

Dijkstra's Algorithm

Benedikt Nordhoff Peter Lammich

May 26, 2024

Abstract

We implement and prove correct Dijkstra's algorithm for the single source shortest path problem, conceived in 1956 by E. Dijkstra. The algorithm is implemented using the data refinement framework for monadic, nondeterministic programs. An efficient implementation is derived using data structures from the Isabelle Collection Framework.

Contents

1	Introduction and Overview	3
2	Miscellaneous Lemmas	3
3	Graphs	4
3.1	Definitions	5
3.2	Basic operations on Graphs	5
3.3	Paths	6
3.3.1	Splitting Paths	7
3.4	Weighted Graphs	8
4	Weights for Dijkstra's Algorithm	9
4.1	Type Classes Setup	9
4.2	Adding Infinity	10
4.2.1	Unboxing	11
5	Dijkstra's Algorithm	12
5.1	Graph's for Dijkstra's Algorithm	12
5.2	Specification of Correct Result	12
5.3	Dijkstra's Algorithm	13
5.4	Structural Refinement of Update	15
5.5	Refinement to Cached Weights	16

6	Graph Interface	19
6.1	Adjacency Lists	22
6.2	Record Based Interface	22
6.3	Refinement Framework Bindings	23
7	Generic Algorithms for Graphs	24
8	Implementing Graphs by Maps	26
9	Graphs by Hashmaps	31
10	Implementation of Dijkstra’s-Algorithm using the ICF	32
11	Implementation of Dijkstra’s-Algorithm using Automatic De-	36
	terminization	
11.1	Setup	36
11.1.1	Infinity	36
11.1.2	Graph	38
11.2	Refinement	39
12	Performance Test	42

1 Introduction and Overview

Dijkstra’s algorithm [1] is an algorithm used to find shortest paths from one given vertex to all other vertices in a non-negatively weighted graph.

The implementation of the algorithm is meant to be an application of our extensions to the Isabelle Collections Framework (ICF) [4, 6, 7]. Moreover, it serves as a test case for our data refinement framework [5]. We use ICF-Maps to efficiently represent the graph and result and the newly introduced unique priority queues for the work list.

For a documentation of the refinement framework see [5], that also contains a userguide and some simpler examples.

The development utilizes a stepwise refinement approach. Starting from an abstract algorithm that has a nice correctness proof, we stepwise refine the algorithm until we end up with an efficient implementation, for that we generate code using Isabelle/HOL’s code generator[2, 3].

Structure of the Submission. The abstract version of the algorithm with the correctness proof, as well as the main refinement steps are contained in the theory `Dijkstra`. The refinement steps involving the ICF and code generation are contained in `Dijkstra-Impl`. The theory `Infty` contains an extension of numbers with an infinity element. The theory `Graph` contains a formalization of graphs, paths, and related concepts. The theories `GraphSpec`, `GraphGA`, `GraphByMap`, `HashGraphImpl` contain an ICF-style specification of graphs. The theory `Test` contains a small performance test on random graphs. It uses the ML-code generated by the code generator.

2 Miscellaneous Lemmas

```
theory Dijkstra-Misc
```

```
imports Main
```

```
begin
```

```
  inductive-set least-map for f S where
```

```
     $\llbracket x \in S; \forall x' \in S. f\ x \leq f\ x' \rrbracket \implies x \in \text{least-map } f\ S$ 
```

```
  lemma least-map-subset: least-map f S  $\subseteq$  S
```

```
    <proof>
```

```
  lemmas least-map-lemD = subsetD[OF least-map-subset]
```

```
  lemma least-map-leD:
```

```
    assumes  $x \in \text{least-map } f\ S$ 
```

```
    assumes  $y \in S$ 
```

```
    shows  $f\ x \leq f\ y$ 
```

```
    <proof>
```

lemma *least-map-empty*[simp]: *least-map* *f* {} = {}
⟨*proof*⟩

lemma *least-map-singleton*[simp]: *least-map* (*f*::'a⇒'b::order) {*x*} = {*x*}
⟨*proof*⟩

lemma *least-map-insert-min*:
fixes *f*::'a⇒'b::order
assumes $\forall y \in S. f\ x \leq f\ y$
shows $x \in \text{least-map } f\ (\text{insert } x\ S)$
⟨*proof*⟩

lemma *least-map-insert-nmin*:
[[$x \in \text{least-map } f\ S; f\ x \leq f\ a$]] ⇒ $x \in \text{least-map } f\ (\text{insert } a\ S)$
⟨*proof*⟩

context *semilattice-inf*
begin
lemmas [simp] = *inf-absorb1 inf-absorb2*

lemma *inf-absorb-less*[simp]:
 $a < b \implies \text{inf } a\ b = a$
 $a < b \implies \text{inf } b\ a = a$
⟨*proof*⟩

end

end

3 Graphs

theory *Graph*
imports *Main*
begin

This theory defines a notion of graphs. A graph is a record that contains a set of nodes V and a set of labeled edges $E \subseteq V \times W \times V$, where W are the edge labels.

3.1 Definitions

A graph is represented by a record.

```
record ('v,'w) graph =
  nodes :: 'v set
  edges :: ('v × 'w × 'v) set
```

In a valid graph, edges only go from nodes to nodes.

```
locale valid-graph =
  fixes G :: ('v,'w) graph
  assumes E-valid: fst'edges G ⊆ nodes G
  snd'snd'edges G ⊆ nodes G
```

begin

```
abbreviation V ≡ nodes G
abbreviation E ≡ edges G
```

```
lemma E-validD: assumes (v,e,v') ∈ E
shows v ∈ V v' ∈ V
⟨proof⟩
```

end

3.2 Basic operations on Graphs

The empty graph.

```
definition empty where
  empty ≡ (| nodes = {}, edges = {} |)
```

Adds a node to a graph.

```
definition add-node where
  add-node v g ≡ (| nodes = insert v (nodes g), edges=edges g |)
```

Deletes a node from a graph. Also deletes all adjacent edges.

```
definition delete-node where delete-node v g ≡ (|
  nodes = nodes g - {v},
  edges = edges g ∩ (-{v}) × UNIV × (-{v})
|)
```

Adds an edge to a graph.

```
definition add-edge where add-edge v e v' g ≡ (|
  nodes = {v,v'} ∪ nodes g,
  edges = insert (v,e,v') (edges g)
|)
```

Deletes an edge from a graph.

```
definition delete-edge where delete-edge v e v' g ≡ (|
  nodes = nodes g, edges = edges g - {(v,e,v')} |)
```

Successors of a node.

definition $succ :: ('v, 'w) graph \Rightarrow 'v \Rightarrow ('w \times 'v) set$
where $succ\ G\ v \equiv \{(w, v') . (v, w, v') \in edges\ G\}$

Now follow some simplification lemmas.

lemma $empty-valid[simp]: valid-graph\ empty$
 $\langle proof \rangle$

lemma $add-node-valid[simp]: assumes\ valid-graph\ g$
shows $valid-graph\ (add-node\ v\ g)$
 $\langle proof \rangle$

lemma $delete-node-valid[simp]: assumes\ valid-graph\ g$
shows $valid-graph\ (delete-node\ v\ g)$
 $\langle proof \rangle$

lemma $add-edge-valid[simp]: assumes\ valid-graph\ g$
shows $valid-graph\ (add-edge\ v\ e\ v'\ g)$
 $\langle proof \rangle$

lemma $delete-edge-valid[simp]: assumes\ valid-graph\ g$
shows $valid-graph\ (delete-edge\ v\ e\ v'\ g)$
 $\langle proof \rangle$

lemma $succ-finite[simp, intro]: finite\ (edges\ G) \Longrightarrow finite\ (succ\ G\ v)$
 $\langle proof \rangle$

lemma $nodes-empty[simp]: nodes\ empty = \{\}$ $\langle proof \rangle$

lemma $edges-empty[simp]: edges\ empty = \{\}$ $\langle proof \rangle$

lemma $succ-empty[simp]: succ\ empty\ v = \{\}$ $\langle proof \rangle$

lemma $nodes-add-node[simp]: nodes\ (add-node\ v\ g) = insert\ v\ (nodes\ g)$
 $\langle proof \rangle$

lemma $nodes-add-edge[simp]:$
 $nodes\ (add-edge\ v\ e\ v'\ g) = insert\ v\ (insert\ v'\ (nodes\ g))$
 $\langle proof \rangle$

lemma $edges-add-edge[simp]:$
 $edges\ (add-edge\ v\ e\ v'\ g) = insert\ (v, e, v')\ (edges\ g)$
 $\langle proof \rangle$

lemma $edges-add-node[simp]:$
 $edges\ (add-node\ v\ g) = edges\ g$
 $\langle proof \rangle$

lemma **(in** $valid-graph$) $succ-subset: succ\ G\ v \subseteq UNIV \times V$
 $\langle proof \rangle$

3.3 Paths

A path is represented by a list of adjacent edges.

type-synonym $('v, 'w) path = ('v \times 'w \times 'v) list$

context $valid-graph$

begin

The following predicate describes a valid path:

fun *is-path* :: '*v* ⇒ ('*v*, '*w*) *path* ⇒ '*v* ⇒ *bool* **where**
 is-path *v* [] *v'* ⇔ *v=v' ∧ v'∈V* |
 is-path *v* ((*v1*, *w*, *v2*)#*p*) *v'* ⇔ *v=v1 ∧ (v1, w, v2)∈E ∧ is-path v2 p v'*

lemma *is-path-simps*[*simp*, *intro!*]:
 is-path *v* [] *v* ⇔ *v∈V*
 is-path *v* [(*v*, *w*, *v'*)] *v'* ⇔ (*v*, *w*, *v'*)∈*E*
 ⟨*proof*⟩

lemma *is-path-memb*[*simp*]:
 is-path *v* *p* *v'* ⇒ *v∈V ∧ v'∈V*
 ⟨*proof*⟩

lemma *is-path-split*:
 is-path *v* (*p1*@*p2*) *v'* ⇔ (∃ *u*. *is-path* *v* *p1* *u* ∧ *is-path* *u* *p2* *v'*)
 ⟨*proof*⟩

lemma *is-path-split'*[*simp*]:
 is-path *v* (*p1*@(*u*, *w*, *u'*)#*p2*) *v'*
 ⇔ *is-path* *v* *p1* *u* ∧ (*u*, *w*, *u'*)∈*E* ∧ *is-path* *u'* *p2* *v'*
 ⟨*proof*⟩

end

Set of intermediate vertices of a path. These are all vertices but the last one. Note that, if the last vertex also occurs earlier on the path, it is contained in *int-vertices*.

definition *int-vertices* :: ('*v*, '*w*) *path* ⇒ '*v* *set* **where**
 int-vertices *p* ≡ *set* (*map fst p*)

lemma *int-vertices-simps*[*simp*]:
 int-vertices [] = {}
 int-vertices (*vv*#*p*) = *insert* (*fst vv*) (*int-vertices* *p*)
 int-vertices (*p1*@*p2*) = *int-vertices* *p1* ∪ *int-vertices* *p2*
 ⟨*proof*⟩

lemma (**in** *valid-graph*) *int-vertices-subset*:
 is-path *v* *p* *v'* ⇒ *int-vertices* *p* ⊆ *V*
 ⟨*proof*⟩

lemma *int-vertices-empty*[*simp*]: *int-vertices* *p* = {} ⇔ *p*=[]
 ⟨*proof*⟩

3.3.1 Splitting Paths

Split a path at the point where it first leaves the set *W*:

lemma (in *valid-graph*) *path-split-set*:
assumes *is-path* v p v' **and** $v \in W$ **and** $v' \notin W$
obtains $p1$ $p2$ u w u' **where**
 $p = p1 @ (u, w, u') \# p2$ **and**
int-vertices $p1 \subseteq W$ **and** $u \in W$ **and** $u' \notin W$
 ⟨*proof*⟩

Split a path at the point where it first enters the set W :

lemma (in *valid-graph*) *path-split-set'*:
assumes *is-path* v p v' **and** $v' \in W$
obtains $p1$ $p2$ u **where**
 $p = p1 @ p2$ **and**
is-path v $p1$ u **and**
is-path u $p2$ v' **and**
int-vertices $p1 \subseteq -W$ **and** $u \in W$
 ⟨*proof*⟩

Split a path at the point where a given vertex is first visited:

lemma (in *valid-graph*) *path-split-vertex*:
assumes *is-path* v p v' **and** $u \in \text{int-vertices } p$
obtains $p1$ $p2$ **where**
 $p = p1 @ p2$ **and**
is-path v $p1$ u **and**
 $u \notin \text{int-vertices } p1$
 ⟨*proof*⟩

3.4 Weighted Graphs

locale *valid-mgraph* = *valid-graph* G **for** $G :: ('v, 'w :: \text{monoid-add}) \text{ graph}$

definition *path-weight* :: $('v, 'w :: \text{monoid-add}) \text{ path} \Rightarrow 'w$
where *path-weight* $p \equiv \text{sum-list } (\text{map } (\text{fst} \circ \text{snd}) p)$

lemma *path-weight-split[simp]*:
 $(\text{path-weight } (p1 @ p2) :: 'w :: \text{monoid-add}) = \text{path-weight } p1 + \text{path-weight } p2$
 ⟨*proof*⟩

lemma *path-weight-empty[simp]*: $\text{path-weight } [] = 0$
 ⟨*proof*⟩

lemma *path-weight-cons[simp]*:
 $(\text{path-weight } (e \# p) :: 'w :: \text{monoid-add}) = \text{fst } (\text{snd } e) + \text{path-weight } p$
 ⟨*proof*⟩

end

4 Weights for Dijkstra's Algorithm

```
theory Weight  
imports Complex-Main  
begin
```

In this theory, we set up a type class for weights, and a typeclass for weights with an infinity element. The latter one is used internally in Dijkstra's algorithm.

Moreover, we provide a datatype that adds an infinity element to a given base type.

4.1 Type Classes Setup

```
class weight = ordered-ab-semigroup-add + comm-monoid-add + linorder  
begin
```

```
lemma add-nonneg-nonneg [simp]:  
  assumes  $0 \leq a$  and  $0 \leq b$  shows  $0 \leq a + b$   
  <proof>
```

```
lemma add-nonpos-nonpos[simp]:  
  assumes  $a \leq 0$  and  $b \leq 0$  shows  $a + b \leq 0$   
  <proof>
```

```
lemma add-nonneg-eq-0-iff:  
  assumes  $x: 0 \leq x$  and  $y: 0 \leq y$   
  shows  $x + y = 0 \iff x = 0 \wedge y = 0$   
  <proof>
```

```
lemma add-incr:  $0 \leq b \implies a \leq a + b$   
  <proof>
```

```
lemma add-incr-left[simp, intro!]:  $0 \leq b \implies a \leq b + a$   
  <proof>
```

```
lemma sum-not-less[simp, intro!]:  
   $0 \leq b \implies \neg (a + b < a)$   
   $0 \leq a \implies \neg (a + b < b)$   
  <proof>
```

```
end
```

```
instance nat :: weight <proof>  
instance int :: weight <proof>  
instance rat :: weight <proof>  
instance real :: weight <proof>
```

```
term top
```

```

class top-weight = order-top + weight +
  assumes inf-add-right[simp]: a + top = top
begin

```

```

lemma inf-add-left[simp]: top + a = top
  ⟨proof⟩

```

```

lemmas [simp] = top-unique less-top[symmetric]

```

```

lemma not-less-inf[simp]:
  ¬ (a < top) ⟷ a=top
  ⟨proof⟩

```

```

end

```

4.2 Adding Infinity

We provide a standard way to add an infinity element to any type.

```

datatype 'a infty = Infty | Num 'a

```

```

primrec val where val (Num d) = d

```

```

lemma num-val-iff[simp]: e ≠ Infty ⟹ Num (val e) = e ⟨proof⟩

```

```

type-synonym NatB = nat infty

```

```

instantiation infty :: (weight) top-weight

```

```

begin

```

```

  definition (0::'a infty) == Num 0

```

```

  definition top ≡ Infty

```

```

fun less-eq-infty where

```

```

  less-eq Infty (Num _) ⟷ False |

```

```

  less-eq - Infty ⟷ True |

```

```

  less-eq (Num a) (Num b) ⟷ a ≤ b

```

```

lemma [simp]: Infty ≤ a ⟷ a = Infty
  ⟨proof⟩

```

```

fun less-infty where

```

```

  less Infty - ⟷ False |

```

```

  less (Num _) Infty ⟷ True |

```

```

  less (Num a) (Num b) ⟷ a < b

```

```

lemma [simp]: less a Infty ⟷ a ≠ Infty
  ⟨proof⟩

```

```

fun plus-infty where
  plus - Infty = Infty |
  plus Infty - = Infty |
  plus (Num a) (Num b) = Num (a+b)

```

```

lemma [simp]: plus Infty a = Infty <proof>

```

```

instance
  <proof>
end

```

4.2.1 Unboxing

Conversion between the constants defined by the typeclass, and the concrete functions on the 'a infty type.

```

lemma infty-inf-unbox:

```

```

  Num a ≠ top
  top ≠ Num a
  Infty = top
  <proof>

```

```

lemma infty-ord-unbox:

```

```

  Num a ≤ Num b ↔ a ≤ b
  Num a < Num b ↔ a < b
  <proof>

```

```

lemma infty-plus-unbox:

```

```

  Num a + Num b = Num (a+b)
  <proof>

```

```

lemma infty-zero-unbox:

```

```

  Num a = 0 ↔ a = 0
  Num 0 = 0
  <proof>

```

```

lemmas infty-unbox =

```

```

  infty-inf-unbox infty-zero-unbox infty-ord-unbox infty-plus-unbox

```

```

lemma inf-not-zero[simp]:

```

```

  top ≠ (0::- infty) (0::- infty) ≠ top
  <proof>

```

```

lemma num-val-iff'[simp]: e ≠ top ⇒ Num (val e) = e

```

```

  <proof>

```

```

lemma infty-neE:

```

```

  [[a ≠ Infty; ∧ d. a = Num d ⇒ P]] ⇒ P
  [[a ≠ top; ∧ d. a = Num d ⇒ P]] ⇒ P

```

<proof>

end

5 Dijkstra's Algorithm

theory *Dijkstra*

imports

Graph

Dijkstra-Misc

Collections.Refine-Dft-ICF

Weight

begin

This theory defines Dijkstra's algorithm. First, a correct result of Dijkstra's algorithm w.r.t. a graph and a start vertex is specified. Then, the refinement framework is used to specify Dijkstra's Algorithm, prove it correct, and finally refine it to datatypes that are closer to an implementation than the original specification.

5.1 Graph's for Dijkstra's Algorithm

A graph annotated with weights.

locale *weighted-graph = valid-graph* *G*

for *G* :: ('V,'W::weight) graph

5.2 Specification of Correct Result

context *weighted-graph*

begin

A result of Dijkstra's algorithm is correct, if it is a map from nodes v to the shortest path from the start node $v0$ to v . Iff there is no such path, the node is not in the map.

definition *is-shortest-path-map* :: 'V \Rightarrow ('V \rightarrow ('V,'W) path) \Rightarrow bool

where

is-shortest-path-map $v0$ $res \equiv \forall v \in V. (case\ res\ v\ of$

None $\Rightarrow \neg(\exists p. is-path\ v0\ p\ v) \mid$

Some $p \Rightarrow is-path\ v0\ p\ v$

$\wedge (\forall p'. is-path\ v0\ p'\ v \longrightarrow path-weight\ p \leq path-weight\ p')$

)

end

The following function returns the weight of an optional path, where *None* is interpreted as infinity.

fun *path-weight'* **where**

path-weight' *None* = *top* \mid

path-weight' (*Some* p) = *Num* (*path-weight* p)

5.3 Dijkstra's Algorithm

The state in the main loop of the algorithm consists of a workset wl of vertexes that still need to be explored, and a map res that contains the current shortest path for each vertex.

type-synonym (V, W) $state = (V\ set) \times (V \rightarrow (V, W)\ path)$

The preconditions of Dijkstra's algorithm, i.e., that it operates on a valid and finite graph, and that the start node is a node of the graph, are summarized in a locale.

```

locale Dijkstra = weighted-graph  $G$ 
for  $G :: (V, W :: weight)\ graph+$ 
fixes  $v0 :: V$ 
assumes finite[simp, intro!]: finite  $V$  finite  $E$ 
assumes v0-in-V[simp, intro!]:  $v0 \in V$ 
assumes nonneg-weights[simp, intro!]:  $(v, w, v') \in edges\ G \implies 0 \leq w$ 
begin

```

Paths have non-negative weights.

lemma *path-nonneg-weight*: $is-path\ v\ p\ v' \implies 0 \leq path-weight\ p$
<proof>

Invariant of the main loop:

- The workset only contains nodes of the graph.
- If the result set contains a path for a node, it is actually a path, and uses only intermediate vertices outside the workset.
- For all vertices outside the workset, the result map contains the shortest path.
- For all vertices in the workset, the result map contains the shortest path among all paths that only use intermediate vertices outside the workset.

definition *dinvar* $\sigma \equiv let\ (wl, res) = \sigma\ in$
 $wl \subseteq V \wedge$
 $(\forall v \in V. \forall p. res\ v = Some\ p \longrightarrow is-path\ v0\ p\ v \wedge int-vertices\ p \subseteq V - wl) \wedge$
 $(\forall v \in V - wl. \forall p. is-path\ v0\ p\ v$
 $\longrightarrow path-weight'\ (res\ v) \leq path-weight'\ (Some\ p)) \wedge$
 $(\forall v \in wl. \forall p. is-path\ v0\ p\ v \wedge int-vertices\ p \subseteq V - wl$
 $\longrightarrow path-weight'\ (res\ v) \leq path-weight'\ (Some\ p)$
 $)$

Sanity check: The invariant is strong enough to imply correctness of result.

lemma *invar-imp-correct*: $dinvar\ (\{\}, res) \implies is-shortest-path-map\ v0\ res$

$\langle proof \rangle$

The initial workset contains all vertices. The initial result maps $v0$ to the empty path, and all other vertices to *None*.

definition $dinit :: ('V, 'W) \text{ state nres where}$
 $dinit \equiv SPEC (\lambda(wl, res) .$
 $wl = V \wedge res\ v0 = Some [] \wedge (\forall v \in V - \{v0\}. res\ v = None))$

The initial state satisfies the invariant.

lemma $dinit\text{-invar}: dinit \leq SPEC\ dinvar$
 $\langle proof \rangle$

In each iteration, the main loop of the algorithm pops a minimal node from the workset, and then updates the result map accordingly.

Pop a minimal node from the workset. The node is minimal in the sense that the length of the current path for that node is minimal.

definition $pop\text{-min} :: ('V, 'W) \text{ state} \Rightarrow ('V \times ('V, 'W) \text{ state}) \text{ nres where}$
 $pop\text{-min}\ \sigma \equiv do \{$
 $let\ (wl, res) = \sigma;$
 $ASSERT\ (wl \neq \{\});$
 $v \leftarrow RES\ (least\text{-map}\ (path\text{-weight}' \circ res)\ wl);$
 $RETURN\ (v, (wl - \{v\}, res))$
 $\}$

Updating the result according to a node v is done by checking, for each successor node, whether the path over v is shorter than the path currently stored into the result map.

inductive $update\text{-spec} :: 'V \Rightarrow ('V, 'W) \text{ state} \Rightarrow ('V, 'W) \text{ state} \Rightarrow bool$
where
 $\llbracket \forall v' \in V.$
 $res'\ v' \in least\text{-map}\ path\text{-weight}' (\$
 $\{ res\ v' \} \cup \{ Some\ (p@[v, w, v']) \mid p\ w.\ res\ v = Some\ p \wedge (v, w, v') \in E \}$
 $)$
 $\rrbracket \Longrightarrow update\text{-spec}\ v\ (wl, res)\ (wl, res')$

In order to ease the refinement proof, we will assert the following precondition for updating.

definition $update\text{-pre} :: 'V \Rightarrow ('V, 'W) \text{ state} \Rightarrow bool \text{ where}$
 $update\text{-pre}\ v\ \sigma \equiv let\ (wl, res) = \sigma\ in\ v \in V$
 $\wedge (\forall v' \in V - wl.\ v' \neq v \longrightarrow (\forall p.\ is\text{-path}\ v0\ p\ v'$
 $\longrightarrow path\text{-weight}'\ (res\ v') \leq path\text{-weight}'\ (Some\ p)))$
 $\wedge (\forall v' \in V.\ \forall p.\ res\ v' = Some\ p \longrightarrow is\text{-path}\ v0\ p\ v')$

definition $update :: 'V \Rightarrow ('V, 'W) \text{ state} \Rightarrow ('V, 'W) \text{ state nres where}$
 $update\ v\ \sigma \equiv do \{ ASSERT\ (update\text{-pre}\ v\ \sigma); SPEC\ (update\text{-spec}\ v\ \sigma) \}$

Finally, we define Dijkstra's algorithm:

definition *dijkstra* **where**

```

dijkstra ≡ do {
  σ 0 ← dinit;
  (-, res) ← WHILETdinvar (λ(wl, -). wl ≠ {})
    (λσ.
      do { (v, σ') ← pop-min σ; update v σ' }
    )
  σ 0;
  RETURN res }

```

The following theorem states (total) correctness of Dijkstra's algorithm.

theorem *dijkstra-correct*: *dijkstra* ≤ SPEC (*is-shortest-path-map* v0)
 ⟨proof⟩

5.4 Structural Refinement of Update

Now that we have proved correct the initial version of the algorithm, we start refinement towards an efficient implementation.

First, the update function is refined to iterate over each successor of the selected node, and update the result on demand.

definition *winvar*

```

:: 'V ⇒ 'V set ⇒ - ⇒ ('W × 'V) set ⇒ ('V, 'W) state ⇒ bool where
winvar v wl res it σ ≡ let (wl', res') = σ in wl' = wl
  ∧ (∀ v' ∈ V.
    res' v' ∈ least-map path-weight' (
      { res v' } ∪ { Some (p@[v, w, v']) | p w. res v = Some p
        ∧ (w, v') ∈ succ G v - it }
    ))
  ∧ (∀ v' ∈ V. ∀ p. res' v' = Some p → is-path v0 p v')
  ∧ res' v = res v

```

definition *update'* :: 'V ⇒ ('V, 'W) state ⇒ ('V, 'W) state nres **where**

```

update' v σ ≡ do {
  ASSERT (update-pre v σ);
  let (wl, res) = σ;
  let wv = path-weight' (res v);
  let pv = res v;
  FOREACHwinvar v wl res (succ G v) (λ(w', v') (wl, res).
    if (wv + Num w' < path-weight' (res v')) then do {
      ASSERT (v' ∈ wl ∧ pv ≠ None);
      RETURN (wl, res(v' ↦ the pv@[v, w', v']))
    } else RETURN (wl, res)
  ) (wl, res) }

```

lemma *update'-refines*:

assumes v' = v **and** σ' = σ

shows $update' v' \sigma' \leq \Downarrow Id (update v \sigma)$
 ⟨proof⟩

We integrate the new update function into the main algorithm:

definition *dijkstra'* **where**

```

dijkstra' ≡ do {
  σ0 ← dinit;
  (-,res) ← WHILETdinvar (λ(wl,-). wl≠{ })
    (λσ. do {(v,σ') ← pop-min σ; update' v σ'})
    σ0;
  RETURN res
}

```

lemma *dijkstra'-refines*: $dijkstra' \leq \Downarrow Id dijkstra$
 ⟨proof⟩

end

5.5 Refinement to Cached Weights

Next, we refine the data types of the workset and the result map. The workset becomes a map from nodes to their current weights. The result map stores, in addition to the shortest path, also the weight of the shortest path. Moreover, we store the shortest paths in reversed order, which makes appending new edges more efficient.

These refinements allow to implement the workset as a priority queue, and save recomputation of the path weights in the inner loop of the algorithm.

type-synonym $('V, 'W) mwl = ('V \rightarrow 'W \text{ infty})$

type-synonym $('V, 'W) mres = ('V \rightarrow (('V, 'W) \text{ path} \times 'W))$

type-synonym $('V, 'W) mstate = ('V, 'W) mwl \times ('V, 'W) mres$

Map a path with cached weight to one without cached weight.

```

fun mpath' :: (('V, 'W) path × 'W) option → ('V, 'W) path where
  mpath' None = None |
  mpath' (Some (p,w)) = Some p

```

```

fun mpath-weight' :: (('V, 'W) path × 'W) option ⇒ ('W::weight) infty where
  mpath-weight' None = top |
  mpath-weight' (Some (p,w)) = Num w

```

context *Dijkstra*

begin

definition $\alpha w :: ('V, 'W) mwl \Rightarrow 'V \text{ set}$ **where** $\alpha w \equiv dom$

definition $\alpha r :: ('V, 'W) mres \Rightarrow 'V \rightarrow ('V, 'W) \text{ path}$ **where**

$\alpha r \equiv \lambda res v. \text{ case } res \text{ of } None \Rightarrow None \mid Some (p,w) \Rightarrow Some (rev p)$

definition $\alpha s :: ('V, 'W) mstate \Rightarrow ('V, 'W) \text{ state}$ **where**

$\alpha s \equiv \text{map-prod } \alpha w \alpha r$

Additional invariants for the new state. They guarantee that the cached weights are consistent.

definition *res-invarm* :: ($'V \rightarrow (('V, 'W) \text{ path} \times 'W)$) \Rightarrow *bool* **where**
res-invarm *res* \equiv ($\forall v. \text{case } \text{res } v \text{ of}$
None \Rightarrow *True* |
Some (*p,w*) \Rightarrow *w = path-weight (rev p)*)

definition *dinvarm* :: ($'V, 'W$) *mstate* \Rightarrow *bool* **where**
dinvarm $\sigma \equiv$ *let* (*wl,res*) = σ *in*
 $(\forall v \in \text{dom } \text{wl}. \text{the } (\text{wl } v) = \text{mpath-weight}' (\text{res } v)) \wedge \text{res-invarm } \text{res}$

lemma *mpath-weight'-correct*: $\llbracket \text{dinvarm } (\text{wl}, \text{res}) \rrbracket \Longrightarrow$
 $\text{mpath-weight}' (\text{res } v) = \text{path-weight}' (\alpha \text{r } \text{res } v)$

$\langle \text{proof} \rangle$

lemma *mpath'-correct*: $\llbracket \text{dinvarm } (\text{wl}, \text{res}) \rrbracket \Longrightarrow$
 $\text{mpath}' (\text{res } v) = \text{map-option rev } (\alpha \text{r } \text{res } v)$
 $\langle \text{proof} \rangle$

lemma *wl-weight-correct*:
assumes *INV*: *dinvarm* (*wl,res*)
assumes *WLW*: *wl v = Some w*
shows *path-weight' (αr res v) = w*
 $\langle \text{proof} \rangle$

The initial state is constructed using an iterator:

definition *mdinit* :: ($'V, 'W$) *mstate nres* **where**
mdinit \equiv *do* {
wl \leftarrow *FOREACH* *V* ($\lambda v \text{ wl}. \text{RETURN } (\text{wl}(v \mapsto \text{Infty}))$) *Map.empty*;
RETURN (*wl*($v0 \mapsto \text{Num } 0$), [$v0 \mapsto ([], 0)$])
}

lemma *mdinit-refines*: $\text{mdinit} \leq \Downarrow (\text{build-rel } \alpha \text{s } \text{dinvarm}) \text{dinit}$
 $\langle \text{proof} \rangle$

The new pop function:

definition
mpop-min :: ($'V, 'W$) *mstate* \Rightarrow ($'V \times 'W \text{ infty} \times ('V, 'W) \text{ mstate}$) *nres*
where
mpop-min $\sigma \equiv$ *do* {
let (*wl,res*) = σ ;
 $(v, w, \text{wl}') \leftarrow \text{prio-pop-min } \text{wl}$;
RETURN (*v,w,(wl',res)*)
}

lemma *mpop-min-refines*:
 $\llbracket (\sigma, \sigma') \in \text{build-rel } \alpha \text{s } \text{dinvarm} \rrbracket \Longrightarrow$
 $\text{mpop-min } \sigma \leq$

$\Downarrow(\text{build-rel}$
 $(\lambda(v,w,\sigma). (v,\alpha s \sigma))$
 $(\lambda(v,w,\sigma). \text{dinvarm } \sigma \wedge w = \text{mpath-weight}'(\text{snd } \sigma v))$
 $(\text{pop-min } \sigma')$

— The two algorithms are structurally different, so we use the nofail/inres method to prove refinement.

$\langle \text{proof} \rangle$

The new update function:

definition $\text{uinvarm } v \text{ wl res it } \sigma \equiv$
 $\text{uinvar } v \text{ wl res it } (\alpha s \sigma) \wedge \text{dinvarm } \sigma$

definition $\text{mupdate} :: 'V \Rightarrow 'W \text{ infty} \Rightarrow ('V, 'W) \text{ mstate} \Rightarrow ('V, 'W) \text{ mstate nres}$
where

$\text{mupdate } v \text{ ww } \sigma \equiv \text{do } \{$
 $\text{ASSERT } (\text{update-pre } v (\alpha s \sigma) \wedge \text{ww} = \text{mpath-weight}'(\text{snd } \sigma v));$
 $\text{let } (\text{wl}, \text{res}) = \sigma;$
 $\text{let } \text{pv} = \text{mpath}'(\text{res } v);$
 $\text{FOREACH}^{\text{uinvarm } v} (\alpha w \text{ wl}) (\alpha r \text{ res}) (\text{succ } G \ v) (\lambda(w',v') (\text{wl}, \text{res}).$
 $\text{if } (\text{ww} + \text{Num } w' < \text{mpath-weight}'(\text{res } v')) \text{ then do } \{$
 $\text{ASSERT } (v' \in \text{dom } \text{wl} \wedge \text{pv} \neq \text{None});$
 $\text{ASSERT } (\text{ww} \neq \text{Infty});$
 $\text{RETURN } (\text{wl}(v' \mapsto \text{ww} + \text{Num } w'),$
 $\text{res}(v' \mapsto ((v, w', v') \# \text{the } \text{pv}, \text{val } \text{ww} + w'))$
 $\} \text{ else RETURN } (\text{wl}, \text{res})$
 $\} (\text{wl}, \text{res})$
 $\}$

lemma mupdate-refines :

assumes $\text{SREF}: (\sigma, \sigma') \in \text{build-rel } \alpha s \text{ dinvarm}$

assumes $\text{WV}: \text{ww} = \text{mpath-weight}'(\text{snd } \sigma v)$

assumes $\text{VV}': v' = v$

shows $\text{mupdate } v \text{ ww } \sigma \leq \Downarrow(\text{build-rel } \alpha s \text{ dinvarm}) (\text{update}' \ v' \ \sigma')$

$\langle \text{proof} \rangle$

Finally, we assemble the refined algorithm:

definition mdijkstra **where**

$\text{mdijkstra} \equiv \text{do } \{$
 $\sigma 0 \leftarrow \text{mdinit};$
 $(-, \text{res}) \leftarrow \text{WHILE}_T^{\text{dinvarm}} (\lambda(\text{wl}, -). \text{dom } \text{wl} \neq \{\})$
 $(\lambda \sigma. \text{do } \{ (v, \text{ww}, \sigma') \leftarrow \text{mpop-min } \sigma; \text{mupdate } v \text{ ww } \sigma' \})$
 $\sigma 0;$
 $\text{RETURN } \text{res}$
 $\}$

lemma mdijkstra-refines : $\text{mdijkstra} \leq \Downarrow(\text{build-rel } \alpha r \text{ res-invarm}) \text{dijkstra}'$

$\langle \text{proof} \rangle$

end

end

6 Graph Interface

```
theory GraphSpec
imports Main Graph
    Collections.Collections
```

begin

This theory defines an ICF-style interface for graphs.

```
type-synonym ('V,'W,'G) graph- $\alpha$  = 'G  $\Rightarrow$  ('V,'W) graph
```

```
locale graph =
  fixes  $\alpha$  :: 'G  $\Rightarrow$  ('V,'W) graph
  fixes invar :: 'G  $\Rightarrow$  bool
  assumes finite[simp, intro!]:
    invar g  $\Rightarrow$  finite (nodes ( $\alpha$  g))
    invar g  $\Rightarrow$  finite (edges ( $\alpha$  g))
  assumes valid: invar g  $\Rightarrow$  valid-graph ( $\alpha$  g)
```

```
type-synonym ('V,'W,'G) graph-empty = unit  $\Rightarrow$  'G
```

```
locale graph-empty = graph +
  constrains  $\alpha$  :: 'G  $\Rightarrow$  ('V,'W) graph
  fixes empty :: unit  $\Rightarrow$  'G
  assumes empty-correct:
     $\alpha$  (empty ()) = Graph.empty
    invar (empty ())
```

```
type-synonym ('V,'W,'G) graph-add-node = 'V  $\Rightarrow$  'G  $\Rightarrow$  'G
```

```
locale graph-add-node = graph +
  constrains  $\alpha$  :: 'G  $\Rightarrow$  ('V,'W) graph
  fixes add-node :: 'V  $\Rightarrow$  'G  $\Rightarrow$  'G
  assumes add-node-correct:
    invar g  $\Rightarrow$  invar (add-node v g)
    invar g  $\Rightarrow$   $\alpha$  (add-node v g) = Graph.add-node v ( $\alpha$  g)
```

```
type-synonym ('V,'W,'G) graph-delete-node = 'V  $\Rightarrow$  'G  $\Rightarrow$  'G
```

```
locale graph-delete-node = graph +
  constrains  $\alpha$  :: 'G  $\Rightarrow$  ('V,'W) graph
  fixes delete-node :: 'V  $\Rightarrow$  'G  $\Rightarrow$  'G
  assumes delete-node-correct:
    invar g  $\Rightarrow$  invar (delete-node v g)
    invar g  $\Rightarrow$   $\alpha$  (delete-node v g) = Graph.delete-node v ( $\alpha$  g)
```

```
type-synonym ('V,'W,'G) graph-add-edge = 'V  $\Rightarrow$  'W  $\Rightarrow$  'V  $\Rightarrow$  'G  $\Rightarrow$  'G
```

```
locale graph-add-edge = graph +
```

```

constrains  $\alpha :: 'G \Rightarrow ('V, 'W) \text{ graph}$ 
fixes  $\text{add-edge} :: 'V \Rightarrow 'W \Rightarrow 'V \Rightarrow 'G \Rightarrow 'G$ 
assumes  $\text{add-edge-correct}$ :
   $\text{invar } g \Longrightarrow \text{invar } (\text{add-edge } v \ e \ v' \ g)$ 
   $\text{invar } g \Longrightarrow \alpha (\text{add-edge } v \ e \ v' \ g) = \text{Graph.add-edge } v \ e \ v' \ (\alpha \ g)$ 

type-synonym  $('V, 'W, 'G) \text{ graph-delete-edge} = 'V \Rightarrow 'W \Rightarrow 'V \Rightarrow 'G \Rightarrow 'G$ 
locale  $\text{graph-delete-edge} = \text{graph} +$ 
constrains  $\alpha :: 'G \Rightarrow ('V, 'W) \text{ graph}$ 
fixes  $\text{delete-edge} :: 'V \Rightarrow 'W \Rightarrow 'V \Rightarrow 'G \Rightarrow 'G$ 
assumes  $\text{delete-edge-correct}$ :
   $\text{invar } g \Longrightarrow \text{invar } (\text{delete-edge } v \ e \ v' \ g)$ 
   $\text{invar } g \Longrightarrow \alpha (\text{delete-edge } v \ e \ v' \ g) = \text{Graph.delete-edge } v \ e \ v' \ (\alpha \ g)$ 

type-synonym  $('V, 'W, 's, 'G) \text{ graph-nodes-it} = 'G \Rightarrow ('V, 's) \text{ set-iterator}$ 

locale  $\text{graph-nodes-it-defs} =$ 
fixes  $\text{nodes-list-it} :: 'G \Rightarrow ('V, 'V \text{ list}) \text{ set-iterator}$ 
begin
definition  $\text{nodes-it } g \equiv \text{it-to-it } (\text{nodes-list-it } g)$ 
end

locale  $\text{graph-nodes-it} = \text{graph } \alpha \ \text{invar} + \text{graph-nodes-it-defs } \text{nodes-list-it}$ 
for  $\alpha :: 'G \Rightarrow ('V, 'W) \text{ graph}$  and  $\text{invar}$  and
 $\text{nodes-list-it} :: 'G \Rightarrow ('V, 'V \text{ list}) \text{ set-iterator}$ 
+
assumes  $\text{nodes-list-it-correct}$ :
   $\text{invar } g \Longrightarrow \text{set-iterator } (\text{nodes-list-it } g) (\text{Graph.nodes } (\alpha \ g))$ 
begin
lemma  $\text{nodes-it-correct}$ :
   $\text{invar } g \Longrightarrow \text{set-iterator } (\text{nodes-it } g) (\text{Graph.nodes } (\alpha \ g))$ 
   $\langle \text{proof} \rangle$ 

lemma  $\text{pi-nodes-it}[\text{icf-proper-iterator}I]$ :
   $\text{proper-it } (\text{nodes-it } S) (\text{nodes-it } S)$ 
   $\langle \text{proof} \rangle$ 

lemma  $\text{nodes-it-proper}[\text{proper-it}]$ :
   $\text{proper-it}' \ \text{nodes-it } \ \text{nodes-it}$ 
   $\langle \text{proof} \rangle$ 

end

type-synonym  $('V, 'W, 's, 'G) \text{ graph-edges-it}$ 
   $= 'G \Rightarrow (('V \times 'W \times 'V), 's) \text{ set-iterator}$ 

locale  $\text{graph-edges-it-defs} =$ 
fixes  $\text{edges-list-it} :: ('V, 'W, ('V \times 'W \times 'V) \text{ list}, 'G) \text{ graph-edges-it}$ 
begin

```

definition $edges-it\ g \equiv it-to-it\ (edges-list-it\ g)$
end

locale $graph-edges-it = graph\ \alpha\ invar + graph-edges-it-defs\ edges-list-it$
for $\alpha :: 'G \Rightarrow ('V, 'W)\ graph$ **and** $invar$ **and**
 $edges-list-it :: ('V, 'W, ('V \times 'W \times 'V)\ list, 'G)\ graph-edges-it$
 $+$
assumes $edges-list-it-correct$:
 $invar\ g \Longrightarrow set-iterator\ (edges-list-it\ g)\ (Graph.edges\ (\alpha\ g))$
begin
lemma $edges-it-correct$:
 $invar\ g \Longrightarrow set-iterator\ (edges-it\ g)\ (Graph.edges\ (\alpha\ g))$
 $\langle proof \rangle$

lemma $pi-edges-it[icf-proper-iteratorI]$:
 $proper-it\ (edges-it\ S)\ (edges-it\ S)$
 $\langle proof \rangle$

lemma $edges-it-proper[proper-it]$:
 $proper-it'\ edges-it\ edges-it$
 $\langle proof \rangle$

end

type-synonym $('V, 'W, 's, 'G)\ graph-succ-it =$
 $'G \Rightarrow 'V \Rightarrow ('W \times 'V, 's)\ set-iterator$

locale $graph-succ-it-defs =$
fixes $succ-list-it :: 'G \Rightarrow 'V \Rightarrow ('W \times 'V, ('W \times 'V)\ list)\ set-iterator$
begin
definition $succ-it\ g\ v \equiv it-to-it\ (succ-list-it\ g\ v)$
end

locale $graph-succ-it = graph\ \alpha\ invar + graph-succ-it-defs\ succ-list-it$
for $\alpha :: 'G \Rightarrow ('V, 'W)\ graph$ **and** $invar$ **and**
 $succ-list-it :: 'G \Rightarrow 'V \Rightarrow ('W \times 'V, ('W \times 'V)\ list)\ set-iterator +$
assumes $succ-list-it-correct$:
 $invar\ g \Longrightarrow set-iterator\ (succ-list-it\ g\ v)\ (Graph.succ\ (\alpha\ g)\ v)$
begin
lemma $succ-it-correct$:
 $invar\ g \Longrightarrow set-iterator\ (succ-it\ g\ v)\ (Graph.succ\ (\alpha\ g)\ v)$
 $\langle proof \rangle$

lemma $pi-succ-it[icf-proper-iteratorI]$:
 $proper-it\ (succ-it\ S\ v)\ (succ-it\ S\ v)$
 $\langle proof \rangle$

lemma $succ-it-proper[proper-it]$:
 $proper-it'\ (\lambda S. succ-it\ S\ v)\ (\lambda S. succ-it\ S\ v)$

<proof>

end

6.1 Adjacency Lists

type-synonym ('V,'W) *adj-list* = 'V list × ('V × 'W × 'V) list

definition *adjl-α* :: ('V,'W) *adj-list* ⇒ ('V,'W) *graph* **where**

adjl-α l ≡ *let* (nl,el) = l *in* {
 nodes = set nl ∪ fst'set el ∪ snd'snd'set el,
 edges = set el
}

lemma *adjl-is-graph*: *graph adjl-α* (λ-. True)

<proof>

type-synonym ('V,'W,'G) *graph-from-list* = ('V,'W) *adj-list* ⇒ 'G

locale *graph-from-list* = *graph* +

constrains α :: 'G ⇒ ('V,'W) *graph*

fixes *from-list* :: ('V,'W) *adj-list* ⇒ 'G

assumes *from-list-correct*:

invar (*from-list* l)

α (*from-list* l) = *adjl-α* l

type-synonym ('V,'W,'G) *graph-to-list* = 'G ⇒ ('V,'W) *adj-list*

locale *graph-to-list* = *graph* +

constrains α :: 'G ⇒ ('V,'W) *graph*

fixes *to-list* :: 'G ⇒ ('V,'W) *adj-list*

assumes *to-list-correct*:

invar g ⇒ *adjl-α* (*to-list* g) = α g

6.2 Record Based Interface

record ('V,'W,'G) *graph-ops* =

gop-α :: ('V,'W,'G) *graph-α*

gop-invar :: 'G ⇒ bool

gop-empty :: ('V,'W,'G) *graph-empty*

gop-add-node :: ('V,'W,'G) *graph-add-node*

gop-delete-node :: ('V,'W,'G) *graph-delete-node*

gop-add-edge :: ('V,'W,'G) *graph-add-edge*

gop-delete-edge :: ('V,'W,'G) *graph-delete-edge*

gop-from-list :: ('V,'W,'G) *graph-from-list*

gop-to-list :: ('V,'W,'G) *graph-to-list*

gop-nodes-list-it :: 'G ⇒ ('V,'V list) *set-iterator*

gop-edges-list-it :: ('V,'W,('V × 'W × 'V) list,'G) *graph-edges-it*

gop-succ-list-it :: 'G ⇒ 'V ⇒ ('W × 'V,('W × 'V) list) *set-iterator*

locale *StdGraphDefs* =

```

    graph-nodes-it-defs gop-nodes-list-it ops
  + graph-edges-it-defs gop-edges-list-it ops
  + graph-succ-it-defs gop-succ-list-it ops
  for ops :: ('V,'W,'G,'m) graph-ops-scheme
begin
  abbreviation  $\alpha$  where  $\alpha \equiv gop-\alpha$  ops
  abbreviation invar where invar  $\equiv gop-invar$  ops
  abbreviation empty where empty  $\equiv gop-empty$  ops
  abbreviation add-node where add-node  $\equiv gop-add-node$  ops
  abbreviation delete-node where delete-node  $\equiv gop-delete-node$  ops
  abbreviation add-edge where add-edge  $\equiv gop-add-edge$  ops
  abbreviation delete-edge where delete-edge  $\equiv gop-delete-edge$  ops
  abbreviation from-list where from-list  $\equiv gop-from-list$  ops
  abbreviation to-list where to-list  $\equiv gop-to-list$  ops
  abbreviation nodes-list-it where nodes-list-it  $\equiv gop-nodes-list-it$  ops
  abbreviation edges-list-it where edges-list-it  $\equiv gop-edges-list-it$  ops
  abbreviation succ-list-it where succ-list-it  $\equiv gop-succ-list-it$  ops
end

locale StdGraph = StdGraphDefs +
  graph  $\alpha$  invar +
  graph-empty  $\alpha$  invar empty +
  graph-add-node  $\alpha$  invar add-node +
  graph-delete-node  $\alpha$  invar delete-node +
  graph-add-edge  $\alpha$  invar add-edge +
  graph-delete-edge  $\alpha$  invar delete-edge +
  graph-from-list  $\alpha$  invar from-list +
  graph-to-list  $\alpha$  invar to-list +
  graph-nodes-it  $\alpha$  invar nodes-list-it +
  graph-edges-it  $\alpha$  invar edges-list-it +
  graph-succ-it  $\alpha$  invar succ-list-it
begin
  lemmas correct = empty-correct add-node-correct delete-node-correct
    add-edge-correct delete-edge-correct
    from-list-correct to-list-correct
end

```

6.3 Refinement Framework Bindings

```

lemma (in graph-nodes-it) nodes-it-is-iterator[refine-transfer]:
  invar  $g \implies set-iterator (nodes-it\ g) (nodes\ (\alpha\ g))$ 
  <proof>

```

```

lemma (in graph-edges-it) edges-it-is-iterator[refine-transfer]:
  invar  $g \implies set-iterator (edges-it\ g) (edges\ (\alpha\ g))$ 
  <proof>

```

```

lemma (in graph-succ-it) succ-it-is-iterator[refine-transfer]:

```

invar g \implies *set-iterator (succ-it g v) (Graph.succ (α g) v)*
 <proof>

lemma (in *graph*) *drh[refine-dref-RELATES]: RELATES (build-rel α invar)*
 <proof>

end

7 Generic Algorithms for Graphs

theory *GraphGA*

imports

GraphSpec

begin

definition *gga-from-list* ::
 (*'V, 'W, 'G*) *graph-empty* \Rightarrow (*'V, 'W, 'G*) *graph-add-node*
 \Rightarrow (*'V, 'W, 'G*) *graph-add-edge*
 \Rightarrow (*'V, 'W, 'G*) *graph-from-list*

where

gga-from-list e a u l \equiv
 let (*nl, el*) = *l*;
g1 = *foldl* ($\lambda g v. a v g$) (*e* ()) *nl*
 in *foldl* ($\lambda g (v, e, v'). u v e v' g$) *g1 el*

lemma *gga-from-list-correct*:
fixes $\alpha :: 'G \Rightarrow ('V, 'W)$ *graph*
assumes *graph-empty* α *invar e*
assumes *graph-add-node* α *invar a*
assumes *graph-add-edge* α *invar u*
shows *graph-from-list* α *invar (gga-from-list e a u)*
 <proof>

term *map-iterator-product*

locale *gga-edges-it-defs* =
graph-nodes-it-defs nodes-list-it +
graph-succ-it-defs succ-list-it
for *nodes-list-it* :: (*'V, 'W, 'V list, 'G*) *graph-nodes-it*
and *succ-list-it* :: (*'V, 'W, ('W \times 'V) list, 'G*) *graph-succ-it*
begin

definition *gga-edges-list-it* ::
 (*'V, 'W, ('V \times 'W \times 'V) list, 'G*) *graph-edges-it*
where *gga-edges-list-it G* \equiv *set-iterator-product*
 (*nodes-it G*) (*succ-it G*)
 <ML>


```

end
⟨ML⟩

locale gga-edges-it = gga-edges-it-defs nodes-list-it succ-list-it
  + graph  $\alpha$  invar
  + graph-nodes-it  $\alpha$  invar nodes-list-it
  + graph-succ-it  $\alpha$  invar succ-list-it
for  $\alpha :: 'G \Rightarrow ('V, 'W)$  graph
and invar
and nodes-list-it ::  $('V, 'W, 'V \text{ list}, 'G)$  graph-nodes-it
and succ-list-it ::  $('V, 'W, ('W \times 'V) \text{ list}, 'G)$  graph-succ-it
begin
  lemma gga-edges-list-it-impl:
    shows graph-edges-it  $\alpha$  invar gga-edges-list-it
    ⟨proof⟩
end

locale gga-to-list-defs-loc =
  graph-nodes-it-defs nodes-list-it
  + graph-edges-it-defs edges-list-it
for nodes-list-it ::  $('V, 'W, 'V \text{ list}, 'G)$  graph-nodes-it
and edges-list-it ::  $('V, 'W, ('V \times 'W \times 'V) \text{ list}, 'G)$  graph-edges-it
begin
  definition gga-to-list ::
     $('V, 'W, 'G)$  graph-to-list
  where
    gga-to-list  $g \equiv$ 
      (nodes-it  $g$  ( $\lambda$ -. True) (#) [], edges-it  $g$  ( $\lambda$ -. True) (#) [])
end

locale gga-to-list-loc = gga-to-list-defs-loc nodes-list-it edges-list-it +
  graph  $\alpha$  invar
  + graph-nodes-it  $\alpha$  invar nodes-list-it
  + graph-edges-it  $\alpha$  invar edges-list-it
for  $\alpha :: 'G \Rightarrow ('V, 'W)$  graph and invar
and nodes-list-it ::  $('V, 'W, 'V \text{ list}, 'G)$  graph-nodes-it
and edges-list-it ::  $('V, 'W, ('V \times 'W \times 'V) \text{ list}, 'G)$  graph-edges-it
begin

  lemma gga-to-list-correct:
    shows graph-to-list  $\alpha$  invar gga-to-list
    ⟨proof⟩

end

end

```

8 Implementing Graphs by Maps

theory *GraphByMap*

imports

GraphSpec

GraphGA

begin

definition *map-Sigma* $M1\ F2 \equiv \{$
 $(x,y). \exists v. M1\ x = \text{Some } v \wedge y \in F2\ v$
 $\}$

lemma *map-Sigma-alt*: $\text{map-Sigma } M1\ F2 = \text{Sigma } (\text{dom } M1) (\lambda x.$
 $F2\ (\text{the } (M1\ x)))$
 $\langle \text{proof} \rangle$

lemma *ranE*:

assumes $v \in \text{ran } m$

obtains k **where** $m\ k = \text{Some } v$

$\langle \text{proof} \rangle$

lemma *option-bind-alt*:

$\text{Option.bind } x\ f = (\text{case } x\ \text{of } \text{None} \Rightarrow \text{None} \mid \text{Some } v \Rightarrow f\ v)$

$\langle \text{proof} \rangle$

locale *GraphByMapDefs* =

$m1: \text{StdMapDefs } m1\text{-ops} +$

$m2: \text{StdMapDefs } m2\text{-ops} +$

$s3: \text{StdSetDefs } s3\text{-ops}$

for $m1\text{-ops}::('V, 'm2, 'm1, -)$ *map-ops-scheme*

and $m2\text{-ops}::('V, 's3, 'm2, -)$ *map-ops-scheme*

and $s3\text{-ops}::('W, 's3, -)$ *set-ops-scheme*

and $m1\text{-mvif} :: ('V \Rightarrow 'm2 \rightarrow 'm2) \Rightarrow 'm1 \Rightarrow 'm1$

begin

definition *gbm- α* $:: ('V, 'W, 'm1)$ *graph- α* **where**

$\text{gbm-}\alpha\ m1 \equiv$

$(\mid \text{nodes} = \text{dom } (m1.\alpha\ m1),$

$\text{edges} = \{(v,w,v').$

$\exists m2\ s3. m1.\alpha\ m1\ v = \text{Some } m2$

$\wedge m2.\alpha\ m2\ v' = \text{Some } s3$

$\wedge w \in s3.\alpha\ s3$

$\}$

$\mid)$

definition *gbm-invar* $m1 \equiv$

$m1.\text{invar } m1 \wedge$

$(\forall m2 \in \text{ran } (m1.\alpha\ m1). m2.\text{invar } m2 \wedge$

$(\forall s3 \in \text{ran } (m2.\alpha\ m2). s3.\text{invar } s3)$

$) \wedge \text{valid-graph } (\text{gbm-}\alpha\ m1)$

definition *gbm-empty* :: ('V,'W,'m1) *graph-empty* **where**
gbm-empty \equiv *m1.empty*

definition *gbm-add-node* :: ('V,'W,'m1) *graph-add-node* **where**
gbm-add-node *v g* \equiv *case m1.lookup v g of*
None \Rightarrow *m1.update v (m2.empty ()) g* |
Some - \Rightarrow *g*

definition *gbm-delete-node* :: ('V,'W,'m1) *graph-delete-node* **where**
gbm-delete-node *v g* \equiv *let g=m1.delete v g in*
m1-mvif (λ - *m2. Some (m2.delete v m2)*) *g*

definition *gbm-add-edge* :: ('V,'W,'m1) *graph-add-edge* **where**
gbm-add-edge *v e v' g* \equiv
let g = (case m1.lookup v' g of
None \Rightarrow *m1.update v' (m2.empty ()) g* | *Some -* \Rightarrow *g*
) in
case m1.lookup v g of
None \Rightarrow (*m1.update v (m2.sng v' (s3.sng e)) g*) |
Some m2 \Rightarrow (*case m2.lookup v' m2 of*
None \Rightarrow *m1.update v (m2.update v' (s3.sng e) m2) g* |
Some s3 \Rightarrow *m1.update v (m2.update v' (s3.ins e s3) m2) g*)

definition *gbm-delete-edge* :: ('V,'W,'m1) *graph-delete-edge* **where**
gbm-delete-edge *v e v' g* \equiv
case m1.lookup v g of
None \Rightarrow *g* |
Some m2 \Rightarrow (
case m2.lookup v' m2 of
None \Rightarrow *g* |
Some s3 \Rightarrow *m1.update v (m2.update v' (s3.delete e s3) m2) g*
)

definition *gbm-nodes-list-it*
:: ('V,'W,'V list,'m1) *graph-nodes-it*
where
gbm-nodes-list-it *g* \equiv *map-iterator-dom (m1.iteratei g)*
<ML>

definition *gbm-edges-list-it*
:: ('V,'W,('V \times 'W \times 'V) list,'m1) *graph-edges-it*
where
gbm-edges-list-it *g* \equiv *set-iterator-image*
($\lambda((v1,m1),(v2,m2),w). (v1,w,v2)$)
(*set-iterator-product* (*m1.iteratei* *g*)
($\lambda(v,m2). \textit{set-iterator-product}$))

$(m2.iteratei\ m2)\ (\lambda(w,s3).\ s3.iteratei\ s3))$

$\langle ML \rangle$

definition *gbm-succ-list-it* ::

$(\ 'V, 'W, ('W \times 'V)\ list, 'm1)\ graph-succ-it$

where

gbm-succ-list-it $g\ v \equiv$ case *m1.lookup* $v\ g$ of

None \Rightarrow *set-iterator-emp* |

Some $m2 \Rightarrow$

set-iterator-image $(\lambda((v',m2),w).\ (w,v'))$

$(\text{set-iterator-product } (m2.iteratei\ m2)\ (\lambda(v',s).\ s3.iteratei\ s))$

$\langle ML \rangle$

definition

gbm-from-list \equiv *gga-from-list* *gbm-empty* *gbm-add-node* *gbm-add-edge*

lemma *gbm-nodes-list-it-unf*:

it-to-it (*gbm-nodes-list-it* g)

\equiv *map-iterator-dom* (*it-to-it* (*m1.list-it* g))

$\langle proof \rangle$

lemma *gbm-edges-list-it-unf*:

it-to-it (*gbm-edges-list-it* g)

\equiv *set-iterator-image*

$(\lambda((v1,m1),(v2,m2),w).\ (v1,w,v2))$

$(\text{set-iterator-product } (\text{it-to-it } (m1.list-it\ g))$

$(\lambda(v,m2).\ \text{set-iterator-product}$

$(\text{it-to-it } (m2.list-it\ m2))\ (\lambda(w,s3).\ (\text{it-to-it } (s3.list-it\ s3))))))$

$\langle proof \rangle$

lemma *gbm-succ-list-it-unf*:

it-to-it (*gbm-succ-list-it* $g\ v$) \equiv

case *m1.lookup* $v\ g$ of

None \Rightarrow *set-iterator-emp* |

Some $m2 \Rightarrow$

set-iterator-image $(\lambda((v',m2),w).\ (w,v'))$

$(\text{set-iterator-product } (\text{it-to-it } (m2.list-it\ m2))$

$(\lambda(v',s).\ (\text{it-to-it } (s3.list-it\ s))))$

$\langle proof \rangle$

end

sublocale *GraphByMapDefs* $<$ *graph-nodes-it-defs* *gbm-nodes-list-it* $\langle proof \rangle$

```

sublocale GraphByMapDefs < graph-edges-it-defs gbm-edges-list-it <proof>
sublocale GraphByMapDefs < graph-succ-it-defs gbm-succ-list-it <proof>
sublocale GraphByMapDefs
  < gga-to-list-defs-loc gbm-nodes-list-it gbm-edges-list-it <proof>

```

```

context GraphByMapDefs
begin

```

```

definition [icf-rec-def]: gbm-ops ≡ (
  gop-α = gbm-α,
  gop-invar = gbm-invar,
  gop-empty = gbm-empty,
  gop-add-node = gbm-add-node,
  gop-delete-node = gbm-delete-node,
  gop-add-edge = gbm-add-edge,
  gop-delete-edge = gbm-delete-edge,
  gop-from-list = gbm-from-list,
  gop-to-list = gga-to-list,
  gop-nodes-list-it = gbm-nodes-list-it,
  gop-edges-list-it = gbm-edges-list-it,
  gop-succ-list-it = gbm-succ-list-it
)

```

```

  )
  <ML>

```

```

end

```

```

locale GraphByMap = GraphByMapDefs m1-ops m2-ops s3-ops m1-mvif +
  m1: StdMap m1-ops +
  m2: StdMap m2-ops +
  s3: StdSet s3-ops +
  m1: map-value-image-filter m1.α m1.invar m1.α m1.invar m1-mvif
for m1-ops::('V,'m2,'m1,-) map-ops-scheme
and m2-ops::('V,'s3,'m2,-) map-ops-scheme
and s3-ops::('W,'s3,-) set-ops-scheme
and m1-mvif :: ('V ⇒ 'm2 → 'm2) ⇒ 'm1 ⇒ 'm1

```

```

begin

```

```

lemma gbm-invar-split:

```

```

  assumes gbm-invar g

```

```

  shows

```

```

    m1.invar g

```

```

    ∧ v m2. m1.α g v = Some m2 ⇒ m2.invar m2

```

```

    ∧ v m2 v' s3. m1.α g v = Some m2 ⇒ m2.α m2 v' = Some s3 ⇒ s3.invar

```

```

  s3

```

```

    valid-graph (gbm-α g)

```

```

    <proof>

```

```

end

```

```

sublocale GraphByMap < graph gbm-α gbm-invar
  <proof>

```

context *GraphByMap*

begin

lemma *gbm-empty-impl:*

graph-empty gbm- α gbm-invar gbm-empty
<proof>

lemma *gbm-add-node-impl:*

graph-add-node gbm- α gbm-invar gbm-add-node
<proof>

lemma *gbm-delete-node-impl:*

graph-delete-node gbm- α gbm-invar gbm-delete-node
<proof>

lemma *gbm-add-edge-impl:*

graph-add-edge gbm- α gbm-invar gbm-add-edge
<proof>

lemma *gbm-delete-edge-impl:*

graph-delete-edge gbm- α gbm-invar gbm-delete-edge
<proof>

lemma *gbm-nodes-list-it-impl:*

shows *graph-nodes-it gbm- α gbm-invar gbm-nodes-list-it*
<proof>

lemma *gbm-edges-list-it-impl:*

shows *graph-edges-it gbm- α gbm-invar gbm-edges-list-it*
<proof>

lemma *gbm-succ-list-it-impl:*

shows *graph-succ-it gbm- α gbm-invar gbm-succ-list-it*
<proof>

lemma *gbm-from-list-impl:*

shows *graph-from-list gbm- α gbm-invar gbm-from-list*
<proof>

end

sublocale *GraphByMap* < *graph-nodes-it gbm- α gbm-invar gbm-nodes-list-it*
<proof>

sublocale *GraphByMap* < *graph-edges-it gbm- α gbm-invar gbm-edges-list-it*
<proof>

sublocale *GraphByMap* < *graph-succ-it gbm- α gbm-invar gbm-succ-list-it*
<proof>

sublocale *GraphByMap*
 < *gga-to-list-loc gbm- α gbm-invar gbm-nodes-list-it gbm-edges-list-it*
 <*proof*>

context *GraphByMap*

begin

lemma *gbm-to-list-impl: graph-to-list gbm- α gbm-invar gga-to-list*
 <*proof*>

lemma *gbm-ops-impl: StdGraph gbm-ops*
 <*proof*>

end

<*ML*>

end

9 Graphs by Hashmaps

theory *HashGraphImpl*

imports

GraphByMap

begin

Abbreviation: *hlg*

type-synonym (*'V, 'E*) *hlg* =
 (*'V, ('V, 'E* *ls*) *HashMap.hashmap*) *HashMap.hashmap*

<*ML*>

interpretation *hh-mvif: g-value-image-filter-loc hm-ops hm-ops*
 <*proof*>

interpretation *hlg-gbm: GraphByMap hm-ops hm-ops ls-ops*
hh-mvif.g-value-image-filter

<*proof*>

<*ML*>

definition [*icf-rec-def*]: *hlg-ops* \equiv *hlg-gbm.gbm-ops*

<*ML*>

interpretation *hlg: StdGraph hlg-ops*

<*proof*>

<*ML*>

thm *map-iterator-dom-def set-iterator-image-def*
set-iterator-image-filter-def

definition *test-codegen* **where** *test-codegen* \equiv (
hlg.empty,

```

    hlg.add-node,
    hlg.delete-node,
    hlg.add-edge,
    hlg.delete-edge,
    hlg.from-list,
    hlg.to-list,
    hlg.nodes-it,
    hlg.edges-it,
    hlg.succ-it
  )

export-code test-codegen in SML

end

```

10 Implementation of Dijkstra's-Algorithm using the ICF

```

theory Dijkstra-Impl
imports
  Dijkstra
  GraphSpec
  HashGraphImpl
  HOL-Library.Code-Target-Numeral
begin

```

In this second refinement step, we use interfaces from the Isabelle Collection Framework (ICF) to implement the priority queue and the result map. Moreover, we use a graph interface (that is not contained in the ICF, but in this development) to represent the graph.

The data types of the first refinement step were designed to fit the abstract data types of the used ICF-interfaces, which makes this refinement quite straightforward.

Finally, we instantiate the ICF-interfaces by concrete implementations, obtaining an executable algorithm, for that we generate code using Isabelle/HOL's code generator.

```

locale dijkstraC =
  g: StdGraph g-ops +
  mr: StdMap mr-ops +
  qw: StdUprio qw-ops
  for g-ops :: ('V,'W::weight,'G,'moreg) graph-ops-scheme
  and mr-ops :: ('V, (('V,'W) path × 'W), 'mr,'more-mr) map-ops-scheme
  and qw-ops :: ('V,'W infty,'qw,'more-qw) uprio-ops-scheme
begin
  definition αsc == map-prod qw.α mr.α
  definition dinvarC-add == λ(wl,res). qw.invar wl ∧ mr.invar res

```



```

definition cdinit :: 'G ⇒ 'V ⇒ ('qw×'mr) nres where
  cdinit g v0 ≡ do {
    wl ← FOREACH (nodes (g.α g))
      (λv wl. RETURN (qw.insert wl v Weight.Infty)) (qw.empty ());
    RETURN (qw.insert wl v0 (Num 0),mr.sng v0 ([],0))
  }

definition cpop-min :: ('qw×'mr) ⇒ ('V×'W infty×('qw×'mr)) nres where
  cpop-min σ ≡ do {
    let (wl,res) = σ;
    let (v,w,wl')=qw.pop wl;
    RETURN (v,w,(wl',res))
  }

definition cupdate :: 'G ⇒ 'V ⇒ 'W infty ⇒ ('qw×'mr) ⇒ ('qw×'mr) nres
where
  cupdate g v ww σ = do {
    ASSERT (dinvarC-add σ);
    let (wl,res)=σ;
    let pv=mpath' (mr.lookup v res);
    FOREACH (succ (g.α g) v) (λ(w',v') (wl,res).
      if (ww + Num w' < mpath-weight' (mr.lookup v' res)) then do {
        RETURN (qw.insert wl v' (ww+Num w'),
          mr.update v' ((v,w',v')#the pv,val ww + w') res)
      } else RETURN (wl,res)
    ) (wl,res)
  }

definition cdijkstra where
  cdijkstra g v0 ≡ do {
    σ0 ← cdinit g v0;
    (-,res) ← WHILE_T (λ(wl,-). ¬ qw.isEmpty wl)
      (λσ. do { (v,ww,σ') ← cpop-min σ; cupdate g v ww σ' } )
    σ0;
    RETURN res
  }

end

locale dijkstraC-fixg = dijkstraC g-ops mr-ops qw-ops +
  Dijkstra ga v0
for g-ops :: ('V,'W::weight,'G,'moreg) graph-ops-scheme
and mr-ops :: ('V, (('V,'W) path × 'W), 'mr,'more-mr) map-ops-scheme
and qw-ops :: ('V,'W infty,'qw,'more-qw) uprio-ops-scheme
and ga :: ('V,'W) graph
and v0 :: 'V +
fixes g :: 'G
assumes g-rel: (g,ga)∈br g.α g.invar
begin

```

```

schematic-goal cdinit-refines:
  notes [refine] = inj-on-id
  shows cdinit g v0  $\leq \Downarrow ?R$  mdinit
   $\langle \textit{proof} \rangle$ 

schematic-goal cpop-min-refines:
   $(\sigma, \sigma') \in \textit{build-rel } \alpha \textit{sc } \textit{dinvarC-add}$ 
   $\implies \textit{cpop-min } \sigma \leq \Downarrow ?R (\textit{mpop-min } \sigma')$ 
   $\langle \textit{proof} \rangle$ 

schematic-goal cupdate-refines:
  notes [refine] = inj-on-id
  shows  $(\sigma, \sigma') \in \textit{build-rel } \alpha \textit{sc } \textit{dinvarC-add} \implies v = v' \implies wv = wv' \implies$ 
   $\textit{cupdate } g v wv \sigma \leq \Downarrow ?R (\textit{mupdate } v' wv' \sigma')$ 
   $\langle \textit{proof} \rangle$ 

lemma dijkstra-refines:
   $\textit{dijkstra } g v0 \leq \Downarrow (\textit{build-rel } \textit{mr} . \alpha \textit{mr} . \textit{invar}) \textit{mdijkstra}$ 
   $\langle \textit{proof} \rangle$ 
end

context dijkstraC
begin

  thm g.nodes-it-is-iterator

  schematic-goal idijkstra-refines-aux:
    assumes g.invar g
    shows RETURN ?f  $\leq \textit{cdijkstra } g v0$ 
     $\langle \textit{proof} \rangle$ 

  concrete-definition idijkstra for g ?v0.0 uses idijkstra-refines-aux

  lemma idijkstra-refines:
    assumes g.invar g
    shows RETURN (idijkstra g v0)  $\leq \textit{cdijkstra } g v0$ 
     $\langle \textit{proof} \rangle$ 

end

```

The following theorem states correctness of the algorithm independent from the refinement framework.

Intuitively, the first goal states that the abstraction of the returned result is correct, the second goal states that the result datastructure satisfies its invariant, and the third goal states that the cached weights in the returned result are correct.

Note that this is the main theorem for a user of Dijkstra's algorithm in some bigger context. It may also be specialized for concrete instances of the

implementation, as exemplarily done below.

```
theorem (in dijkstraC-fixg) idijkstra-correct:  
  shows  
    weighted-graph.is-shortest-path-map ga v0 ( $\alpha r$  (mr.alpha (idijkstra g v0)))  
    (is ?G1)  
  and mr.invar (idijkstra g v0) (is ?G2)  
  and Dijkstra.res-invarm (mr.alpha (idijkstra g v0)) (is ?G3)  
<proof>
```

```
theorem (in dijkstraC) idijkstra-correct:  
  assumes INV: g.invar g  
  assumes V0: v0  $\in$  nodes (g.alpha g)  
  assumes nonneg-weights:  $\bigwedge v w v'. (v,w,v') \in \text{edges } (g.alpha g) \implies 0 \leq w$   
  shows  
    weighted-graph.is-shortest-path-map (g.alpha g) v0  
    (Dijkstra.alpha (mr.alpha (idijkstra g v0))) (is ?G1)  
  and Dijkstra.res-invarm (mr.alpha (idijkstra g v0)) (is ?G2)  
<proof>
```

Example instantiation with HashSet-based graph, red-black-tree based result map, and finger-tree based priority queue.

```
<ML>  
interpretation hrf: dijkstraC hlg-ops rm-ops aluprioi-ops  
  <proof>  
<ML>
```

```
definition hrf-dijkstra  $\equiv$  hrf.idijkstra  
lemmas hrf-dijkstra-correct = hrf.idijkstra-correct[folded hrf-dijkstra-def]
```

```
export-code hrf-dijkstra checking SML  
export-code hrf-dijkstra in OCaml  
export-code hrf-dijkstra in Haskell  
export-code hrf-dijkstra checking Scala
```

```
definition hrfn-dijkstra :: (nat,nat) hlg  $\Rightarrow$  -  
  where hrfn-dijkstra  $\equiv$  hrf-dijkstra
```

```
export-code hrfn-dijkstra in SML
```

```
lemmas hrfn-dijkstra-correct =  
  hrf-dijkstra-correct[where ?'a = nat and ?'b = nat, folded hrfn-dijkstra-def]
```

```
term hrfn-dijkstra  
term hlg.from-list
```

```
definition test-hrfn-dijkstra
```

\equiv *rm.to-list*
(*hrfn-dijkstra* (*hlg.from-list* ($[0..<4]$, $[(0,3,1),(0,4,2),(2,1,3),(1,4,3)]$))) 0)

$\langle ML \rangle$

end

11 Implementation of Dijkstra's-Algorithm using Automatic Determinization

theory *Dijkstra-Impl-Adet*
imports
Dijkstra
GraphSpec
HashGraphImpl
Collections.Refine-Dft-ICF
HOL-Library.Code-Target-Numeral
begin

11.1 Setup

11.1.1 Infinity

definition *infty-rel-internal-def*:

$infty\text{-rel } R \equiv \{(Num\ a, Num\ a') \mid a\ a'.\ (a, a') \in R\} \cup \{(Infty, Infty)\}$

lemma *infty-rel-def[refine-rel-defs]*:

$\langle R \rangle infty\text{-rel} = \{(Num\ a, Num\ a') \mid a\ a'.\ (a, a') \in R\} \cup \{(Infty, Infty)\}$
 $\langle proof \rangle$

lemma *infty-relI*:

$(Infty, Infty) \in \langle R \rangle infty\text{-rel}$
 $(a, a') \in R \implies (Num\ a, Num\ a') \in \langle R \rangle infty\text{-rel}$
 $\langle proof \rangle$

lemma *infty-relE*:

assumes $(x, x') \in \langle R \rangle infty\text{-rel}$
obtains $x = Infty$ **and** $x' = Infty$
 $\mid a\ a'$ **where** $x = Num\ a$ **and** $x' = Num\ a'$ **and** $(a, a') \in R$
 $\langle proof \rangle$

lemma *infty-rel-simps[simp]*:

$(Infty, x') \in \langle R \rangle infty\text{-rel} \longleftrightarrow x' = Infty$
 $(x, Infty) \in \langle R \rangle infty\text{-rel} \longleftrightarrow x = Infty$
 $(Num\ a, Num\ a') \in \langle R \rangle infty\text{-rel} \longleftrightarrow (a, a') \in R$
 $\langle proof \rangle$

lemma *infty-rel-sv[relator-props]*:

$single\text{-valued } R \implies single\text{-valued } (\langle R \rangle infty\text{-rel})$
 $\langle proof \rangle$

lemma *infty-rel-id*[*simp, relator-props*]: $\langle Id \rangle \text{infty-rel} = Id$
 $\langle \text{proof} \rangle$

consts *i-infty* :: *interface* \Rightarrow *interface*

lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of infty-rel i-infty*]

lemma *autoref-infty*[*param, autoref-rules*]:

$(Infty, Infty) \in \langle R \rangle \text{infty-rel}$
 $(Num, Num) \in R \rightarrow \langle R \rangle \text{infty-rel}$
 $(\text{case-infty}, \text{case-infty}) \in Rr \rightarrow (R \rightarrow Rr) \rightarrow \langle R \rangle \text{infty-rel} \rightarrow Rr$
 $(\text{rec-infty}, \text{rec-infty}) \in Rr \rightarrow (R \rightarrow Rr) \rightarrow \langle R \rangle \text{infty-rel} \rightarrow Rr$
 $\langle \text{proof} \rangle$

definition [*simp*]: *is-Infty* $x \equiv \text{case } x \text{ of } Infty \Rightarrow True \mid - \Rightarrow False$

context begin interpretation *autoref-syn* $\langle \text{proof} \rangle$

lemma *pat-is-Infty*[*autoref-op-pat*]:

$x = Infty \equiv (OP \text{ is-Infty} \ ::: \langle I \rangle_i \text{ i-infty} \rightarrow_i \text{ i-bool}) \$ x$
 $Infty = x \equiv (OP \text{ is-Infty} \ ::: \langle I \rangle_i \text{ i-infty} \rightarrow_i \text{ i-bool}) \$ x$
 $\langle \text{proof} \rangle$

end

lemma *autoref-is-Infty*[*autoref-rules*]:

$(\text{is-Infty}, \text{is-Infty}) \in \langle R \rangle \text{infty-rel} \rightarrow \text{bool-rel}$
 $\langle \text{proof} \rangle$

definition *infty-eq* $eq \ v1 \ v2 \equiv$

case (*v1, v2*) *of*
 $(Infty, Infty) \Rightarrow True$
 $\mid (Num \ a1, Num \ a2) \Rightarrow eq \ a1 \ a2$
 $\mid - \Rightarrow False$

lemma *infty-eq-autoref*[*autoref-rules (overloaded)*]:

$\llbracket GEN-OP \ eq \ (=) \ (R \rightarrow R \rightarrow \text{bool-rel}) \rrbracket$
 $\Longrightarrow (\text{infty-eq} \ eq, (=)) \in \langle R \rangle \text{infty-rel} \rightarrow \langle R \rangle \text{infty-rel} \rightarrow \text{bool-rel}$
 $\langle \text{proof} \rangle$

lemma *infty-eq-expand*[*autoref-struct-expand*]: $(=) = \text{infty-eq} \ (=)$

$\langle \text{proof} \rangle$

context begin interpretation *autoref-syn* $\langle \text{proof} \rangle$

lemma *infty-val-autoref*[*autoref-rules*]:

$\llbracket SIDE-PRECOND \ (x \neq Infty); \ (xi, x) \in \langle R \rangle \text{infty-rel} \rrbracket$
 $\Longrightarrow (\text{val } xi, (OP \ \text{val} \ ::: \langle R \rangle \text{infty-rel} \rightarrow R) \ \$ \ x) \in R$
 $\langle \text{proof} \rangle$

end

definition *infty-plus* **where**

$\text{infty-plus pl } a \ b \equiv \text{case } (a,b) \text{ of } (\text{Num } a, \text{Num } b) \Rightarrow \text{Num } (\text{pl } a \ b) \mid - \Rightarrow \text{Infty}$

lemma $\text{infty-plus-param}[param]$:

$(\text{infty-plus}, \text{infty-plus}) \in (R \rightarrow R \rightarrow R) \rightarrow \langle R \rangle \text{infty-rel} \rightarrow \langle R \rangle \text{infty-rel} \rightarrow \langle R \rangle \text{infty-rel}$
 $\langle \text{proof} \rangle$

lemma $\text{infty-plus-eq-plus}$: $\text{infty-plus } (+) = (+)$

$\langle \text{proof} \rangle$

lemma $\text{infty-plus-autoref}[autoref-rules]$:

$GEN-OP \text{ pl } (+) (R \rightarrow R \rightarrow R)$

$\Rightarrow (\text{infty-plus pl}, (+)) \in \langle R \rangle \text{infty-rel} \rightarrow \langle R \rangle \text{infty-rel} \rightarrow \langle R \rangle \text{infty-rel}$
 $\langle \text{proof} \rangle$

11.1.2 Graph

consts $i\text{-graph} :: \text{interface} \Rightarrow \text{interface} \Rightarrow \text{interface}$

definition $\text{graph-more-rel-internal-def}$:

$\text{graph-more-rel } Rm \ Rv \ Rw \equiv \{ (g, g') \}$

$(\text{graph.nodes } g, \text{graph.nodes } g') \in \langle Rv \rangle \text{set-rel}$

$\wedge (\text{graph.edges } g, \text{graph.edges } g') \in \langle \langle Rv, \langle Rw, Rv \rangle \text{prod-rel} \rangle \text{prod-rel} \rangle \text{set-rel}$

$\wedge (\text{graph.more } g, \text{graph.more } g') \in Rm \}$

lemma $\text{graph-more-rel-def}[refine-rel-defs]$:

$\langle Rm, Rv, Rw \rangle \text{graph-more-rel} \equiv \{ (g, g') \}$

$(\text{graph.nodes } g, \text{graph.nodes } g') \in \langle Rv \rangle \text{set-rel}$

$\wedge (\text{graph.edges } g, \text{graph.edges } g') \in \langle \langle Rv, \langle Rw, Rv \rangle \text{prod-rel} \rangle \text{prod-rel} \rangle \text{set-rel}$

$\wedge (\text{graph.more } g, \text{graph.more } g') \in Rm \}$

$\langle \text{proof} \rangle$

abbreviation $\text{graph-rel} \equiv \langle \text{unit-rel} \rangle \text{graph-more-rel}$

lemmas $\text{graph-rel-def} = \text{graph-more-rel-def}[\text{where } Rm = \text{unit-rel}, \text{simplified}]$

lemma $\text{graph-rel-id}[simp]$: $\langle Id, Id \rangle \text{graph-rel} = Id$

$\langle \text{proof} \rangle$

lemma $\text{graph-more-rel-sv}[relator-props]$:

$\llbracket \text{single-valued } Rm; \text{single-valued } Rv; \text{single-valued } Rw \rrbracket$

$\Rightarrow \text{single-valued } (\langle Rm, Rv, Rw \rangle \text{graph-more-rel})$

$\langle \text{proof} \rangle$

lemma $[\text{autoref-itype}]$:

$\text{graph.nodes} ::_i \langle Iv, Iv \rangle_i i\text{-graph} \rightarrow_i \langle Iv \rangle_i i\text{-set}$

$\langle \text{proof} \rangle$

thm $\text{is-map-to-sorted-list-def}$

definition *nodes-to-list* $g \equiv it\text{-to-sorted-list } (\lambda - . \text{ True}) (graph.\text{nodes } g)$
lemma *nodes-to-list-itype*[*autoref-itype*]: *nodes-to-list* $::_i \langle Iv, Iv \rangle_i i\text{-graph} \rightarrow_i \langle \langle Iv \rangle_i i\text{-list} \rangle_i i\text{-nres}$
 $\langle proof \rangle$
lemma *nodes-to-list-pat*[*autoref-op-pat*]: *it-to-sorted-list* $(\lambda - . \text{ True}) (graph.\text{nodes } g) \equiv \text{nodes-to-list } g$
 $\langle proof \rangle$

definition *succ-to-list* $g v \equiv it\text{-to-sorted-list } (\lambda - . \text{ True}) (Graph.\text{succ } g v)$
lemma *succ-to-list-itype*[*autoref-itype*]:
succ-to-list $::_i \langle Iv, Iv \rangle_i i\text{-graph} \rightarrow_i Iv \rightarrow_i \langle \langle \langle Iv, Iv \rangle_i i\text{-prod} \rangle_i i\text{-list} \rangle_i i\text{-nres}$ $\langle proof \rangle$
lemma *succ-to-list-pat*[*autoref-op-pat*]: *it-to-sorted-list* $(\lambda - . \text{ True}) (Graph.\text{succ } g v) \equiv \text{succ-to-list } g v$
 $\langle proof \rangle$

context *graph begin*

definition *rel-def-internal*: *rel* $Rv Rw \equiv br \alpha \text{ invar } O \langle Rv, Rw \rangle \text{graph-rel}$

lemma *rel-def*: $\langle Rv, Rw \rangle \text{rel} \equiv br \alpha \text{ invar } O \langle Rv, Rw \rangle \text{graph-rel}$
 $\langle proof \rangle$

lemma *rel-id[simp]*: $\langle Id, Id \rangle \text{rel} = br \alpha \text{ invar}$ $\langle proof \rangle$

lemma *rel-sv[relator-props]*:

$\llbracket \text{single-valued } Rv; \text{single-valued } Rw \rrbracket \implies \text{single-valued } (\langle Rv, Rw \rangle \text{rel})$
 $\langle proof \rangle$

lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of rel i-graph*]

end

lemma (**in** *graph-nodes-it*) *autoref-nodes-it*[*autoref-rules*]:

assumes *ID*: *PREFER-id* Rv

shows $(\lambda s. \text{RETURN } (it\text{-to-list } \text{nodes-it } s), \text{nodes-to-list}) \in \langle Rv, Rw \rangle \text{rel} \rightarrow \langle \langle Rv \rangle \text{list-rel} \rangle \text{nres-rel}$
 $\langle proof \rangle$

lemma (**in** *graph-succ-it*) *autoref-succ-it*[*autoref-rules*]:

assumes *ID*: *PREFER-id* Rv *PREFER-id* Rw

shows $(\lambda s v. \text{RETURN } (it\text{-to-list } (\lambda s. \text{succ-it } s v) s), \text{succ-to-list})$

$\in \langle Rv, Rw \rangle \text{rel} \rightarrow Rv \rightarrow \langle \langle \langle Rv, Rw \rangle \text{prod-rel} \rangle \text{list-rel} \rangle \text{nres-rel}$
 $\langle proof \rangle$

11.2 Refinement

locale *dijkstraC* =

g: *StdGraph* *g-ops* +

mr: *StdMap* *mr-ops* +

qw: *StdUprio* *qw-ops*

for *g-ops* $:: ('V, 'W :: \text{weight}, 'G, 'moreg) \text{graph-ops-scheme}$

and *mr-ops* $:: ('V, (('V, 'W) \text{path} \times 'W), 'mr, 'more-mr) \text{map-ops-scheme}$

and *qw-ops* $:: ('V, 'W \text{infy}, 'qw, 'more-qw) \text{uprio-ops-scheme}$

begin
end

locale *dijkstraC-fixg* = *dijkstraC* *g-ops* *mr-ops* *qw-ops* +
Dijkstra *ga* *v0*
for *g-ops* :: ('V,'W::weight,'G,'moreg) *graph-ops-scheme*
and *mr-ops* :: ('V, (('V,'W) *path* × 'W), 'mr,'more-mr) *map-ops-scheme*
and *qw-ops* :: ('V,'W *infty*, 'qw,'more-qw) *uprio-ops-scheme*
and *ga*::('V,'W) *graph* **and** *v0*::'V **and** *g* :: 'G+
assumes *ga-trans*: (g,ga)∈br g.α g.invar

begin

abbreviation *v-rel* ≡ *Id* :: ('V×'V) *set*
abbreviation *w-rel* ≡ *Id* :: ('W×'W) *set*

definition *i-node* :: *interface* **where** *i-node* ≡ *undefined*
definition *i-weight* :: *interface* **where** *i-weight* ≡ *undefined*

lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of v-rel i-node*]
lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of w-rel i-weight*]

lemma *weight-plus-autoref*[*autoref-rules*]:

(0,0) ∈ *w-rel*
((+),(+)) ∈ *w-rel* → *w-rel* → *w-rel*
((+),(+)) ∈ ⟨*w-rel*⟩*infty-rel* → ⟨*w-rel*⟩*infty-rel* → ⟨*w-rel*⟩*infty-rel*
((<),(<)) ∈ ⟨*w-rel*⟩*infty-rel* → ⟨*w-rel*⟩*infty-rel* → *bool-rel*
⟨*proof*⟩

lemma [*autoref-rules*]: (g,ga)∈⟨*v-rel,w-rel*⟩*g.rel* ⟨*proof*⟩

lemma [*autoref-rules*]: (v0,v0)∈*v-rel* ⟨*proof*⟩

term *mpath-weight'*

lemma [*autoref-rules*]:

(*mpath-weight'*,*mpath-weight'*)
∈ ⟨⟨*v-rel*×_r*w-rel*×_r*v-rel*⟩*list-rel*×_r*w-rel*⟩*option-rel* → ⟨*w-rel*⟩*infty-rel*
(*mpath'*, *mpath'*)
∈ ⟨⟨*v-rel*×_r*w-rel*×_r*v-rel*⟩*list-rel*×_r*w-rel*⟩*option-rel*
→ ⟨⟨*v-rel*×_r*w-rel*×_r*v-rel*⟩*list-rel*⟩*option-rel*
⟨*proof*⟩

term *mdinit*

lemmas [*autoref-tyrel*] =

ty-REL[**where** *R*=*v-rel*]
ty-REL[**where** *R*=*w-rel*]
ty-REL[**where** *R*=⟨*w-rel*⟩*infty-rel*]
ty-REL[**where** *R*=⟨*v-rel*,⟨*w-rel*⟩*infty-rel*⟩*qw.rel*]
ty-REL[**where** *R*=⟨*v-rel*,⟨*v-rel*×_r*w-rel*×_r*v-rel*⟩*list-rel*×_r*w-rel*⟩*mr.rel*]
ty-REL[**where** *R*=⟨*v-rel*×_r*w-rel*×_r*v-rel*⟩*list-rel*]

lemmas [autoref-op-pat] = uprio-pats[where 'e = 'V and 'a = 'W infly]

schematic-goal *cdijkstra-refines-aux*:

shows (?c::?'c,
 mdijkstra
) ∈ ?R
⟨proof⟩

end

context *dijkstraC*

begin

concrete-definition *cdijkstra* for *g* ?v0.0

uses *dijkstraC-fixg.cdijkstra-refines-aux*
[of *g-ops mr-ops qw-ops*]

term *cdijkstra*

end

context *dijkstraC-fixg*

begin

term *cdijkstra*

term *mdijkstra*

lemma *cdijkstra-refines*:

RETURN (*cdijkstra g v0*) ≤ ↓(*build-rel mr.α mr.invar*) *mdijkstra*
⟨proof⟩

theorem *cdijkstra-correct*:

shows

weighted-graph.is-shortest-path-map ga v0 (αr (*mr.α* (*cdijkstra g v0*)))

(is ?G1)

and *mr.invar* (*cdijkstra g v0*) (is ?G2)

and *res-invarm* (*mr.α* (*cdijkstra g v0*)) (is ?G3)

⟨proof⟩

end

theorem (in *dijkstraC*) *cdijkstra-correct*:

assumes *INV*: *g.invar g*

assumes *V0*: *v0* ∈ *nodes* (*g.α g*)

assumes *nonneg-weights*: $\bigwedge v w v'. (v,w,v') \in \text{edges } (g.\alpha g) \implies 0 \leq w$

shows

weighted-graph.is-shortest-path-map (*g.α g*) *v0*

(*Dijkstra.αr* (*mr.α* (*cdijkstra g v0*))) (is ?G1)

and *Dijkstra.res-invarm* (*mr.alpha* (*cdijkstra g v0*)) (**is** ?G2)
 <proof>

Example instantiation with HashSet-based graph, red-black-tree based result map, and finger-tree based priority queue.

<ML>

interpretation *hrf*: *dijkstraC hlg-ops rm-ops aluprioi-ops*
 <proof>

<ML>

definition *hrf-dijkstra* \equiv *hrf.cdijkstra*

lemmas *hrf-dijkstra-correct* = *hrf.cdijkstra-correct*[*folded hrf-dijkstra-def*]

export-code *hrf-dijkstra checking* *SML*

export-code *hrf-dijkstra in OCaml*

export-code *hrf-dijkstra in Haskell*

export-code *hrf-dijkstra checking* *Scala*

definition *hrfn-dijkstra* :: (*nat,nat*) *hlg* \Rightarrow -
where *hrfn-dijkstra* \equiv *hrf-dijkstra*

export-code *hrfn-dijkstra checking* *SML*

lemmas *hrfn-dijkstra-correct* =

hrf-dijkstra-correct[**where** ?'a = *nat* **and** ?'b = *nat*, *folded hrfn-dijkstra-def*]

end

12 Performance Test

theory *Test*

imports *Dijkstra-Impl-Adet*

begin

In this theory, we test our implementation of Dijkstra's algorithm for larger, randomly generated graphs.

Simple linear congruence generator for (low-quality) random numbers:

definition *lcg-next s* = $((81::nat)*s + 173) \bmod 268435456$

Generate a complete graph over the given number of vertices, with random weights:

definition *ran-graph* :: *nat* \Rightarrow *nat* \Rightarrow (*nat list* \times (*nat* \times *nat* \times *nat*) *list*) **where**

ran-graph vertices seed ==

$([0::nat..<vertices],fst$

$(while (\lambda (g,v,s). v < vertices)$

$(\lambda (g,v,s).$

$let (g'',v'',s'') = (while (\lambda (g',v',s'). v' < vertices)$

```

    (λ (g',v',s'). ((v,s',v')#g',v'+1,lcg-next s'))
    (g,0,s))
  in (g'',v+1,s'')
  ([],0,lcg-next seed)))

```

To experiment with the exported code, we fix the node type to natural numbers, and add a from-list conversion:

```

type-synonym nat-res = (nat,((nat,nat) path × nat)) rm
type-synonym nat-list-res = (nat × (nat,nat) path × nat) list

```

```

definition nat-dijkstra :: (nat,nat) hlg ⇒ nat ⇒ nat-res where
  nat-dijkstra ≡ hrfn-dijkstra

```

```

definition hlg-from-list-nat :: (nat,nat) adj-list ⇒(nat,nat) hlg where
  hlg-from-list-nat ≡ hlg.from-list

```

```

definition
  nat-res-to-list :: nat-res ⇒ nat-list-res
where nat-res-to-list ≡ rm.to-list

```

```

value nat-res-to-list (nat-dijkstra (hlg-from-list-nat (ran-graph 4 8912)) 0)

```

⟨ML⟩

end

References

- [1] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik 1*, pages 269–271, 1959.
- [2] F. Haftmann. *Code Generation from Specifications in Higher Order Logic*. PhD thesis, Technische Universität München, 2009.
- [3] F. Haftmann and T. Nipkow. Code generation via higher-order rewrite systems. In *Functional and Logic Programming (FLOPS 2010)*, LNCS. Springer, 2010.
- [4] P. Lammich. Collections framework. In G. Klein, T. Nipkow, and L. Paulson, editors, *Archive of Formal Proofs*. <http://isa-afp.org/entries/collections.shtml>, Dec. 2009. Formal proof development.
- [5] P. Lammich. Refinement for monadic programs. 2011. Submitted to AFP.
- [6] P. Lammich and A. Lochbihler. The Isabelle collections framework. In M. Kaufmann and L. Paulson, editors, *Interactive Theorem Pro-*

ing, volume 6172 of *Lecture Notes in Computer Science*, pages 339–354. Springer, 2010.

- [7] B. Nordhoff, S. Körner, and P. Lammich. Finger trees. In G. Klein, T. Nipkow, and L. Paulson, editors, *Archive of Formal Proofs*. <http://isa-afp.org/entries/Tree-Automata.shtml>, Oct. 2010. Formal proof development.