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

7 września 2024

Abstract

Dijkstra algorithm is one of the most fundamental algorithms operating on graphs, yet its most efficient variants present problems for verification of their correctness and complexity. In this thesis we create a framework allowing us to verify elaborate programs in separation logic. We use this system to verify time complexity of one of the optimized versions of Dijkstra algorithm, what requires from us to exploit the amortized time specifications of data structures it uses.

Algorytm Dijkstry jest jednym z najbardziej podstawowych algoritmów operujących na grafach, mimo to jego najefektywniejsze wersje stanowią wyzwanie dla weryfikujących ich poprawność i złożoność czasową. W niniejszej pracy tworzymy system pozwalający na weryfikację złożonych programów w logice separacyjnej. Wykorzystujemy ten system aby zweryfikować złożoność czasową jednej z optymalizacji algorytmu Dijkstry, co wymaga od nas wykorzystania specyfikacji struktur danych określających ich złożoność zamortyzowaną.

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 (see [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.

Listing 2.1: Memory model

```
Definition Map (V : Set) : Set := list (Label * option (Value V
   )).
```

Thus we obtain a finite map from labels to values with the ability to easily measure the size of the allocated memory, what allows for extending the system with verification of space complexity, and to distinguish between non-allocated cells and cells allocated but not yet assigned, what lets us verify the safety of memory read/write operations.

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.

It is worth to mention that we do not distinguish between state-mutating commands and other expressions. In result mutations of the state may happen in every expression evaluation and we had to account for that writing separation logic rules.

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 (see [Charguéraud, 2024]) to reason about programs in a syntax directed manner.

In separation logic properties of programs are described in a form of triples $\{P\}\ t\ \{Q\}$, where t is a term, P is a state assertion (see below: **??**) describing the initial state and Q is a function assigning state assertions to values, describing the relation between the final state and the return value. We call P a precondition and Q a postcondition of a triple. Mathematically, we can define the triple $\{P\}\ t\ \{Q\}$ as

$$\forall s \in \texttt{State}.P(s) \implies \exists v \in \texttt{Value}.\exists s' \in \texttt{State}.\langle t, s\rangle \twoheadrightarrow_\beta \langle v, s'\rangle \land Q(v, s')$$

In file *src/LamRefLogicFactsTotal_credits_perm.v* we give an analogous definition of our `triple` predicate in Coq. One difference is, in our formalization a state consists of two resources:

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

Another difference is that with our definition of the heap the frame rule (see [Charguéraud, 2024]) does not automatically hold, so we had to use the technique of baking it in, i.e. we define

$$\texttt{triple t P Q} \overset{\text{def}}{\iff} \forall H \in \texttt{StateAssertions}.\{P * H\}\ t\ \{Q * H\}$$

(see [Charguéraud, 2024] for explanation of the technique, see 2.2.1 for the definition of $(*)$).

2.2.1 State assertions

State assertions are predicates describing the properties of initial and final states. In file *src/LambdaAssertions_credits_perm.v* we give a definition of a state assertion in Coq and define a number of connectives.

The most important is the star $(*)$ connective, which asserts that a given state can be partitioned into two states satisfying left and right assertion respectively. In our scenario, partitioning of the state means dividing the heap into two disjoint heaps, and presenting the number of credits as a sum of two natural numbers (see Charguéraud and Pottier [2015]). Formally, the star can be defined as

$$(P * Q)(c, m) \overset{\text{def}}{\iff} \exists c_1, c_2 \in \mathbb{N}. \exists m_1, m_2 \in \texttt{Store}. P(c_1, m_1) \land Q(c_2, m_2) \land \\ c = c_1 + c_2 \land m = m_1 \uplus m_2$$

where '$\uplus$' means a disjoint union of finite maps. In the code we use the notation `P <*> Q` for this connective.

Together with the star we define the magic wand operator ($\twoheadrightarrow$, see Charguéraud [2024]), satisfying the property

$$P * (P \twoheadrightarrow Q) \rightarrow Q$$

where

$$(P \rightarrow Q) \overset{\text{def}}{\iff} \forall c, m. P(c, m) \implies Q(c, m)$$

for assertions P, Q. In the code we write `P <-*> Q`.

Other connectives we use include:

- the empty assertion ([ ]) - satisfied only by 0 credits and empty heap (Charguéraud [2024])

- the pure assertion ($[P]$) - just like the empty assertion, but requires proposition P to be true (Charguéraud [2024])
- an assertion introducing time credits ($\$c$) - just like the empty assertion, but with c credits instead of 0 (Charguéraud and Pottier [2015])

We also define

- the existential quantifier ($\exists x.P$, where P is an assertion) (Charguéraud [2024])
- a single cell assertion ($l :== v$) - satisfied by a single cell with location l and containing a value v (see the definition of $l \hookrightarrow v$ in Charguéraud and Pottier [2015])
- an empty cell assertion ($l :==$) - meaning the cell has been allocated but not yet assigned
- any cell assertion - combining the previous two (see file *src/LambdaAssertions_credits_perm.v* for details)
- an empty array assertion ($l : \backslash\, n \,\backslash$) - meaning an empty array of n contiguous cells, starting at location l, is allocated (see *src/LambdaAssertions_credits_perm.v*)

We also decided to define a general `array_content` assertion simplifying the reasoning about the content of arrays.

Listing 2.2: Definition of `array_content`

```
Inductive array_content {V} : list (option (Value V)) -> Value
   V -> StateAssertion V :=
| array_nil l : array_content nil (Lab l) 0 nil
| array_cons ov A n c m m' :
  Interweave [(OfNat n, ov)] m m' ->
  array_content A (Lab (OfNat (S n))) c m ->
  array_content (ov::A) (Lab (OfNat n)) c m'.
```

2.2.2 Logical rules

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 were also useful 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 *src/LambdaTotalTriple_credits_perm.v* contains all proved facts about state assertions and separation logic triples.

Some of the most complicated rules include

Listing 2.3: A triple for applications

```
Theorem triple_app (V : Set) (e1 e2 : Expr V)
  P1 P2 Q2 Q3 :
  triple e1
    P1
    (fun v => <exists> e1',
      <[v = (-\e1') /\
        (forall v : Value V, triple (subst_e e1' v) (Q2 v) Q3)
           ]> <*>
      P2) ->
  triple e2 P2 Q2 ->
  triple (App e1 e2) (sa_credits 1 <*> P1) Q3.
```

where we first compute the function generating expression in the initial state (function properties are described in the function expression's precondition), then, in the resulting state, we compute its argument, and finally we compute the result of a substitution,

Listing 2.4: A triple for records

```
Theorem triple_rec (V : Set) (es : list (Expr V)) (vs : list (
   Value V))
  n Ps P Q :
  n = List.length es ->
  n = List.length vs ->
  1+n = List.length Ps ->
  Some P = List.head Ps ->
  Some Q = last_error Ps ->
  (forall i e v P Q,
    Nth i es e ->
    Nth i vs v ->
    Nth i Ps P ->
    Nth (1+i) Ps Q ->
    triple e
      P
      (fun v' => <[v' = v]> <*> Q)) ->
  triple (RecE es)
    (sa_credits 1 <*> P)
    (fun v => <[v = RecV vs]> <*> Q).
```

where we the value of each field sequentially,

Listing 2.5: A triple for array declarations

```
Theorem triple_new_array_content (V : Set) (e : Expr V) P Q :
  triple e
    P
    (fun v => <exists> i, <[v = Int i]> <*> <[(i >= 0)%Z]> <*>
       Q i) ->
  triple (NewArray e)
    (sa_credits 1 <*> P)
    (fun v =>
      <exists> i, array_content (List.repeat None (Z.to_nat i))
          v <*> Q i).
```

where we first compute the expression generating the nonnegative size i of the array, and then we allocate i adjacent memory cells,

Listing 2.6: A triple for assignments

```
Theorem triple_assign (V : Set) (e1 e2 : Expr V) (v : option (
    Value V)) l P1 P2 Q2 :
  triple e1
    (<(l :?= v)> <*> P1)
    (fun v'' => <[v'' = Lab l]> <*> <(l :?= v)> <*> P2) ->
  triple e2
    (<(l :?= v)> <*> P2)
    (fun v' => <(l :?= v)> <*> Q2 v') ->
  triple (Assign e1 e2)
    (sa_credits 1 <*> <(l :?= v)> <*> P1)
    (fun v'' => <[v'' = U_val]> <*> <exists> v', <(l :== v')>
       <*> Q2 v').
```

where we first compute the memory location that must already be allocated (left expression), then we compute the value to assign and substitute the content of the cell at the location with the value,

Listing 2.7: A triple for conditionals

```
Theorem triple_if (V : Set) (e1 e2 e3 : Expr V) P1 Q1 Q2 :
  triple e1 P1 (fun v' => <exists> b, <[v' = Bool b]> <*> Q1 b)
      ->
  triple e2 (Q1 true) (fun v => Q2 v) ->
  triple e3 (Q1 false) (fun v => Q2 v) ->
  triple (If e1 e2 e3) (sa_credits 1 <*> P1) Q2.
```

where we first compute the conditional expression and then one of the branches, it is worth to notice that different branches may have different preconditions (depending on the result of the conditional expression evaluation)

Listing 2.8: A triple for loops

```
Theorem triple_while (V : Set) (e1 e2 : Expr V) P Q :
  triple e1
    P
    (fun v => <exists> b, <[v = Bool b]> <*> Q b) ->
  triple e2
    (Q true)
    (fun v => <[v = U_val]> <*> sa_credits 3 <*> P) ->
  triple (While e1 e2)
    (sa_credits 2 <*> P)
    (fun v => <[v = U_val]> <*> Q false).
```

where we have a loop invariant split into two: we first compute the conditional expression coming from the first invariant to the second, and then we compute the body, coming back to the first invariant and decreasing the number of credits (what guarantees termination).

After the rules we prove basic properties of the auxiliary triples, especially:

Listing 2.9: Basic rule for unary functions

```
Theorem triple_fun_app (V : Set) (v : Value V) e P Q1 Q2 :
  triple_fun v Q1 Q2 ->
  triple e P Q1 ->
  triple (v <* e) P Q2.
```

Listing 2.10: General rule for unary functions

```
Theorem triple_fun_app2
    (V : Set) (e1 e2 : Expr V) P1 P2 Q1 Q2 :
  triple e1 P1 (fun v => <[triple_fun v Q1 Q2]> <*> P2) ->
  triple e2 P2 Q1 ->
  triple (e1 <* e2) P1 Q2.
```

Listing 2.11: The rule for n-ary functions

```
Theorem triple_fun_n_ary_app (V : Set) (v : Value V) e es
  (P : StateAssertion V) Q1 Q2 :
  triple_fun_n_ary (List.length es) v Q1 Q2 ->
  triple_list_curry (e::es) P Q1 Q2
    (triple (n_ary_app v (e::es)) ($(List.length es) <*> P)).
```

where

Listing 2.12: The definition of `triple_list_curry`

```
Fixpoint triple_list_curry {V : Set} (es : list (Expr V)) (P :
    StateAssertion V) :
```

```
n_ary_fun_type (List.length es) (Value V) (StateAssertion V)
   ->
(Value V -> n_ary_fun_type (List.length es) (Value V) (
   StateAssertion V)) ->
((Value V -> StateAssertion V) -> Prop) ->
  Prop :=
match es with
| [] => fun Q Qlast f => P ->> $1 <*> Q -> f Qlast
| (e::es’) => fun Q Qlast f =>
  forall Qmid x, triple e P (fun v => <[v = x]> <*> Qmid x)
     ->
    triple_list_curry es’ (Qmid x) (Q x) (Qlast x) f
end%list.
```

Chapter 3

Mathematical analysis of Dijkstra algorithm

We formalize in our modeling language and verify Dijkstra algorithm as it is described in [Ahuja et al., 1993]. To make verification more modular we parametrize the code with all its auxiliary functions.

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", (*[-\] "dst",*)
    [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"] read_array_at <* 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"]
    [end]
    [end]
    [end]
    [end]
    [end]%string.
```

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].

3.1 Graphs library

We define a (directed) graph as a pair of a set of vertices `V` of type `A`, being a unary relation on `A`, and a set of edges `E`, being a binary relation on elements of `A` satisfying `V`.

Listing 3.2: Definition of directed graphs

```
Record graph A : Type := {
  V : A -> Prop;
  E : A -> A -> Prop;
  E_closed1 : forall u v, E u v -> V u;
  E_closed2 : forall u v, E u v -> V v
}.
```

We also define some basic notions, like

- the empty graph
- a complete graph - where the edge relation is full on the given vertex set
- a one-vertex graph - a graph with just one vertex and no edges
- the neighbourhood of a vertex - its all adjacent vertices
- the neighbourhood of a set of vertices - vertices not in the set, adjacent to some vertex in the set
- the edge of a set of vertices - vertices in the set, adjacent to some vertex outside the set
- graph intersection - a graph which vertex set and edge set are intersections of the vertex sets and edge sets of the two graphs respectively (we use notation: `g1 &&& g2`)
- a sum of graphs - like above, but we sum the sets instead of intersecting (with notation: `g1 ||| g2`)
- an induced subgraph - a graph limited to the given set of vertices
- a (directed) walk - a sequence of vertices, in which every two subsequent are connected
- a (directed) path - a walk without duplicates
- undirected walks and paths - walks and paths in the graph if we ignore its directedness

- a cycle - a sequence, where the first vertex is equal to the last and all the others form an undirected path
- a tree - a connected graph with no cycles
- a rooted tree - a tree with a special vertex called *root* such that any other vertex is connected via a directed walk from the root

and define abbreviations for some often used complex compositions of those notions, like `induced_subgraph_with_edge_and_vx` that is the sum of the induced subgraph on a set `P` after including a given vertex in it, and `P`’s edge

Listing 3.3: Definition of `induced_subgraph_with_edge_and_vx`

```
Definition induced_subgraph_with_edge_and_vx {A}
  (P : A -> Prop) (v : A) (g : graph A) : graph A :=
  induced_subgraph (set_sum P (single v)) g |||
     induced_subgraph_edge P g.
```

Then we define weighted graphs as equipped with a natural function on pairs of vertices and create functions operating on weights.

Listing 3.4: Definition of directed graphs

```
Record wgraph A : Type := {
  G :> graph A;
  W : A -> A -> nat
}.
```

Listing 3.5: A function computing the cost of a walk

```
Fixpoint walk_cost {A} (W : A -> A -> nat) (p : list A) : nat
   :=
  match p with
  | [] => 0
  | [x] => 0
  | u::(v::_) as p’ => W u v + walk_cost W p’
  end.
```

We also prove many technical lemmas, that we later exploit in the algorithm-specific part.

3.2 Algorithm verification

3.2.1 Definitions

We define three predicates that help us verify the outer loop of the algorithm.

`Dijkstra_predecessors_invariant`

The most important of them is `Dijkstra_invariant`. It defines a relation on five objects: `D` - the partial function of distance, `pred` - the partial predecessor function, `P` - the set of processed vertices, `s` - the source vertex of the graph, `g` - the weighted graph we process. Its main purpose is to be the invariant of the loop, but it also formalizes the idea of partitioning the graph into the set of processed vertices, with fixed labels, and the rest, which labels we are decreasing. It is defined as a conjunction of three invariants: `Dijkstra_connectedness_invariant`, `Dijkstra_distance_invariant` and `Dijkstra_predecessors_invariant`.

Listing 3.6: Definition of `Dijkstra_invariant`

```
Definition Dijkstra_invariant {A}
  (D : A -> option nat) (pred : A -> option A) (P : A -> Prop)
     (s : A) (g : wgraph A) :=
  Dijkstra_connectedness_invariant P s g /\
  Dijkstra_distance_invariant D P s g /\
  Dijkstra_predecessors_invariant pred P s g.
```

Each of these invariants is committed to a different property of the algorithm. `Dijkstra_connectedness_invariant` simply states that the source `s` is connected to every vertex in the subgraph induced by `P`, i.e. we process only the reachable vertices.

Listing 3.7: Definition of `Dijkstra_connectedness_invariant`

```
Definition Dijkstra_connectedness_invariant {A}
  (P : A -> Prop) (s : A) (g : graph A) :=
  forall g', g' = (induced_subgraph_with_edge_and_vx P s g) ->
  is_root g' s.
```

`Dijkstra_distance_invariant` states that the distance function `D` is correct in `P`, that is it returns the weight of the shortest path from the source to each vertex that is connected via vertices in `P` (i.e. the inner vertices of the path must be in `P`, but its ends don't have to).

Listing 3.8: Definition of `Dijkstra_distance_invariant`

```
Definition Dijkstra_distance_invariant {A}
  (D : A -> option nat) (P : A -> Prop) (s : A) (g : wgraph A)
     :=
  are_valid_distances D s (wg_lift (
     induced_subgraph_with_edge_and_vx P s) g).
```

Listing 3.9: Definition of `are_valid_distances`

```
Definition are_valid_distances {A} (D : A -> option nat) (s : A
   ) (g : wgraph A) :=
  forall v p, is_shortest_path g s v p -> D v = Some (walk_cost
      (W g) p).
```

`Dijkstra_predecessors_invariant` asserts two properties. The first one, `are_valid_predecessors` , states that the predecessor function `pred` is correct inside `P`, what means it returns the predecessor of a vertex in some tree of shortest paths (like above we require the path to have all internal vertices in `P`). In other words, if we create a graph where the vertices are either the source vertex `s` or a vertex for which `pred` is defined and the edges go from a vertex `pred`(x) to x then it is a subtree of `g`, rooted at `s` and its inner vertices are in `P`.

Listing 3.10: Definition of `Dijkstra_predecessors_invariant`

```
Definition Dijkstra_predecessors_invariant {A}
  (pred : A -> option A) (P : A -> Prop) (s : A) (g : wgraph A)
      :=
  forall g', g' = (wg_lift (induced_subgraph_with_edge_and_vx P
      s) g) ->
    are_valid_predecessors pred s g' /\
       are_maximal_predecessors pred s g'.
```

Listing 3.11: Definition of `are_valid_predecessors`

```
Definition are_valid_predecessors {A}
  (pred : A -> option A) (s : A) (g : wgraph A) :=
  is_shortest_paths_tree s (pred2graph s pred) g.
```

Listing 3.12: Definition of `are_maximal_predecessors`

```
Definition are_maximal_predecessors {A}
  (pred : A -> option A) (s : A) (g : graph A) :=
  forall v p, is_walk g s v p -> exists p', is_walk (pred2graph
      s pred) s v p'.
```

Listing 3.13: Definition of `pred2graph`

```
Definition pred2graph {A} (root : A) (pred : A -> option A) :
   graph A := {|
  V x := x = root \/ (exists y, Some y = pred x) \/ exists y,
     Some x = pred y;
  E u v := Some u = pred v;
  E_closed1 u v HE := or_intror (or_intror (ex_intro _ v HE));
  E_closed2 u v HE := or_intror (or_introl (ex_intro _ u HE));
|}.
```

Listing 3.14: Definition of `is_shortest_paths_tree`

```
Definition is_shortest_paths_tree {A} (s : A) (t : graph A) (g
   : wgraph A) :=
  is_rooted_tree s t /\
    forall v p, is_path t s v p -> is_shortest_path g s v p.
```

The second one, are_maximal_predecessors, states that the tree encoded by pred is maximal in P, i.e. every vertex reachable from the source in g via nodes from P can be reached along the edges chosen by the pred function, i.e. pred omits no node.

Dijkstra_initial and Dijkstra_final

Other two predicates that help us verify the loop are Dijkstra_initial and Dijkstra_final .

Listing 3.15: Definition of Dijkstra_final

```
Definition Dijkstra_final {A}
  (D : A -> option nat) (pred : A -> option A) (s : A) (g :
     wgraph A) :=
  are_valid_distances D s g /\
  are_valid_predecessors pred s g /\
  are_maximal_predecessors pred s g.
```

Listing 3.16: Definition of Dijkstra_initial

```
Definition Dijkstra_initial {A}
  (D : A -> option nat) (pred : A -> option A) (s : A) :=
  D s = Some 0 /\ (forall v, v <> s -> D v = None) /\ (forall v
     , pred v = None).
```

Dijkstra_final asserts the same thing as Dijkstra_invariant with P equal to the vertex set of g (of course P can as well be bigger). It states that D returns the shortest distance from s to each vertex, pred returns vertex predecessor on the shortest path from s and pred is defined for a vertex iff the vertex is reachable from or equal to s.

Dijkstra_initial asserts that we start the outer loop with a function D that is defined only on s and a function pred with empty domain.

Auxiliary relations

To write properties satisfied by the predicates above we define two more relations. One of them, distance_decrease binds two pairs of functions to state the effect of decreasing the distance labels of the neighbours of a vertex. In such a situation we set D to the shorter distance and pred to the vertex v that we currently process. The rest of the labels stay the same.

Listing 3.17: Definition of `distance_decrease`

```
Definition distance_decrease {A}
  (g : wgraph A) (v : A) (D D’ : A -> option nat) (pred pred’ :
      A -> option A) :=
  exists dv, D v = Some dv /\
  forall u,
    (E g v u ->
      (D u = None -> D’ u = Some (dv + W g v u) /\ pred’ u =
          Some v) /\
      (forall du, D u = Some du ->
        (dv + W g v u < du -> D’ u = Some (dv + W g v u) /\
            pred’ u = Some v) /\
        (dv + W g v u >= du -> D’ u = D u /\ pred’ u = pred u))
            ) /\
    (~ E g v u -> D’ u = D u /\ pred’ u = pred u).
```

Another relation defines the meaning of the closest neighbour using a relation `min_cost_elem` defined in *Graphs.v* asserting that its argument is the element of the given set with minimal cost measured by the given function

Listing 3.18: Definition of `closest_neighbour`

```
Definition closest_neighbour {A}
  (g : wgraph A) (P : A -> Prop) (D : A -> option nat) (v : A)
     :=
  min_cost_elem (neighbourhood g P) D v.
```

3.2.2 Properties

We prove the following properties:

Listing 3.19: Correctness of `Dijkstra_initial`

```
Theorem valid_initial A
  (D : A -> option nat) (pred : A -> option A) (s : A) (g :
     wgraph A) :
  V g s ->
  ~ E g s s ->
  Dijkstra_initial D pred s ->
  Dijkstra_invariant D pred empty s g.
```

Listing 3.20: Correctness of `Dijkstra_invariant`

```
Theorem valid_invariant A
  (D D’ : A -> option nat) (pred pred’ : A -> option A) (P : A
     -> Prop)
  (s : A) (g : wgraph A) (v : A) :
  Dijkstra_invariant D pred P s g ->
  closest_neighbour g P D v ->
  distance_decrease g v D D' pred pred' ->
  Dijkstra_invariant D' pred' (add_single P v) s g.
```

Listing 3.21: Correctness of `Dijkstra_final`

```
Theorem valid_final A
  (D : A -> option nat) (pred : A -> option A) (s : A) (g :
     wgraph A) :
  Dijkstra_invariant D pred full s g ->
  Dijkstra_final D pred s g.
```

`valid_initial` simply states that the state in which we enter the outer loop already satisfies the invariant and `valid_final` means that we leave the loop in the desired final state. `valid_invariant` states that the invariant holds after one iteration of the loop. In a single iteration we pull the closest neighbour `v` from the neighbourhood of `P`, update labels `D` and `pred` of all neighbours of `v` as specified by `distance_decrease` and extend `P` with `v`. We state that after these operations loop invariant still holds.

Chapter 4

Verification of the code

4.1 Assertions for data structures

To state the specification of the code implementing Dijkstra algorithm we needed to define state assertions for memory representation of graphs, heaps (priority queues) and lists. We defined weighted graphs as arrays of lists of neighbours and distances to them. We assume the vertices are natural numbers 0, 1, ..., $n-1$ and keep the data about the neighbourhood of i-th vertex in i-th cell of the array. To make the code as similar to the informal description from [Ahuja et al., 1993] we assume some auxiliary data, i.e. graph's size and upper bound on edge labels, is precomputed and stored together with the graph. That data can be obtained in an obvious way in $O(m)$ time, where m is the number of edges, what does not exceed the asymptotic complexity of our program, so it is safe to ignore that computation.

Listing 4.1: Representation of weighted graphs

```
Inductive is_weighted_graph {A} : wgraph nat -> Value A ->
   StateAssertion A :=
| is_weighted_graph_intro n (g : wgraph nat) C s c m :
  (forall i, V g i ->
    exists v Lv L,
      Lookup (OfNat (n + i)) m v /\
      is_list Lv v c m /\
      is_elem_weighted_unique_list (neighbours g i) (W g i) L
         /\
      Lv = nat_pairs2values L) ->
  is_max_label g C ->
  is_set_size (V g) s ->
  is_weighted_graph g
    (RecV [Lab (OfNat n); Int (Z.of_nat s); Int (Z.of_nat C)])
       c m.
```

We focused on verification of the generic scheme of Dijkstra algorithm, so we axiomatized the two other data structures.

Listing 4.2: Representation of lists

```
Parameter is_list : forall {V}, list (Value V) -> Value V ->
   StateAssertion V.
```

Listing 4.3: Representation of priority queues

```
Parameter is_heap :
  forall {V} (n C : nat) (P : nat -> Prop) (W : nat -> option
     nat)
    (* asymptotic number of credits required to extract all the
        elements *)
    (potential : nat),
    Value V -> StateAssertion V.
```

We designed our system so that we could reason about its complicated variants described in [Ahuja et al., 1993], especially the variant using Radix Heap data structure as the priority queue, so `is_heap` assertion is parameterized by graph-specific values, like the number of vertices `n` and the maximal edge label `C`, and assigns a natural number (a potential) to the current state of the queue to allow for amortised time analysis.

4.2 Specifications of auxiliary functions

Each data structure needs a set of specific functions to operate on it. We do not provide definitions for them, but for each function we define a predicate stating its specification. We later use those specifications to verify the algorithm in its most general form, with all auxiliary functions being its parameters.

4.2.1 Methods of a graph

We use three constant-time functions operating on graphs. Two of them, `get_size` and `get_max_label`, are just accessors of record fields keeping graph's size and maximum label.

Listing 4.4: Specification of `get_size`

```
Definition get_size_spec {A} (get_size : Value A) : Prop :=
  forall vg g c,
    get_size_cost c ->
    triple_fun get_size
      (fun v => $c <*> <[v = vg]> <*> is_weighted_graph g vg)
      (fun v => <exists> n,
        <[v = Int (Z.of_nat n)]> <*> <[is_set_size (V g) n]>
           <*>
        is_weighted_graph g vg).
```

Listing 4.5: Specification of `get_max_label`

```
Definition get_max_label_spec {A} (get_max_label : Value A) :
   Prop :=
  forall vg g c,
    get_max_label_cost c ->
    triple_fun get_max_label
      (fun v => $c <*> <[v = vg]> <*> is_weighted_graph g vg)
      (fun v => <exists> C,
        <[v = Int (Z.of_nat C)]> <*> <[is_max_label g C]> <*>
        is_weighted_graph g vg).
```

The last function, `get_neighbours_spec`, returns the list of neighbours of the given vertex. It returns the location of a structure that is part of another structure (the graph), so we expressed it using the magic wand operator (see 2.2.1 for definition).

Listing 4.6: Specification of `get_neighbours`

```
Definition get_neighbours_spec {A} (get_neighbours : Value A) :
    Prop :=
  forall vg n (g : wgraph nat) c,
    get_neighbours_cost c ->
    triple_fun_n_ary 1 get_neighbours
      (fun v1 v2 => $c <*>
        <[v1 = vg]> <*> <[v2 = Int (Z.of_nat n)]> <*> <[V g n]>
            <*>
        is_weighted_graph g vg)
      (fun v1 v2 res => <exists> L,
        <[is_elem_weighted_unique_list (neighbours g n) (W g n)
            L]> <*>
        is_list (nat_pairs2values L) res <*>
        (is_list (nat_pairs2values L) res <-*>
           is_weighted_graph g vg)).
```

We also axiomatize existence of numbers of time credits satisfying those specifications.

4.2.2 Methods of a heap

Our algorithm makes use of five heap-operating functions. The first one, `mkheap`, creates an empty heap with zero potential. Its running time is linear with respect to the value of its both parameters `n` and `C`.

Listing 4.7: Specification of `mkheap`

```
Definition mkheap_spec {V} (mkheap : Value V) : Prop :=
  forall n C c,
    mkheap_cost n C c ->
    triple_fun mkheap
      (fun v => $1 <*> <[v = Int (Z.of_nat n)]>)
      (fun v => <[
        triple_fun v
          (fun v => $c <*> <[v = Int (Z.of_nat C)]>)
          (is_heap n C empty (fun _ => None) 0)
      ]>).
```

Next is h_insert that inserts a value with a given weight to the heap, increasing its potential. Its running time depends on the current state of the heap and therefore is hard to predict or even write the proper type for this dependency, so instead we use an upper bound t on that time, allowing it to depend only on values used for constructing the heap, that is n and C. To exploit the specification later in the proof, we need to provide t credits. Because t is an overapproximation some credits may stay unconsumed after running the function. We collect those credits in the postcondition with an existential assertion. We also assume that the upper bound on credits is also the upper bound on the increase of heap’s potential. This is crucial to perform the amortized time analisys in specializations of the algorithm with Radix Heap as the priority queue.

Listing 4.8: Specification of h_insert

```
Definition h_insert_spec {V} (h_insert : Value V) : Prop :=
  forall n C (P : nat -> Prop) (W : nat -> option nat) (p s k d
      c t : nat),
    c = t ->
    heap_time_bound n C t ->
    is_set_size P s ->
    s < n ->
    ~ P k ->
    triple_fun_n_ary 2 h_insert
      (fun h v2 v3 => $c <*> <[v2 = Int (Z.of_nat k)]> <*> <[v3
          = Int (Z.of_nat d)]> <*> is_heap n C P W p h)
      (fun h v2 v3 res => (<exists> c’, $c’) <*> <[res = U_val
         ]> <*>
        <exists> p’, <[p’ < p + t]> <*>
          is_heap n C (set_sum P (single k)) (set_value_at W k
             d) p’ h).
```

Another function is h_empty checking whether the heap is empty. We assume the size of the heap is be tracked by its methods and therefore can be accessed in constant time.

Listing 4.9: Specification of h_empty

```
Definition h_empty_spec {V} (h_empty : Value V) : Prop :=
  forall n C (P : nat -> Prop) (W : nat -> option nat) h s c p,
    h_empty_cost c ->
    is_set_size P s ->
    triple_fun h_empty
      (fun v => $c <*> <[v = h]> <*> is_heap n C P W p h)
      (fun v => <[v = Bool (s =? 0)]> <*> is_heap n C P W p h).
```

Next two methods, h_extract_min and h_decrease_key, respectively return and remove the element with the lowest label from the heap, and set the label of a given element to the given smaller value. Just like with h_insert their running times are hard to specify. Our situation is even more complex when we use a structure like Radix Heap, because its complexity analisys requries us to consider total cost of all calls to h_extract_min and h_decrease_key together. We escape this problem by making use of potential. As we have seen above, h_insert always increases the potential by a value limited by a constant, dependent on n and C. This potential can then be decreased by calling either h_extract_min or h_decrease_key and the size of this decrease is always bigger then the decrease in time credits. This way we assert that the total number of credits that will be used to extract the inserted element from the heap is limited by the increase of potential its insertion causes. This increase is limited by the running time of h_insert, what implies that when we use Radix Heap time complexity of the algorithm is asymptotically equivalent to the total time of all insertions, plus iterating over all edges, i.e. $n * t + m$.

Listing 4.10: Specification of h_extract_min

```
Definition h_extract_min_spec {V} (h_extract_min : Value V) :
   Prop :=
  forall n C (P : nat -> Prop) (W : nat -> option nat) p h k d
     c,
    c = p ->
    min_cost_elem P W k ->
    W k = Some d ->
    triple_fun h_extract_min
      (fun v => $c <*> <[v = h]> <*> is_heap n C P W p h)
      (fun v => <exists> c’ cx p’, $c’ <*> <[c’ = p’ + cx]> <*>
        <[v = pair2Value nat2value nat2value (k,d)]> <*>
        is_heap n C (set_remove P k) W p’ h).
```

Listing 4.11: Specification of h_decrease_key

```
Definition h_decrease_key_spec {V} (h_decrease_key : Value V) :
    Prop :=
  forall n C (P : nat -> Prop) (W : nat -> option nat) p h k d
     c,
  c = p ->
  P k ->
  triple_fun_n_ary 2 h_decrease_key
```

```
    (fun v1 v2 v3 => $1 <*> $c <*> <[v1 = h]> <*> <[v2 = Int (Z
       .of_nat k)]> <*>
      <[v3 = Int (Z.of_nat d)]> <*> is_heap n C P W p h)
    (fun v1 v2 v3 res => <exists> c' cx p', $c' <*> <[c' <= p'
       + cx]> <*>
      <[res = U_val]> <*> is_heap n C P (set_value_at W k d) p'
         h).
```

The last heap function is `h_free` that works in time linear w.r.t `n`, `C` and the number of remaining elements and frees the memory occupied by the heap.

Listing 4.12: Specification of `h_free`

```
Definition h_free_spec {V} (h_free : Value V) : Prop :=
  forall n C (P : nat -> Prop) (W : nat -> option nat) s c p,
  is_set_size P s ->
  h_free_cost n C s c ->
  triple_fun h_free
    (is_heap n C P W p <*>+ $c)
    (fun v => <exists> c', $c').
```

4.2.3 Methods of a list

We use three list-operating functions. One of them tests whether the list is empty (i.e. whether it is a `nil`)

Listing 4.13: Specification of `l_is_nil`

```
Definition l_is_nil_spec {V} (l_is_nil : Value V) : Prop :=
  forall (L : list (Value V)) l c,
    l_is_nil_cost c ->
    triple_fun l_is_nil
      (fun v => $c <*> <[v = l]> <*> is_list L l)
      (fun v => <[v = Bool (is_nil_b L)]> <*> is_list L l).
```

The other two return the head and the tail of a nonempty list respectively. Like in the case of `get_neighbours` we use the wand connective to state that the returned tail of a list is a part of the longer list.

Listing 4.14: Specification of `l_head`

```
Definition l_head_spec {V} (l_head : Value V) : Prop :=
  forall (L : list (Value V)) h l c,
    l_head_cost c ->
    triple_fun l_head
      (fun v => $ c <*> <[v = l]> <*> is_list (h::L)%list l)
      (fun v => <[v = h]> <*> is_list (h::L)%list l).
```

Listing 4.15: Specification of l_tail

```
Definition l_tail_spec {V} (l_tail : Value V) : Prop :=
  forall (L : list (Value V)) h l t c,
    l_tail_cost c ->
    triple_fun l_tail
      (fun v => $c <*> <[v = l]> <*> is_list (h::L)%list l)
      (fun v => <[v = t]> <*> is_list L l <*> (is_list L l <-*>
          is_list (h::L)%list t)).
```

4.3 Tactics

To simplify the process of writing the proofscript and improve its readability, we created tactics automating some of the work. Some of those tactics apply proper separation logic rule and sometimes can solve all resulting goals, but most of them are designed to do reordering in a chain of state assertions bound with the star connective. That is because the proofs of equivalence of such chains were pervasive, very elaborate, technical and did not bring any value to the proof. Four of the tactics we defined proved especially useful. They are:

- triple_reorder_pure, that puts all pure assertions on the left of the precondition, allowing to introduce them all into the context by repeatedly applying triple_pull_pure

- triple_reorder_exists, that puts all existential quantifiers at the top of the precondition, allowing to introduce all existential variables into the context by repeatedly applying triple_pull_exists

- triple_reorder_credits, that puts all credit assertions on the left of the precondition

- prove_implies and prove_implies_rev, that try to prove or simplify an entailment, in which assertions are permutations each other - this is similar to the behaviour of xsimpl from [Charguéraud, 2024], but not the same, because we tried to define it independently

4.4 Verification of auxiliary function

Some parts of the code occurred in multiple places and their behaviour was not specific to the algorithm. We changed such parts into auxiliary functions we verified separately. Those functions are: assign_array_at, read_array_at, incr, init_array.

4.5 The proof

We prove the following specification:

Listing 4.16: 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]>).
```

This theorem starts with requirements that its parameters are closed terms and satisfy their corresponding specifications. Then we introduce three constants with which we state the asymptotic time bound of the function ($c_0 + c_m * m + c_n * n * t$, that is $O(m + n * t)$). We also make some basic assumptions about the shape of the graph and write specification using the triple for n-ary functions we defined in 2.2.2. The proof is very long but most of the lines serve the purpose of exploiting the specifications of auxiliary functions. Those places together with control flow statements required us to explicitly state pre- and postconditions of the rules we apply. We wanted to avoid referring to Coq-generated names in the proofscript, so in every such place we use `lazymatch` on the goal to assign a temporary name to a variable.

Chapter 5

Conclusion and future work

We took the idea of using potential to verify time complexity from [Charguéraud and Pottier, 2015], although authors use this technique to verify the amortized complexity of a set of methods, while we use it to exploit the amortized complexity of methods to verify worst case complexity of the code using them.

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.