effects.

There are some sub-cubic algorithms for the computation of reflexive–transitive closures of relations. Most of these algorithms are based upon sub-cubic algorithms for matrix multiplication, that is, on Strassen's well-known method (see [39]) and its refinements. There is also a refinement of Warshall's algorithm (more precisely, of Floyd's extension for the all-pairs shortest paths problem given in [22]) that computes the transitive closure of a relation in a running time that is in $O(n^{2.5})$; see [23]. But all these sub-cubic algorithms pay for their exponents by an intricacy that, particularly with regard to practice, makes it difficult to implement them correctly (w.r.t. the input/output behaviour as well as the theoretical running time bound) in a conventional programming language like C, Java or Haskell. The complexity of the sub-cubic algorithms also leads in the O-estimation to such large constant coefficients that, again concerning practice, results are computed faster normally only in the case of very dense relations or of very large carrier sets. But both cases hardly appear in practical applications.

The present paper is an extended version of [11]. To get on with the results of [10] and [11], in it we present an alternative method for computing reflexive–transitive closures. Using relation algebra (in the sense of [34,37,40,41]) as the methodical tool, we first develop a generic algorithm for computing R^*, that is based on the decomposition of the relation R by so-called rectangles, and then provide some instantiations. Each concrete algorithm – we now speak of a program – is determined by a specific choice of a rectangle that is removed from R. Due to its generality, our generic algorithm not only allows simple and efficient imperative array-based implementations, but also simple and efficient purely functional ones, which base upon a very simple representation of relations via successor lists. In this paper we focus on functional programming and use Haskell as programming language.

The remainder of the paper is organized as follows. Section 2 is devoted to the relation-algebraic preliminaries. In Section 3, we first prove a property about reflexive–transitive closures and rectangles that is the base of the generic algorithm and then present the algorithm itself and its instantiations in the second part of the section in a functional style. Section 4 shows how the generic algorithm can be implemented in the functional programming language Haskell. First, we present a direct translation of the generic algorithm over relations into Haskell-code using a Haskell type class for relation algebra. The advantage of this approach is its simplicity, elegance and generality; its disadvantage is that we do not get a cubic running time as desired. Therefore, after the direct translation, we develop more efficient Haskell solutions using data refinement. Especially, we will develop a simple Haskell program that uses a representation of relations via linear lists of successor lists and computes reflexive–transitive closures in cubic running time, i.e., has the same time complexity than the aforementioned traditional imperative method. We think that this is a valuable single result. But we also believe that the overall method we will apply in the same way as in [10] is the more valuable contribution of the paper. In Section 5 we will demonstrate our method with two further operations on relations, viz. composition and transposition, and present graph theoretic applications of the developed functions. Section 6 reviews related work and the final section contains concluding remarks.

2. Relation-algebraic preliminaries

Following the Z specification language (see [38], for example), we denote the set (frequently also called: type) of all relations with source X and target Y, i.e., the powerset $2^{X\times Y}$ of the set $X \times Y$, by $[X \leftrightarrow Y]$ and write $R : X \leftrightarrow Y$ instead of $R \in [X \leftrightarrow Y]$. If the sets X and Y are finite, we may consider a relation $R : X \leftrightarrow Y$ as a Boolean matrix with $|X|$ rows and $|Y|$ columns. Since this specific interpretation of relations is well suited for many purposes, we will often use matrix notation and terminology in this paper. In particular, we talk about rows, columns and entries of relations.

We assume the reader to be familiar with the basic operations on relations, viz. R^{T} (transposition), $\overline{R}$ (complement), $R \cup S$ (join), $R \cap S$ (meet), $R;S$ (composition), the predicate indicating $R \subseteq S$ (inclusion) and the special relations O (empty relation), L (universal relation) and I (identity relation). Furthermore, we assume the reader to know the most fundamental laws of relation algebra like