Executable Transitive Closures

René Thiemann

August 28, 2014

Abstract

We provide a generic work-list algorithm to compute the (reflexive-)transitive closure of relations where only successors of newly detected states are generated. In contrast to our previous work [2], the relations do not have to be finite, but each element must only have finitely many (indirect) successors. Moreover, a subsumption relation can be used instead of pure equality. An executable variant of the algorithm is available where the generic operations are instantiated with list operations.

This formalization was performed as part of the IsaFoR/CeTA project\footnote{http://cl-informatik.uibk.ac.at/software/ceta}, and it has been used to certify size-change termination proofs where large transitive closures have to be computed.

Contents

1 A work-list algorithm for reflexive-transitive closures 1
   1.1 The generic case .......................... 2
   1.2 Instantiation using list operations ....... 4

1 A work-list algorithm for reflexive-transitive closures

theory RTrancl
imports ../Regular-Sets/Regexp-Method
begin

In previous work [2] we described a generic work-list algorithm to compute reflexive-transitive closures for \textit{finite} relations: given a finite relation \(r\), it computed \(r^*\).

In the following, we develop a similar, though different work-list algorithm for reflexive-transitive closures, it computes \(r^*''\ \text{init}\) for a given relation \(r\) and finite set \textit{init}. The main differences are that

\footnote{Supported by FWF (Austrian Science Fund) project P22767-N13.}
• The relation \( r \) does not have to be finite, only \( \{ b : (a, b) \in r^* \} \) has to be finite for each \( a \). Moreover, it is no longer required that \( r \) is given explicitly as a list of pairs. Instead \( r \) must be provided in the form of a function which computes for each element the set of one-step successors.

• One can use a subsumption relation to indicate which elements to no longer have to be explored.

These new features have been essential to certify size-change termination proofs [1] where the transitive closure of all size-change graphs has to be computed. Here, the relation is size-change graph composition.

• Given an initial set of size-change graphs with \( n \) arguments, there are roughly \( N := 3^{n^2} \) many potential size-change graphs that have to be considered as left-hand sides of the composition relation. Since the composition relation is even larger than \( N \), an explicit representation of the composition relation would have been too expensive. However, using the new algorithm the number of generated graphs is usually far below the theoretical upper bound.

• Subsumption was useful to generate even fewer elements.

1.1 The generic case

Let \( r \) be some finite relation.

We present a standard work-list algorithm to compute all elements that are reachable from some initial set. The algorithm is generic in the sense that the underlying data structure can freely be chosen, you just have to provide certain operations like union, selection of an element.

In contrast to [2], the algorithm does not demand that \( r \) is finite and that \( r \) is explicitly provided (e.g., as a list of pairs). Instead, it suffices that for every element, only finitely many elements can be reached via \( r \), and \( r \) can be provided as a function which computes for every element \( a \) all one-step successors w.r.t. \( r \). Hence, \( r \) can in particular be any well-founded and finitely branching relation.

The algorithm can further be parametrized by a subsumption relation which allows for early pruning.

In the following locales, \( r \) is a relation of type \( 'a 
\Rightarrow 'a \), the successors of an element are represented by some collection type \( 'b \) which size can be measured using the \textit{size} function. The selection function \( \text{sel} \) is used to meant to split a non-empty collection into one element and a remaining collection. The union on \( 'b \) is given by \textit{un}.

\textbf{locale} \textit{subsumption} =
```plaintext
fixes r :: 'a ⇒ 'b
and subsumes :: 'a ⇒ 'a ⇒ bool
and set-of :: 'b ⇒ 'a set
assumes
  subsumes-refl: ∀ a. subsumes a a
  and subsumes-trans: ∀ a b c. subsumes a b → subsumes b c → subsumes a c
  and subsumes-step: ∀ a b c. subsumes a b → c ∈ set-of (r b) → ∃ d ∈ set-of (r a). subsumes d c
begin
abbreviation R where R ≡ { (a, b) | b ∈ set-of (r a) }
end
locale subsumption-impl = subsumption r subsumes set-of
for r :: 'a ⇒ 'b
and subsumes :: 'a ⇒ 'a ⇒ bool
and set-of :: 'b ⇒ 'a set +
fixes
  sel :: 'b ⇒ 'a × 'b
  and un :: 'b ⇒ 'b ⇒ 'b
  and size :: 'b ⇒ nat
assumes set-of-fin: ∀ b. finite (set-of b)
  and sel: ∀ a b c. set-of b ≠ {} → sel b = (a, c) = set-of b = insert a (set-of c) ∧ size b > size c
  and un: set-of (un a b) = set-of a ∪ set-of b
locale relation-subsumption-impl = subsumption-impl r subsumes set-of sel un size
for r subsumes set-of sel un size +
assumes rtrancl-fin: ∀ a. finite { (a, b) | (a, b) ∈ { (a, b) | b ∈ set-of (r a) } } *=
begin
lemma finite-Rs: assumes init: finite init
  shows finite (R *" init)
  ⟨proof⟩
  a standard work-list algorithm with subsumption
function mk-rtrancl-main where
mk-rtrancl-main todo fin = (if set-of todo = {} then fin
  else let (a, tod) = sel todo
    in (if (∃ b ∈ fin. subsumes b a) then mk-rtrancl-main tod fin
      else mk-rtrancl-main (un (r a) tod) (insert a fin)))
  ⟨proof⟩
termination mk-rtrancl-main
  ⟨proof⟩
declare mk-rtrancl-main.simps[simp del]
```

---

```
```
lemma mk-rtrancl-main-sound: set-of todo ∪ fin ⊆ R⁺⁺ " init --- mk-rtrancl-main
todo fin ⊆ R⁺⁺ " init
⟨proof⟩

lemma mk-rtrancl-main-complete:
[ Λ a. a ∈ init ⇒ ∃ b. b ∈ set-of todo ∪ fin ∧ subsumes b a ]
⇒ [ Λ a. a ∈ fin ⇒ b ∈ set-of (r a) ⇒ ∃ c. c ∈ set-of todo ∪ fin ∧ subsumes c b ]
⇒ c ∈ R⁺⁺ " init
⇒ ∃ b. b ∈ mk-rtrancl-main todo fin ∧ subsumes b c
⟨proof⟩

definition mk-rtrancl where mk-rtrancl init ≡ mk-rtrancl-main init {}

lemma mk-rtrancl-sound: mk-rtrancl init ⊆ R⁺⁺ " set-of init
⟨proof⟩

lemma mk-rtrancl-complete: assumes a: a ∈ R⁺⁺ " set-of init
shows ∃ b. b ∈ mk-rtrancl init ∧ subsumes b a
⟨proof⟩

lemma mk-rtrancl-no-subsumption: assumes subsumes = (op =)
shows mk-rtrancl init = R⁺⁺ " set-of init
⟨proof⟩
end

1.2 Instantiation using list operations

It follows an implementation based on lists. Here, the working list algorithm
is implemented outside the locale so that it can be used for code generation.
In general, it is not terminating, therefore we use partial-function instead of
function.

partial-function (tailrec) mk-rtrancl-list-main where
[code]: mk-rtrancl-list-main subsumes r todo fin = (case todo of [] ⇒ fin
| Cons a tod ⇒
    (if (∃ b ∈ set fin. subsumes b a) then mk-rtrancl-list-main subsumes r
tod fin
    else mk-rtrancl-list-main subsumes r (r a @ tod) (a ≠ fin))))

definition mk-rtrancl-list where
mk-rtrancl-list subsumes r init ≡ mk-rtrancl-list-main subsumes r init []

locale subsumption-list = subsumption r subsumes set
for r :: 'a ⇒ 'a list and subsumes :: 'a ⇒ 'a ⇒ bool

locale relation-subsumption-list = subsumption-list r subsumes for r subsumes +
assumes rtrancl-fin: Λ a. finite { (a,b) ∈ { (a,b) . b ∈ set (r a) } "*}

abbreviation (input) sel-list where sel-list x ≡ case x of Cons h t ⇒ (h,t)
context relation-subsumption-list
begin
  The main equivalence proof between the generic work list algorithm and
  the one operating on lists

lemma mk-rtrancl-list-main: fin = set finl \implies set (mk-rtrancl-list-main subsumes r todo finl) = mk-rtrancl-main todo finl
  ⟨proof⟩

lemma mk-rtrancl-list: set (mk-rtrancl-list subsumes r init) = mk-rtrancl init
  ⟨proof⟩
end

References

