Verification of Dijkstra algorithm in Coq

(Weryfikacja algorytmu Dijkstry w Coqu)

Bartłomiej Królikowski

Praca magisterska

Promotor: dr Małgorzata Biernacka

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

28 sierpnia 2024

Abstract

. . .

. . .

Contents

Chapter 1

Introduction

...

Chapter 2

System overview

We created a system for verification of programs in separation logic, based on [Charguéraud, 2024] and [Charguéraud and Pottier, 2015].

2.1 Model language

The system is built upon a formalization of lambda calculus with references, based on the ones in [Polesiuk, 2023], where the syntax of expressions is parametrized by the set of free variables for better handling of substitution. We use this language to formalize the algorithm we verify.

2.1.1 Syntax

The language features:

- a unit value - used as a dummy return value in state manipulating operations
- integer and boolean values
- integer operations (negation, addition, subtraction, multiplication and division) and comparisons
- boolean operations (negation, conjunction, alternative)
- labels - memory addresses, allowing for state-mutating operations
- records - tuples with arbitrarily large list of fields
- lambda expressions - where bound variables are represented as de Bruijn indices [de Bruijn, 1972] which range is limited by the number of increments of the set parametrizing the grammar (see [Polesiuk, 2023])
- memory allocations - single cell and array allocations

- memory reading and writing
- memory deallocation
- shift label operation - shifting a label by a nonnegative integer to access array cells
- control flow statements - sequencing, if and while

Just like in [Polesiuk, 2023] we make a syntactic distinction between irreducible values and other expressions. The set of values consists of:

- the unit value
- variables from the set the grammar is parametrized by
- integer and boolean values
- labels - memory addresses, allowing for state-mutating operations
- record values - tuples with arbitrarily large list of fields
- lambdas - where bound variables are represented as de Bruijn indices [de Bruijn, 1972] which range is limited by the number of increments of the set parametrizing the grammar (see [Polesiuk, 2023])

2.1.2 Heap

We model the memory as a list of (possibly empty) cells. Thus we obtain a finite map from labels to values with the ability to easily measure the size of the used memory and to distinguish between non-allocated cells and cells allocated but not yet assigned.

2.1.3 Reduction rules

After the syntax, a reduction relation is introduced in SOS style. The relation is deterministic, binding every input expression-heap pair with at most one expression-heap output pair.

Based on one-step reduction we define a many-step reduction relation that tracks the number of required steps. We interpret this number of steps as the time cost of computation of the expression. Like its one-step equivalent it is also deterministic, where every input expression-heap pair is related to at most one output expression-heap-time pair.

2.1.4 Notations

To improve readability of Coq verification scripts we introduce notations for each language construct and we define functions allowing to write lambdas with bound variables of type `string` and computing their proper form with de Bruijn indices.

2.1.5 Properties

We define and prove for each language construct the following properties:

- closeness of expressions and values
  (through an auxiliary predicate `is_limited`)
- a big step reduction relation

Proofs of those with some auxiliary properties are located in *LamRefFacts.v*.

2.2 Separation logic

We created a formalization of separation logic [Charguéraud, 2024] to reason about programs in a syntax directed manner.

2.2.1 State assertions

The state assertions are predicates describing the properties of initial and final states. In our formalization, a state consists of two resources:

- time credits [Charguéraud and Pottier, 2015], being the upper limit of computation time
- the heap, being the partial map from labels to values

2.2.2 Triples

We express total correctness specifications of programs as separation logic triples `triple` t P Q, where t is the term (expression or value), P is a precondition (state assertion) and Q is the postcondition (a function from values to state assertions). The frame rule is baked in as described in [Charguéraud, 2024]. For each language construct we state and prove a corresponding separation logic rule as a theorem. We automated the process of proving most of these theorems. Some auxiliary tactics we defined came in handy during the actual program verification.

We also define three auxiliary types of triples:

- `triple_fun` that serves a purpose of stating properties of a unary function
- `triple_fun_n_ary` that is a generalization of `triple_fun` to n-ary functions
- `triple_list` that allows for describing a result of computing a sequence of expressions

and we prove lemmas allowing for using specifications expressed with those functions in places where the regular `triple` is needed.

The file *LamRefLogicFactsTotal_credits_perm.v* contains all proved facts about state assertions and separation logic triples.

Chapter 3

Dijkstra algorithm

We formalize in our modeling language and verify Dijkstra algorithm as it is described in [Ahuja et al., 1993]. We parametrize the implementation with the heap operating functions and specify its behaviour using auxiliary heap predicates.

Implementation:

Listing 3.1: Formalization of Dijkstra algorithm

```
Definition generic_dijkstra (get_size get_max_label
   get_neighbours mkheap h_insert h_empty h_extract_min
   h_decrease_key h_free l_is_nil l_head l_tail : Value string)
    : Value string :=
  [-\] "g", [-\] "src",
    [let "n"] get_size <* Var "g" [in]
    [let "C"] get_max_label <* Var "g" [in]
    [let "h"] mkheap <* Var "n" <* Var "C" [in]
    [let "dist"] NewArray (Var "n") [in]
    [let "pred"] NewArray (Var "n") [in]
      init_array <* (Var "dist") <* (Var "n") <* (Int (-1));;
      init_array <* (Var "pred") <* (Var "n") <* (Int (-1));;
      assign_array_at <* Var "dist" <* Var "src" <* Int 0;;
      h_insert <* Var "h" <* Var "src" <* Int 0;;
      [while] [~] (h_empty <* Var "h") [do]
        [let "rec_current"] h_extract_min <* (Var "h") [in]
        [let "current"] Get 0 (Var "rec_current") [in]
        [let "dist_current"] Get 1 (Var "rec_current") [in]
        [let "neighs"] Ref (get_neighbours <* Var "g" <* Var "
           current") [in]
        (* neighs : a reference to a list *)
          [while] [~] (l_is_nil <* Var "neighs") [do]
            [let "rec_neigh"] l_head <* Var "neighs" [in]
            [let "neigh"] Get 0 (Var "rec_neigh") [in]
            [let "length"] Get 1 (Var "rec_neigh") [in]
            [let "dist_neigh"] ! (Var "dist" >> Var "neigh") [
               in]
            [let "new_dist"] Var "dist_current" [+] Var "length
```

```
               " [in]
             [if] (Var "dist_neigh" [<] Int 0) [then]
               assign_array_at <* Var "dist" <* Var "neigh" <*
                    Var "new_dist";;
               assign_array_at <*Var "pred" <* Var "neigh" <*
                   Var "current";;
               h_insert <* Var "h" <* Var "neigh" <* Var "
                   new_dist"
             [else] [if] (Var "new_dist" [<] Var "dist_neigh")
                 [then]
               assign_array_at <* Var "dist" <* Var "neigh" <*
                    Var "new_dist";;
               assign_array_at <* Var "pred" <* Var "neigh" <*
                    Var "current";;
               h_decrease_key <* Var "h" <* Var "neigh" <* Var
                    "new_dist"
             [else]
               U_val (* Nothing happens. *)
             [end]
             [end]
           [end]
           [end]
           [end]
           [end]
           [end];;
           Var "neighs" <- l_tail <* Var "neighs"
         [end];;
         Free (Var "neighs")
       [end]
       [end]
       [end]
       [end]
     [end];;
     h_free <* (Var "h");;
     RecV [Var "dist"; Var "pred"]
     (*
     [let "x"] ! (Var "dist" >> Var "dst") [in]
       free_array <* (Var "dist");;
       Var "x"
     [end]
     *)
   [end]
   [end]
   [end]
   [end]
   [end]%string.
```

Chapter 4

Mathematical analysis

In *Graphs.v* we formalize a part of graph theory required to prove key properties of the Dijkstra algorithm, i. e. `Dijkstra_initial`, `Dijkstra_invariant` and `Dijkstra_final`, describing respectively the initial state when computing algorithm's main loop, its invariant and the final state, being also the final state of the algorithm. In this formalization we folow the proof of correctness of Dijkstra algorithm from [Ahuja et al., 1993].

Chapter 5

Verification of the code

We prove the folowing specification:

Listing 5.1: Verification of Dijkstra algorithm

```
Theorem triple_fun_generic_dijkstra
  (get_size get_max_label get_neighbours mkheap h_insert
     h_empty
    h_extract_min h_decrease_key h_free l_is_nil l_head l_tail
       : Value string) :
  is_closed_value get_size ->
  is_closed_value get_max_label ->
  is_closed_value get_neighbours ->
  is_closed_value mkheap ->
  is_closed_value h_insert ->
  is_closed_value h_empty ->
  is_closed_value h_extract_min ->
  is_closed_value h_decrease_key ->
  is_closed_value h_free ->
  is_closed_value l_is_nil ->
  is_closed_value l_head ->
  is_closed_value l_tail ->
  get_size_spec       get_size ->
  get_max_label_spec  get_max_label ->
  get_neighbours_spec get_neighbours ->
  mkheap_spec         mkheap ->
  h_insert_spec       h_insert ->
  h_empty_spec        h_empty ->
  h_extract_min_spec  h_extract_min ->
  h_decrease_key_spec h_decrease_key ->
  h_free_spec         h_free ->
  l_is_nil_spec       l_is_nil ->
  l_head_spec         l_head ->
  l_tail_spec         l_tail ->
  exists c0 cn cm, forall (g : wgraph nat) vg src n m C t,
  n >= 1 ->
  is_init_range (V g) ->
```

```
is_set_size (V g) n ->
is_set_size (uncurry (E g)) m ->
is_max_label g C ->
heap_time_bound n C t ->
triple_fun_n_ary 1
  (generic_dijkstra
    get_size get_max_label get_neighbours mkheap h_insert
       h_empty
    h_extract_min h_decrease_key h_free l_is_nil l_head
       l_tail)
  (fun v1 v2 => $ (c0 + cm*m + cn*n*t) <*>
    <[v1 = vg]> <*> <[v2 = Int (Z.of_nat src)]> <*>
    is_weighted_graph g vg <*> <[V g src]> <*> <[~ E g src
       src]>)
  (fun v1 v2 res => (<exists> c, $c) <*> <exists> lD lpred D
     pred,
    <[res = RecV [Lab lD; Lab lpred]]> <*>
    is_weighted_graph g vg <*> is_nat_function D lD <*>
    is_nat_function pred lpred <*> <[Dijkstra_final D pred
       src g]>).
```

Chapter 6

Final remarks

...

Bibliography

Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* PRENTICE HALL, Upper Saddle River, New Jersey 07458, 1993.

Arthur Charguéraud. *Separation Logic Foundations*, volume 6 of *Software Foundations.* Electronic textbook, 2024. URL `http://softwarefoundations.cis.upenn.edu`. Version 2.0.

Arthur Charguéraud and François Pottier. Machine-checked verification of the correctness and amortized complexity of an efficient union-find implementation. In *6th International Conference on Interactive Theorem Proving (ITP), Aug 2015, Nanjing, China*, 2015. URL `https://hal.science/hal-01245872/`.

Nicolaas G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972. ISSN 1385-7258. URL `https://doi.org/10.1016/1385-7258(72)90034-0`.

Piotr Polesiuk. Lecture notes on type systems, 2023. URL `https://github.com/ppolesiuk/type-systems-notes`. Last accessed 27 August 2024.