Dijkstra's Algorithm

Benedikt Nordhoff Peter Lammich

March 12, 2013

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

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**
    $[\![x{\in}S; \forall x'{\in}S.\ f\ x \leq f\ x']\!] \Longrightarrow x \in$ *least-map* $f\ S$

  lemma *least-map-subset*: *least-map* $f\ S \subseteq S$
    ⟨*proof*⟩

  lemmas *least-map-elemD* = *set-mp*[*OF least-map-subset*]

  lemma *least-map-leD*:
    assumes $x \in$ *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 ∀ *y*∈*S*. *f x* ≤ *f y*
shows *x* ∈ *least-map f* (*insert x S*)
⟨*proof*⟩

lemma *least-map-insert-nmin*:
⟦ *x*∈*least-map f S*; *f x* ≤ *f a* ⟧ ⟹ *x*∈*least-map f* (*insert a S*)
⟨*proof*⟩

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

lemma *inf-absorb-less*[*simp*]:
a < *b* ⟹ *inf a b* = *a*
a < *b* ⟹ *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 :: $('v \times 'w \times 'v)$ *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* $G \subseteq$ *nodes* G
    snd‘*snd*‘*edges* $G \subseteq$ *nodes* G
begin
  abbreviation $V \equiv$ *nodes* G
  abbreviation $E \equiv$ *edges* G

  lemma *E-validD*: **assumes** $(v,e,v') \in E$
    shows $v \in V$ $v' \in V$
    ⟨*proof*⟩

end

3.2 Basic operations on Graphs

The empty graph.

definition *empty* **where**
  empty $\equiv$ (| *nodes* = {}, *edges* = {} |)

Adds a node to a graph.

definition *add-node* **where**
  add-node v $g \equiv$ (| *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 $g \equiv$ (|
  nodes = *nodes* $g - \{v\}$,
  edges = *edges* $g \cap (-\{v\}) \times UNIV \times (-\{v\})$
  |)

Adds an edge to a graph.

definition *add-edge* **where** *add-edge* v e v' $g \equiv$ (|
  nodes = $\{v,v'\} \cup$ *nodes* g,
  edges = *insert* (v,e,v') (*edges* g)
  |)

Deletes an edge from a graph.

definition *delete-edge* **where** *delete-edge* v e v' $g \equiv$ (|
  nodes = *nodes* g, *edges* = *edges* $g - \{(v,e,v')\}$ |)

Successors of a node.

definition *succ* :: ('*v*,'*w*) *graph* ⇒ '*v* ⇒ ('*w*×'*v*) *set*
where *succ G v* ≡ {(*w*,*v'*). (*v*,*w*,*v'*)∈*edges G*}

Now follow some simplification lemmas.

lemma *empty-valid*[*simp*]: *valid-graph empty*
⟨*proof*⟩
lemma *add-node-valid*[*simp*]: **assumes** *valid-graph g*
shows *valid-graph* (*add-node v g*)
⟨*proof*⟩
lemma *delete-node-valid*[*simp*]: **assumes** *valid-graph g*
shows *valid-graph* (*delete-node v g*)
⟨*proof*⟩
lemma *add-edge-valid*[*simp*]: **assumes** *valid-graph g*
shows *valid-graph* (*add-edge v e v' g*)
⟨*proof*⟩
lemma *delete-edge-valid*[*simp*]: **assumes** *valid-graph g*
shows *valid-graph* (*delete-edge v e v' g*)
⟨*proof*⟩

lemma *succ-finite*[*simp*, *intro*]: *finite* (*edges G*) ⟹ *finite* (*succ G v*)
⟨*proof*⟩

lemma *nodes-empty*[*simp*]: *nodes empty* = {} ⟨*proof*⟩
lemma *edges-empty*[*simp*]: *edges empty* = {} ⟨*proof*⟩
lemma *succ-empty*[*simp*]: *succ empty v* = {} ⟨*proof*⟩

lemma *nodes-add-node*[*simp*]: *nodes* (*add-node v g*) = *insert v* (*nodes g*)
⟨*proof*⟩
lemma *nodes-add-edge*[*simp*]:
nodes (*add-edge v e v' g*) = *insert v* (*insert v'* (*nodes g*))
⟨*proof*⟩
lemma *edges-add-edge*[*simp*]:
edges (*add-edge v e v' g*) = *insert* (*v*,*e*,*v'*) (*edges g*)
⟨*proof*⟩
lemma *edges-add-node*[*simp*]:
edges (*add-node v g*) = *edges g*
⟨*proof*⟩

lemma (**in** *valid-graph*) *succ-subset*: *succ G v* ⊆ *UNIV*×*V*
⟨*proof*⟩

3.3 Paths

A path is represented by a list of adjacent edges.

type-synonym ('*v*,'*w*) *path* = ('*v*×'*w*×'*v*) *list*

context *valid-graph*

begin

The following predicate describes a valid path:

fun *is-path* :: $'v \Rightarrow ('v,'w)$ *path* $\Rightarrow 'v \Rightarrow$ *bool* **where**
is-path v [] $v' \longleftrightarrow v{=}v' \wedge v'{\in}V$ |
is-path v $((v1,w,v2)\#p)$ $v' \longleftrightarrow v{=}v1 \wedge (v1,w,v2){\in}E \wedge$ *is-path* $v2$ p v'

lemma *is-path-simps*[*simp*, *intro*!]:
is-path v [] $v \longleftrightarrow v{\in}V$
is-path v $[(v,w,v')]$ $v' \longleftrightarrow (v,w,v'){\in}E$
⟨*proof*⟩

lemma *is-path-memb*[*simp*]:
is-path v p $v' \Longrightarrow v{\in}V \wedge v'{\in}V$
⟨*proof*⟩

lemma *is-path-split*:
is-path v $(p1@p2)$ $v' \longleftrightarrow (\exists\, u.$ *is-path* v $p1$ $u \wedge$ *is-path* u $p2$ $v')$
⟨*proof*⟩

lemma *is-path-split'*[*simp*]:
is-path v $(p1@(u,w,u')\#p2)$ v'
$\longleftrightarrow$ *is-path* v $p1$ $u \wedge (u,w,u'){\in}E \wedge$ *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* $\Rightarrow 'v$ *set* **where**
int-vertices $p \equiv$ *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* $p1 \cup$ *int-vertices* $p2$
⟨*proof*⟩

lemma (**in** *valid-graph*) *int-vertices-subset*:
is-path v p $v' \Longrightarrow$ *int-vertices* $p \subseteq V$
⟨*proof*⟩

lemma *int-vertices-empty*[*simp*]: *int-vertices* p = {} $\longleftrightarrow p{=}[]$
⟨*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** $u \in$ *int-vertices p*
obtains *p1 p2* **where**
$p = p1 @ p2$ **and**
is-path v p1 u **and**
$u \notin$ *int-vertices p1*
⟨*proof*⟩

3.4 Weighted Graphs

locale *valid-mgraph* = *valid-graph G* **for** *G*::('*v*,'*w*::*monoid-add*) *graph*

definition *path-weight* :: ('*v*,'*w*::*monoid-add*) *path* ⇒ '*w*
where *path-weight p* ≡ *listsum* (*map* (*fst* ∘ *snd*) *p*)

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

lemma *path-weight-empty*[*simp*]: *path-weight* [] = *0*
⟨*proof*⟩

lemma *path-weight-cons*[*simp*]:
(*path-weight* (*e*#*p*)::'*w*::*monoid-add*) = *fst* (*snd e*) + *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 $x + y = 0 \longleftrightarrow x = 0 \wedge y = 0$
  ⟨*proof*⟩

lemma *add-incr*: $0 \leq b \Longrightarrow a \leq a+b$
  ⟨*proof*⟩

lemma *add-incr-left*[*simp*, *intro*!]: $0 \leq b \Longrightarrow a \leq b + a$
  ⟨*proof*⟩

lemma *sum-not-less*[*simp*, *intro*!]:
  $0 \leq b \Longrightarrow \neg\ (a+b < a)$
  $0 \leq a \Longrightarrow \neg\ (a+b < b)$
  ⟨*proof*⟩

end

instance *nat* :: *weight* ⟨*proof*⟩
instance *int* :: *weight* ⟨*proof*⟩
instance *rat* :: *weight* ⟨*proof*⟩
instance *real* :: *weight* ⟨*proof*⟩

class *top-weight* = *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*]:
$\neg (a < top) \longleftrightarrow a{=}top$
⟨*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*]: $e{\neq}Infty \Longrightarrow Num\ (val\ e) = e$ ⟨*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*) ⟷ $a{\leq}b$

lemma [*simp*]: $Infty{\leq}a \longleftrightarrow a{=}Infty$
⟨*proof*⟩

fun *less-infty* **where**
less Infty - ⟷ *False* |
less (*Num* -) *Infty* ⟷ *True* |
less (*Num a*) (*Num b*) ⟷ $a{<}b$

lemma [*simp*]: *less a Infty* ⟷ $a \neq Infty$
⟨*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*
../Refine-Monadic/Refine ../Refine-Monadic/Collection-Bindings
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* :: $'V \Rightarrow ('V \rightharpoonup ('V,'W)\ path) \Rightarrow bool$
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*) |
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* |
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* $= ('V\ set) \times ('V \rightharpoonup ('V,'W)\ path)$

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*]: $(v,w,v') \in edges\ G \Longrightarrow 0 \leq w$
begin

Paths have non-negative weights.

lemma *path-nonneg-weight*: *is-path v p v'* $\Longrightarrow$ *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*
$wl \subseteq V \wedge$
$(\forall v \in V.\ \forall p.\ res\ v = Some\ p \longrightarrow is\text{-}path\ v0\ p\ v \wedge int\text{-}vertices\ p \subseteq V - wl) \wedge$
$(\forall v \in V - wl.\ \forall p.\ is\text{-}path\ v0\ p\ v$
$\longrightarrow path\text{-}weight'\ (res\ v) \leq path\text{-}weight'\ (Some\ p)) \wedge$
$(\forall v \in wl.\ \forall p.\ is\text{-}path\ v0\ p\ v \wedge int\text{-}vertices\ p \subseteq V - wl$
$\longrightarrow path\text{-}weight'\ (res\ v) \leq path\text{-}weight'\ (Some\ p)$
)

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

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

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

definition *dinit* :: $('V,'W)$ *state nres* **where**

```
dinit ≡ SPEC ( λ(wl,res) .
    wl=V ∧ res v0 = Some [] ∧ (∀ v∈V−{v0}. res v = None))
```

The initial state satisfies the invariant.

```
lemma dinit-invar: dinit ≤ SPEC dinvar
⟨proof⟩
```

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-min :: ('V,'W) state ⇒ ('V × ('V,'W) state) nres where
  pop-min σ ≡ do {
    let (wl,res)=σ;
    ASSERT (wl≠{});
    v ← RES (least-map (path-weight' ∘ 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-spec :: 'V ⇒ ('V,'W) state ⇒ ('V,'W) state ⇒ bool
  where
  ⟦ ∀ v'∈V.
    res' v' ∈ least-map path-weight' (
      { res v' } ∪ { Some (p@[(v,w,v')]) | p w. res v = Some p ∧ (v,w,v')∈E }
    )
   ⟧ ⟹ update-spec v (wl,res) (wl,res')
```

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

```
definition update-pre :: 'V ⇒ ('V,'W) state ⇒ bool where
  update-pre v σ ≡ let (wl,res)=σ in v∈V
    ∧ (∀ v'∈V−wl. v'≠v ⟶ (∀ p. is-path v0 p v'
        ⟶ path-weight' (res v') ≤ path-weight' (Some p)))
    ∧ (∀ v'∈V. ∀ p. res v' = Some p ⟶ is-path v0 p v')

definition update :: 'V ⇒ ('V,'W) state ⇒ ('V,'W) state nres where
  update v σ ≡ do {ASSERT (update-pre v σ); SPEC (update-spec v σ)}
```

Finally, we define Dijkstra's algorithm:

```
definition dijkstra where
  dijkstra ≡ do {
    σ0←dinit;
    (-,res) ← WHILE_T^dinvar (λ(wl,-). wl≠{})
        (λσ.
```

```
      do { (v,σ') ← pop-min σ; update v σ' }
    )
    σ0;
  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 uinvar
  :: 'V ⇒ 'V set ⇒ - ⇒ ('W×'V) set ⇒ ('V,'W) state ⇒ bool where
  uinvar 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;
    FOREACH^uinvar 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' σ' ≤ ⇓Id (update v σ)
  ⟨proof⟩
```

We integrate the new update function into the main algorithm:

```
  definition dijkstra' where
    dijkstra' ≡ do {
      σ0 ← dinit;
      (-,res) ← WHILE_T^dinvar (λ(wl,-). wl≠{})
            (λσ. do {(v,σ') ← pop-min σ; update' v σ'})
            σ0;
      RETURN res
    }


  lemma dijkstra'-refines: dijkstra' ≤ ⇓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 effcient.

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 ⇀ 'W infty)
type-synonym ('V,'W) mres = ('V ⇀ (('V,'W) path × 'W))
type-synonym ('V,'W) mstate = ('V,'W) mwl × ('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 αw::('V,'W) mwl ⇒ 'V set where αw ≡ dom
  definition αr::('V,'W) mres ⇒ 'V ⇀ ('V,'W) path where
    αr ≡ λres v. case res v of None ⇒ None | Some (p,w) ⇒ Some (rev p)
  definition αs:: ('V,'W) mstate ⇒ ('V,'W) state where
    αs ≡ map-pair αw αr
```

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

```
  definition res-invarm :: ('V ⇀ (('V,'W) path×'W)) ⇒ bool where
```

```
  res-invarm res ≡ (∀ v. case res v of
      None ⇒ True |
      Some (p,w) ⇒ w = path-weight (rev p))
definition dinvarm :: ('V,'W) mstate ⇒ bool where
  dinvarm σ ≡ let (wl,res) = σ in
    (∀ v∈dom wl. the (wl v) = mpath-weight' (res v)) ∧ res-invarm res

lemma mpath-weight'-correct: ⟦dinvarm (wl,res)⟧ ⟹
  mpath-weight' (res v) = path-weight' (αr res v)

  ⟨proof⟩

lemma mpath'-correct: ⟦dinvarm (wl,res)⟧ ⟹
  mpath' (res v) = Option.map rev (αr res v)
  ⟨proof⟩

lemma wl-weight-correct:
  assumes INV: dinvarm (wl,res)
  assumes WLV: wl v = Some w
  shows path-weight' (αr res v) = w
⟨proof⟩
```

The initial state is constructed using an iterator:

```
definition mdinit :: ('V,'W) mstate nres where
  mdinit ≡ do {
    wl ← FOREACH V (λv wl. RETURN (wl(v↦Infty))) Map.empty;
    RETURN (wl(v0↦Num 0),[v0 ↦ ([],0)])
  }

lemma mdinit-refines: mdinit ≤ ⇓(build-rel αs dinvarm) dinit
  ⟨proof⟩
```

The new pop function:

```
definition
  mpop-min :: ('V,'W) mstate ⇒ ('V × 'W infty × ('V,'W) mstate) nres
  where
  mpop-min σ ≡ do {
    let (wl,res) = σ;
    (v,w,wl')←prio-pop-min wl;
    RETURN (v,w,(wl',res))
  }

lemma mpop-min-refines:
  ⟦ (σ,σ') ∈ build-rel αs dinvarm ⟧ ⟹
    mpop-min σ ≤
     ⇓(build-rel
        (λ(v,w,σ). (v,αs σ))
        (λ(v,w,σ). dinvarm σ ∧ w = mpath-weight' (snd σ v)))
    (pop-min σ')
```

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

The new update function:

definition *uinvarm v wl res it σ* ≡
uinvar v wl res it (*αs σ*) ∧ *dinvarm σ*

definition *mupdate* :: *'V* ⇒ *'W infty* ⇒ (*'V*,*'W*) *mstate* ⇒ (*'V*,*'W*) *mstate nres*
where
mupdate v wv σ ≡ *do* {
ASSERT (*update-pre v* (*αs σ*) ∧ *wv*=*mpath-weight'* (*snd σ v*));
let (*wl*,*res*) = *σ*;
let pv = *mpath'* (*res v*);
FOREACH$^{uinvarm\ v}$ (*αw wl*) (*αr res*) (*succ G v*) (λ(*w'*,*v'*) (*wl*,*res*).
if (*wv* + *Num w'* < *mpath-weight'* (*res v'*)) *then do* {
ASSERT (*v'*∈*dom wl* ∧ *pv* ≠ *None*);
RETURN (*wl*(*v'*↦*wv* + *Num w'*),
res(*v'* ↦ ((*v*,*w'*,*v'*)#*the pv*,*val wv* + *w'*)))
} *else RETURN* (*wl*,*res*)
) (*wl*,*res*)
}

lemma *mupdate-refines*:
assumes *SREF*: (*σ*,*σ'*)∈*build-rel αs dinvarm*
assumes *WV*: *wv* = *mpath-weight'* (*snd σ v*)
assumes *VV'*: *v'*=*v*
shows *mupdate v wv σ* ≤ ⇓(*build-rel αs dinvarm*) (*update' v' σ'*)
⟨*proof*⟩

Finally, we assemble the refined algorithm:

definition *mdijkstra* **where**
mdijkstra ≡ *do* {
σ0 ← *mdinit*;
(-,*res*) ← *WHILE*$_T$^{dinvarm} (λ(*wl*,-). *dom wl*≠{})
(λ*σ*. *do* { (*v*,*wv*,*σ'*) ← *mpop-min σ*; *mupdate v wv σ'* })
σ0;
RETURN res
}

lemma *mdijkstra-refines*: *mdijkstra* ≤ ⇓(*build-rel αr res-invarm*) *dijkstra'*
⟨*proof*⟩

end

end

6 Graph Interface

theory *GraphSpec*
imports *Main Graph ../Refine-Monadic/Refine*

begin

This theory defines an ICF-style interface for graphs.

type-synonym $('V,'W,'G)$ *graph-*α $= 'G \Rightarrow ('V,'W)$ *graph*

locale *graph* =
  fixes $\alpha :: 'G \Rightarrow ('V,'W)$ *graph*
  fixes *invar* $:: 'G \Rightarrow$ *bool*
  assumes *finite*[*simp*, *intro*!]:
    invar $g \Longrightarrow$ *finite* (*nodes* $(\alpha\ g)$)
    invar $g \Longrightarrow$ *finite* (*edges* $(\alpha\ g)$)
  assumes *valid*: *invar* $g \Longrightarrow$ *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*:
    α (*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 $g \Longrightarrow$ *invar* (*add-node* $v\ g$)
    invar $g \Longrightarrow \alpha$ (*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 $g \Longrightarrow$ *invar* (*delete-node* $v\ g$)
    invar $g \Longrightarrow \alpha$ (*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 $\alpha :: 'G \Rightarrow ('V,'W)$ *graph*
  fixes *add-edge* $:: 'V \Rightarrow 'W \Rightarrow 'V \Rightarrow 'G \Rightarrow 'G$
  assumes *add-edge-correct*:
    invar $g \Longrightarrow$ *invar* (*add-edge* $v\ e\ v'\ g$)
    invar $g \Longrightarrow \alpha$ (*add-edge* $v\ e\ v'\ g$) = *Graph.add-edge* $v\ e\ v'\ (\alpha\ g)$

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

type-synonym $('V,'W,'\sigma,'G)$ *graph-nodes-it* $= 'G \Rightarrow ('V,'\sigma)$ *set-iterator*
locale *graph-nodes-it* $=$ *graph* $+$
  constrains $\alpha :: 'G \Rightarrow ('V,'W)$ *graph*
  fixes *nodes-it* $:: 'G \Rightarrow ('V,'\sigma)$ *set-iterator*
  assumes *nodes-it-correct*:
    invar $g \Longrightarrow$ *set-iterator* (*nodes-it* g) (*Graph.nodes* (α g))

type-synonym $('V,'W,'\sigma,'G)$ *graph-edges-it*
  $= 'G \Rightarrow (('V \times 'W \times 'V),'\sigma)$ *set-iterator*
locale *graph-edges-it* $=$ *graph* $+$
  constrains $\alpha :: 'G \Rightarrow ('V,'W)$ *graph*
  fixes *edges-it* $:: ('V,'W,'\sigma,'G)$ *graph-edges-it*
  assumes *edges-it-correct*:
    invar $g \Longrightarrow$ *set-iterator* (*edges-it* g) (*Graph.edges* (α g))

type-synonym $('V,'W,'\sigma,'G)$ *graph-succ-it* $=$
  $'G \Rightarrow 'V \Rightarrow ('W \times 'V,'\sigma)$ *set-iterator*
locale *graph-succ-it* $=$ *graph* $+$
  constrains $\alpha :: 'G \Rightarrow ('V,'W)$ *graph*
  fixes *succ-it* $:: 'G \Rightarrow 'V \Rightarrow ('W \times 'V,'\sigma)$ *set-iterator*
  assumes *succ-it-correct*:
    invar $g \Longrightarrow$ *set-iterator* (*succ-it* g v) (*Graph.succ* (α g) v)

6.1 Adjacency Lists

type-synonym $('V,'W)$ *adj-list* $= 'V$ *list* $\times$ $('V \times 'W \times 'V)$ *list*

definition *adjl-*$\alpha :: ('V,'W)$ *adj-list* $\Rightarrow ('V,'W)$ *graph* **where**
  adjl-α $l \equiv$ *let* $(nl,el) = l$ *in* ⦇
    nodes $=$ *set* $nl \cup$ *fst*‘*set* $el \cup$ *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* $\Rightarrow 'G$
locale *graph-from-list* $=$ *graph* $+$
  constrains $\alpha :: 'G \Rightarrow ('V,'W)$ *graph*

fixes *from-list* :: $('V,'W)$ *adj-list* $\Rightarrow$ $'G$
assumes *from-list-correct*:
$invar$ (*from-list* l)
α (*from-list* l) = *adjl-*α l

type-synonym $('V,'W,'G)$ *graph-to-list* = $'G \Rightarrow ('V,'W)$ *adj-list*
locale *graph-to-list* = *graph* +
constrains α :: $'G \Rightarrow ('V,'W)$ *graph*
fixes *to-list* :: $'G \Rightarrow ('V,'W)$ *adj-list*
assumes *to-list-correct*:
$invar\ g \Longrightarrow$ *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 :: $'G \Rightarrow$ *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*

locale *StdGraphDefs* =
fixes *ops* :: $('V,'W,'G,'m)$ *graph-ops-scheme*
begin
abbreviation α **where** $\alpha \equiv$ *gop-*α *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*
end

locale *StdGraph* = *StdGraphDefs* +
graph α *invar* +
graph-empty α *invar empty* +
graph-add-node α *invar add-node* +
graph-delete-node α *invar delete-node* +
graph-add-edge α *invar add-edge* +
graph-delete-edge α *invar delete-edge* +
graph-from-list α *invar from-list* +
graph-to-list α *invar to-list*
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 $\Longrightarrow$ *set-iterator* (*nodes-it g*) (*nodes* (α *g*))
⟨*proof*⟩

lemma (**in** *graph-edges-it*) *edges-it-is-iterator*[*refine-transfer*]:
invar g $\Longrightarrow$ *set-iterator* (*edges-it g*) (*edges* (α *g*))
⟨*proof*⟩

lemma (**in** *graph-succ-it*) *succ-it-is-iterator*[*refine-transfer*]:
invar g $\Longrightarrow$ *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 ../Collections/iterator/SetIteratorOperations*
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* (λ*g v*. *a v g*) (*e* ()) *nl*
in foldl (λ*g* (*v*,*e*,*v'*). *u v e v' g*) *g1 el*

lemma *gga-from-list-correct*:
fixes α :: *'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*

definition *gga-edges-it* ::
($'V$,$'W$,$'\sigma$,$'G$) *graph-nodes-it* ⇒ ($'V$,$'W$,$'\sigma$,$'G$) *graph-succ-it*
⇒ ($'V$,$'W$,$'\sigma$,$'G$) *graph-edges-it*
where *gga-edges-it ni si G* ≡ *set-iterator-product* (*ni G*) (λ*v*. *si G v*)

lemma *gga-edges-it-correct*:
fixes *ni*::($'V$,$'W$,$'\sigma$,$'G$) *graph-nodes-it*
fixes α :: $'G$ ⇒ ($'V$,$'W$) *graph*
assumes *graph-nodes-it* α *invar ni*
assumes *graph-succ-it* α *invar si*
shows *graph-edges-it* α *invar* (*gga-edges-it ni si*)
⟨*proof*⟩

definition *gga-to-list* ::
($'V$,$'W$,-,$'G$) *graph-nodes-it* ⇒ ($'V$,$'W$,-,$'G$) *graph-edges-it* ⇒
($'V$,$'W$,$'G$) *graph-to-list*
where
gga-to-list ni ei g ≡
(*ni g* (λ-. *True*) (*op* #) [], *ei g* (λ-. *True*) (*op* #) [])

lemma *gga-to-list-correct*:
fixes α :: $'G$ ⇒ ($'V$,$'W$) *graph*
assumes *graph-nodes-it* α *invar ni*
assumes *graph-edges-it* α *invar ei*
shows *graph-to-list* α *invar* (*gga-to-list ni ei*)
⟨*proof*⟩

end

8 Implementing Graphs by Maps

theory *GraphByMap*
imports *../Collections/Collections*
GraphSpec GraphGA
begin

locale *gbm-defs* =
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
  definition gbm-α :: ('V,'W,'m1) graph-α where
    gbm-α m1 ≡
    (| nodes = dom (m1.α m1),
      edges = {(v,w,v').
        ∃ m2 s3. m1.α m1 v = Some m2
          ∧ m2.α m2 v' = Some s3
          ∧ w∈s3.α s3
      }
    |)

  definition gbm-invar m1 ≡
    m1.invar m1 ∧
    (∀ m2∈ran (m1.α m1). m2.invar m2 ∧
      (∀ s3∈ran (m2.α m2). s3.invar s3)
    ) ∧ valid-graph (gbm-α m1)


  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

definition map-Sigma M1 F2 ≡ {
  (x,y). ∃ v. M1 x = Some v ∧ y∈F2 v
}

lemma map-Sigma-alt: map-Sigma M1 F2 = Sigma (dom M1) (λx.
  F2 (the (M1 x)))
  ⟨proof⟩

sublocale gbm-defs < graph gbm-α gbm-invar
⟨proof⟩
```

lemma *ranE*:
assumes $v \in ran\ m$
obtains k **where** $m\ k = Some\ v$
⟨*proof*⟩
lemma *option-bind-alt*:
$Option.bind\ x\ f = (case\ x\ of\ None \Rightarrow None \mid Some\ v \Rightarrow f\ v)$
⟨*proof*⟩

context *gbm-defs*
begin

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

lemma *gbm-empty-correct*:
graph-empty gbm-α gbm-invar gbm-empty
⟨*proof*⟩

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

lemma *gbm-add-node-correct*:
graph-add-node gbm-α gbm-invar gbm-add-node
⟨*proof*⟩

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

lemma *gbm-delete-node-correct*:
graph-delete-node gbm-α gbm-invar gbm-delete-node
⟨*proof*⟩

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

lemma *gbm-add-edge-correct*:
graph-add-edge gbm-α gbm-invar gbm-add-edge
⟨*proof*⟩

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

lemma *gbm-delete-edge-correct*:
graph-delete-edge gbm-α gbm-invar gbm-delete-edge
⟨*proof*⟩

definition *gbm-nodes-it*
:: (*'m1* ⇒ (*'V*×*'m2*,*'σ*) *set-iterator*) ⇒ (*'V*,*'W*,*'σ*,*'m1*) *graph-nodes-it*
where
gbm-nodes-it mit g ≡ *map-iterator-dom* (*mit g*)

lemma *gbm-nodes-it-correct*:
assumes *mit*: *map-iteratei m1.α m1.invar mit*
shows *graph-nodes-it gbm-α gbm-invar* (*gbm-nodes-it mit*)
⟨*proof*⟩

term *set-iterator-product*

definition *gbm-edges-it*
::
(*'m1* ⇒ (*'V*×*'m2*,*'σ*) *set-iterator*) ⇒
(*'m2* ⇒ (*'V*×*'s3*,*'σ*) *set-iterator*) ⇒
(*'s3* ⇒ (*'W*,*'σ*) *set-iterator*) ⇒
(*'V*,*'W*,*'σ*,*'m1*) *graph-edges-it*
where
gbm-edges-it mit1 mit2 sit g ≡ *set-iterator-image*
(λ((*v1*,*m1*),(*v2*,*m2*),*w*). (*v1*,*w*,*v2*))
(*set-iterator-product* (*mit1 g*)
(λ(*v*,*m2*). *set-iterator-product* (*mit2 m2*) (λ(*w*,*s3*). *sit s3*)))

lemma *gbm-edges-it-correct*:
assumes *mit1*: *map-iteratei m1.α m1.invar mit1*
assumes *mit2*: *map-iteratei m2.α m2.invar mit2*
assumes *sit*: *set-iteratei s3.α s3.invar sit*

shows *graph-edges-it gbm-α gbm-invar* (*gbm-edges-it mit1 mit2 sit*)
⟨*proof*⟩

definition *gbm-succ-it* ::
(*'m2* ⇒ (*'V*×*'s3*,*'σ*) *set-iterator*) ⇒
(*'s3* ⇒ (*'W*,*'σ*) *set-iterator*) ⇒
(*'V*,*'W*,*'σ*,*'m1*) *graph-succ-it*
where
gbm-succ-it mit2 sit g v ≡ *case m1.lookup v g of*
None ⇒ *set-iterator-emp* |
Some m2 ⇒
set-iterator-image (λ((*v'*,*m2*),*w*). (*w*,*v'*))
(*set-iterator-product* (*mit2 m2*) (λ(*v'*,*s*). *sit s*))

lemma *gbm-succ-it-correct*:
assumes *mit2*: *map-iteratei m2.α m2.invar mit2*
assumes *sit*: *set-iteratei s3.α s3.invar sit*
shows *graph-succ-it gbm-α gbm-invar* (*gbm-succ-it mit2 sit*)
⟨*proof*⟩

definition
gbm-from-list ≡ *gga-from-list gbm-empty gbm-add-node gbm-add-edge*

lemmas *gbm-from-list-correct* = *gga-from-list-correct*[
OF gbm-empty-correct gbm-add-node-correct gbm-add-edge-correct,
folded gbm-from-list-def]

end

end

9 Graphs by Hashmaps

theory *HashGraphImpl*
imports *GraphByMap* ../*Collections*/*Collections*
begin

Abbreviation: hlg

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

interpretation *hlg-defs*!: *gbm-defs hm-ops hm-ops ls-ops hh-map-value-image-filter*
⟨*proof*⟩

definition *hlg-α* :: (*'V*::*hashable*,*'E*) *hlg* ⇒ (*'V*,*'E*) *graph*

where *hlg-α* ≡ *hlg-defs.gbm-α*
definition *hlg-invar* ≡ *hlg-defs.gbm-invar*
definition *hlg-empty* ≡ *hlg-defs.gbm-empty*
definition *hlg-add-node* ≡ *hlg-defs.gbm-add-node*
definition *hlg-delete-node* ≡ *hlg-defs.gbm-delete-node*
definition *hlg-add-edge* ≡ *hlg-defs.gbm-add-edge*
definition *hlg-delete-edge* ≡ *hlg-defs.gbm-delete-edge*
definition *hlg-from-list* ≡ *hlg-defs.gbm-from-list*

definition *hlg-nodes-it* ≡ *hlg-defs.gbm-nodes-it hm-iteratei*
definition *hlg-edges-it* ≡ *hlg-defs.gbm-edges-it hm-iteratei hm-iteratei ls-iteratei*
definition *hlg-succ-it* ≡ *hlg-defs.gbm-succ-it hm-iteratei ls-iteratei*

definition *hlg-to-list* ≡ *gga-to-list hlg-nodes-it hlg-edges-it*

lemmas *hlg-gbm-defs* =
hlg-defs.gbm-empty-def [*abs-def*]
hlg-defs.gbm-add-node-def [*abs-def*]
hlg-defs.gbm-delete-node-def [*abs-def*]
hlg-defs.gbm-add-edge-def [*abs-def*]
hlg-defs.gbm-delete-edge-def [*abs-def*]
hlg-defs.gbm-from-list-def [*abs-def*]
hlg-defs.gbm-nodes-it-def [*abs-def*]
hlg-defs.gbm-succ-it-def [*abs-def*]
hlg-defs.gbm-edges-it-def [*abs-def*]

lemmas *hlg-defs* =
hlg-α-def
hlg-invar-def
hlg-empty-def
hlg-add-node-def
hlg-delete-node-def
hlg-add-edge-def
hlg-delete-edge-def
hlg-from-list-def
hlg-to-list-def
hlg-nodes-it-def
hlg-edges-it-def
hlg-succ-it-def

lemmas [*code*] = *hlg-defs*[*unfolded hlg-gbm-defs*]

lemmas *hlg-empty-impl* = *hlg-defs.gbm-empty-correct*[*folded hlg-defs*]
interpretation *hlg*!: *graph-empty hlg-α hlg-invar hlg-empty*
⟨*proof*⟩
lemmas *hlg-add-node-impl* = *hlg-defs.gbm-add-node-correct*[*folded hlg-defs*]
interpretation *hlg*!: *graph-add-node hlg-α hlg-invar hlg-add-node*

⟨*proof*⟩
lemmas *hlg-delete-node-impl* = *hlg-defs.gbm-delete-node-correct*[*folded hlg-defs*]
interpretation *hlg*!: *graph-delete-node hlg-α hlg-invar hlg-delete-node*
⟨*proof*⟩
lemmas *hlg-add-edge-impl* = *hlg-defs.gbm-add-edge-correct*[*folded hlg-defs*]
interpretation *hlg*!: *graph-add-edge hlg-α hlg-invar hlg-add-edge*
⟨*proof*⟩
lemmas *hlg-delete-edge-impl* = *hlg-defs.gbm-delete-edge-correct*[*folded hlg-defs*]
interpretation *hlg*!: *graph-delete-edge hlg-α hlg-invar hlg-delete-edge*
⟨*proof*⟩
lemmas *hlg-from-list-impl* = *hlg-defs.gbm-from-list-correct*[*folded hlg-defs*]
interpretation *hlg*!: *graph-from-list hlg-α hlg-invar hlg-from-list*
⟨*proof*⟩

lemmas *hlg-nodes-it-impl* = *hlg-defs.gbm-nodes-it-correct*[
folded hlg-defs,
unfolded hm-ops-def, *simplified*, *OF hm-iteratei-impl*, *folded hlg-defs*
]
interpretation *hlg*!: *graph-nodes-it hlg-α hlg-invar hlg-nodes-it*
⟨*proof*⟩

lemmas *hlg-edges-it-impl* = *hlg-defs.gbm-edges-it-correct*[*folded hlg-defs*,
unfolded hm-ops-def ls-ops-def, *simplified*,
OF hm-iteratei-impl hm-iteratei-impl ls-iteratei-impl, *folded hlg-defs*]
interpretation *hlg*!: *graph-edges-it hlg-α hlg-invar hlg-edges-it*
⟨*proof*⟩

lemmas *hlg-succ-it-impl* = *hlg-defs.gbm-succ-it-correct*[*of hm-iteratei ls-iteratei*,
folded hlg-defs, *unfolded hm-ops-def ls-ops-def*, *simplified*,
OF hm-iteratei-impl ls-iteratei-impl]
interpretation *hlg*!: *graph-succ-it hlg-α hlg-invar hlg-succ-it*
⟨*proof*⟩

lemmas *hlg-to-list-impl* = *gga-to-list-correct*[*OF*
hlg-nodes-it-impl hlg-edges-it-impl, *simplified hlg-defs*[*symmetric*]]
interpretation *hlg*!: *graph-to-list hlg-α hlg-invar hlg-to-list*
⟨*proof*⟩

definition *hlg-ops* ≡ (|
gop-α = *hlg-α*,
gop-invar = *hlg-invar*,
gop-empty = *hlg-empty*,
gop-add-node = *hlg-add-node*,
gop-delete-node = *hlg-delete-node*,
gop-add-edge = *hlg-add-edge*,
gop-delete-edge = *hlg-delete-edge*,
gop-from-list = *hlg-from-list*,
gop-to-list = *hlg-to-list*
|)

lemma *hlg-ops-unfold*[*code-unfold*]:
gop-α hlg-ops = *hlg-α*
gop-invar hlg-ops = *hlg-invar*
gop-empty hlg-ops = *hlg-empty*
gop-add-node hlg-ops = *hlg-add-node*
gop-delete-node hlg-ops = *hlg-delete-node*
gop-add-edge hlg-ops = *hlg-add-edge*
gop-delete-edge hlg-ops = *hlg-delete-edge*
gop-from-list hlg-ops = *hlg-from-list*
gop-to-list hlg-ops = *hlg-to-list*
⟨*proof*⟩

lemma *hlgr-impl*: *StdGraph hlg-ops*
⟨*proof*⟩

interpretation *hlgr*!: *StdGraph hlg-ops* ⟨*proof*⟩

end

10 Implementation of Dijkstra's-Algorithm using the ICF

theory *Dijkstra-Impl*
imports *Dijkstra GraphSpec HashGraphImpl* ~~/*src*/*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.

Due to idiosyncrasies of the code generator, we have to split the locale for the definitions, and the locale that assumes the preconditions.

locale *dijkstraC-def* =
g!: *StdGraphDefs g-ops* +
mr!: *StdMapDefs mr-ops* +
qw!: *StdUprioDefs 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
  and nodes-it :: ('V, 'W, 'qw, 'G) graph-nodes-it
  and succ-it :: ('V, 'W, ('qw×'mr), 'G) graph-succ-it
+
  fixes v0 :: 'V
  fixes ga :: ('V,'W) graph
  fixes g :: 'G
begin
  definition αsc == map-pair qw.α mr.α
  definition dinvarC-add == λ(wl,res). qw.invar wl ∧ mr.invar res

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

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

end
```

locale *dijkstraC* =
dijkstraC-def g-ops mr-ops qw-ops nodes-it succ-it v0 ga g +
Dijkstra ga v0 +
g!: *StdGraph g-ops* +
mr!: *StdMap mr-ops* +
qw!: *StdUprio qw-ops* +
g!: *graph-nodes-it g.α g.invar nodes-it* +
g!: *graph-succ-it g.α g.invar succ-it*
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 *nodes-it* :: ('*V*, '*W*, '*qw*, '*G*) *graph-nodes-it*
and *succ-it* :: ('*V*, '*W*, ('*qw*×'*mr*), '*G*) *graph-succ-it*
and *ga*::('*V*,'*W*) *graph* **and** *V* :: '*V set* **and** *v0*::'*V* **and** *g* :: '*G*+
assumes *in-abs*: *g.α g* = *ga*
assumes *g-invar*[*simp*, *intro*!]: *g.invar g*
begin

schematic-lemma *cdinit-refines*:
notes [*refine*] = *inj-on-id*
shows *cdinit* ≤⇓*?R mdinit*
⟨*proof*⟩

schematic-lemma *cpop-min-refines*:
(*σ*,*σ'*) ∈ *build-rel αsc dinvarC-add*
⟹ *cpop-min σ* ≤ ⇓*?R* (*mpop-min σ'*)
⟨*proof*⟩

schematic-lemma *cupdate-refines*:
notes [*refine*] = *inj-on-id*
shows (*σ*,*σ'*)∈*build-rel αsc dinvarC-add* ⟹ *v*=*v'* ⟹ *wv*=*wv'* ⟹
cupdate v wv σ ≤ ⇓*?R* (*mupdate v' wv' σ'*)
⟨*proof*⟩

lemma *cdijkstra-refines*: *cdijkstra* ≤ ⇓(*build-rel mr.α mr.invar*) *mdijkstra*
⟨*proof*⟩

schematic-lemma *idijkstra-refines-aux*:
notes [*refine-transfer*] = *g-invar*
shows *RETURN ?f* ≤ *cdijkstra*
⟨*proof*⟩

thm *idijkstra-refines-aux*[*no-vars*]

end

Copy-Pasted from the above lemma:

definition (**in** *dijkstraC-def*) *idijkstra* ≡
(*let x* = *let x* = *nodes-it g* (λ-. *True*)

```
                (λx s. let s = s in upr-insert qw-ops s x infty.Infty)
                (upr-empty qw-ops ())
        in (upr-insert qw-ops x v0 (Num (0::'W)),
            map-op-sng mr-ops v0 ([], 0::'W));
    (a, b) =
      while (λ(wl, -). ¬ upr-isEmpty qw-ops wl)
       (λxa. let (a, b) =
                  let (a, b) = xa; (aa, ba) = upr-pop qw-ops a
                  in case ba of (ab, bb) ⇒ (aa, ab, bb, b)
             in case b of
                (aa, ba) ⇒
                  let (ab, bb) = ba;
                      xd = mpath' (map-op-lookup mr-ops a bb)
                  in succ-it g a (λ-. True)
                      (λx s. let s = s
                             in case x of
                                (ac, bc) ⇒
                                  case s of
                                  (ad, bd) ⇒
if aa + Num ac < mpath-weight' (map-op-lookup mr-ops bc bd)
then (upr-insert qw-ops ad bc (aa + Num ac),
      map-op-update mr-ops bc ((a, ac, bc) # the xd, val aa + ac) bd)
else (ad, bd))
                      (ab, bb))
        x
  in b)
```

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

```
definition dijkstra-impl ::
  ('V::{hashable,linorder},'W::weight) hlg ⇒ 'V
  ⇒ ('V,('V,'W) path × 'W) rm where
  dijkstra-impl g v0 ≡
    dijkstraC-def.idijkstra rm-ops aluprioi-ops hlg-nodes-it hlg-succ-it v0 g

lemmas [code] = dijkstraC-def.idijkstra-def
```

Code can be exported to all available target languages:

```
export-code dijkstra-impl in SML file –
export-code dijkstra-impl in OCaml file –
export-code dijkstra-impl in Haskell file –
export-code dijkstra-impl in Scala file –


context dijkstraC
begin
  lemma idijkstra-refines: RETURN idijkstra ≤ cdijkstra
    ⟨proof⟩
```

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*) *idijkstra-correct*:
shows
weighted-graph.is-shortest-path-map ga v0 (αr (*mr.*α *idijkstra*)) (**is** *?G1*)
and *mr.invar idijkstra* (**is** *?G2*)
and *res-invarm* (*mr.*α *idijkstra*) (**is** *?G3*)
⟨*proof*⟩

end

We also specialize the correctness theorem. Note that the data structure invariant for red-black trees is encoded into the type, and hence λ-. *True* is constantly *True*. Hence, it is not stated in this theorem.

theorem *dijkstra-impl-correct*:
assumes *INV*: *hlg-invar g*
assumes *V0*: *v0* ∈ *nodes* (*hlg-*α *g*)
assumes *nonneg-weights*: $\bigwedge v\ w\ v'.\ (v,w,v') \in edges\ (hlg\text{-}\alpha\ g) \Longrightarrow 0 \leq w$
shows
weighted-graph.is-shortest-path-map (*hlg-*α *g*) *v0*
(*Dijkstra.*αr (*rm-*α (*dijkstra-impl g v0*))) (**is** *?G1*)
and *Dijkstra.res-invarm* (*rm-*α (*dijkstra-impl g v0*)) (**is** *?G2*)
⟨*proof*⟩

end

11 Implementation of Dijkstra's-Algorithm using Automatic Determinization

theory *Dijkstra-Impl-Adet*
imports *Dijkstra GraphSpec HashGraphImpl ~~/src/HOL/Library/Code-Target-Numeral*
../Refine-Monadic/Autoref-Collection-Bindings
begin

locale *dijkstraC-def* =
g!: *StdGraphDefs g-ops* +
mr!: *StdMapDefs mr-ops* +
qw!: *StdUprioDefs 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* +
fixes *nodes-it* :: (*'V*, *'W*, *'qw*, *'G*) *graph-nodes-it*
fixes *succ-it* :: (*'V*, *'W*, *'qw*×*'mr*, *'G*) *graph-succ-it*
fixes *v0* :: *'V*
fixes *ga* :: (*'V*,*'W*) *graph*
fixes *g* :: *'G*
begin
end

locale *dijkstraC* =
dijkstraC-def g-ops mr-ops qw-ops nodes-it succ-it v0 ga g +
Dijkstra ga v0 +
g!: *StdGraph g-ops* +
mr!: *StdMap mr-ops* +
qw!: *StdUprio qw-ops* +
g!: *graph-nodes-it g.α g.invar nodes-it* +
g!: *graph-succ-it g.α g.invar succ-it*
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 *nodes-it* :: (*'V*, *'W*, *'qw*, *'G*) *graph-nodes-it*
and *succ-it* :: (*'V*, *'W*, *'qw*×*'mr*, *'G*) *graph-succ-it*
and *ga*::(*'V*,*'W*) *graph* **and** *V* :: *'V set* **and** *v0*::*'V* **and** *g* :: *'G*+
assumes *in-abs*: *g.α g* = *ga*
assumes *g-invar*[*simp*, *intro*!]: *g.invar g*
begin

lemma *ga-trans*: (*g*,*ga*)∈*br g.α g.invar* ⟨*proof*⟩

lemma *ga-nodes-trans*: (*nodes* (*g.α g*),*nodes ga*)∈*Id* ⟨*proof*⟩
lemma *ga-succs-trans*: (*v*,*v'*)∈*Id* ⟹ (*succ* (*g.α g*) *v*,*succ ga v'*)∈*Id*
⟨*proof*⟩

schematic-lemma *cdijkstra-refines-aux*:
notes [*autoref-spec*] =
spec-R[**where** *'c*=*bool* **and** *'a*=*bool*]
spec-R[**where** *'c*=*'V* **and** *'a*=*'V*]
spec-R[**where** *'c*=*'W* **and** *'a*=*'W*]
spec-R[**where** *'c*=*'W infty* **and** *'a*=*'W infty*]

spec-R[**where** *'c*=*'mr* **and** *'a*=*'V*⇀((*'V*,*'W*) *path* × *'W*)]
spec-R[**where** *'c*=*'qw* **and** *'a*=*'V*⇀*'W infty*]
spec-R[**where** *'c*=(*'V*,*'W*) *path* **and** *'a*=(*'V*,*'W*) *path*]
spec-R[**where** *'c*=((*'V*,*'W*) *path* × *'W*) **and** *'a*=((*'V*,*'W*) *path* × *'W*)]
spec-R[**where** *'c*=(*'V*,*'W*) *path option* **and** *'a*=(*'V*,*'W*) *path option*]
spec-R[**where** *'c*=((*'V*,*'W*) *path* × *'W*) *option* **and**

```
    'a=(('V,'W) path × 'W) option]

  notes [autoref-ex] = mr.map-lookup-t ga-succs-trans


  shows RETURN (?cdijkstra::'mr) ≤⇓?R  mdijkstra
  ⟨proof⟩

 notepad begin
  ⟨proof⟩
  thm cdijkstra-refines-aux
 end
end

definition (in dijkstraC-def) cdijkstra ≡
 (let (a::'qw, b::'mr) =
      let x::'qw =
           (nodes-it::'G ⇒ ('qw ⇒ bool) ⇒ ('V ⇒ 'qw ⇒ 'qw) ⇒ 'qw ⇒ 'qw)
            (g::'G) (λ-::'qw. True)
            (λ(x::'V) s::'qw.
               let s::'qw = s
               in upr-insert
                   (qw-ops::('V, 'W infty, 'qw,
                             'more-qw) uprio-ops-scheme)
                   s x infty.Infty)
            (upr-empty qw-ops ())
      in (upr-insert qw-ops x (v0::'V) (Num (0::'W)),
          map-op-update
           (mr-ops::('V, ('V × 'W × 'V) list × 'W, 'mr,
                     'more-mr) map-ops-scheme)
           v0 ([], 0::'W) (map-op-empty mr-ops ()))
  in Let (while ((λa::'qw. ¬ upr-isEmpty qw-ops a) ∘ fst)
         (λ(aa::'qw, ba::'mr).
            let (ab::'V, bb::'W infty × 'qw × 'mr) =
                 let (ab::'qw, bb::'mr) = (aa, ba);
                     (ac::'V, bc::'W infty × 'qw) = upr-pop qw-ops ab
                 in case bc of (ad::'W infty, bd::'qw) ⇒ (ac, ad, bd, bb)
            in case bb of
               (ac::'W infty, ad::'qw, bd::'mr) ⇒
                 let (ae::'qw, be::'mr) = (ad, bd);
                     xd::('V × 'W × 'V) list option =
                       mpath' (map-op-lookup mr-ops ab be)
                 in (succ-it::'G ⇒ 'V ⇒ ('qw × 'mr ⇒ bool)
    ⇒ ('W × 'V ⇒ 'qw × 'mr ⇒ 'qw × 'mr) ⇒ 'qw × 'mr ⇒ 'qw × 'mr)
                     g ab (λ-::'qw × 'mr. True)
                     (λ(x::'W × 'V) s::'qw × 'mr.
                        let (af::'qw, bf::'mr) = s
                        in case x of
                           (ag::'W, bg::'V) ⇒
```

```
                                if ac + Num ag
                                  < mpath-weight'
(map-op-lookup mr-ops bg bf)
                                then (upr-insert qw-ops af bg
 (ac + Num ag),
map-op-update mr-ops bg ((ab, ag, bg) # the xd, val ac + ag) bf)
                                else (af, bf))
                        (ae, be))
          (a, b))
     ((λba::'mr. ba) ∘ snd))
```

context *dijkstraC*
begin
lemma *cdijkstra-refines*:
RETURN cdijkstra ≤ ⇓(*build-rel mr.α mr.invar*) *mdijkstra*
⟨*proof*⟩

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 *cdijkstra-correct*:
shows
weighted-graph.is-shortest-path-map ga v0 (*αr* (*mr.α cdijkstra*)) (**is** *?G1*)
and *mr.invar cdijkstra* (**is** *?G2*)
and *res-invarm* (*mr.α cdijkstra*) (**is** *?G3*)
⟨*proof*⟩

end

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

definition *dijkstra-impl* ::
('*V*::{*hashable*,*linorder*},'*W*::*weight*) *hlg* ⇒ '*V*
⇒ ('*V*,('*V*,'*W*) *path* × '*W*) *rm* **where**
dijkstra-impl g v0 ≡
dijkstraC-def.cdijkstra rm-ops aluprioi-ops hlg-nodes-it hlg-succ-it v0 g

We also specialize the correctness theorem. Note that the data structure invariant for red-black trees is encoded into the type, and hence λ-. *True* is constantly *True*. Hence, it is not stated in this theorem.

theorem *dijkstra-impl-correct*:
assumes *INV*: *hlg-invar g*
assumes *V0*: *v0* ∈ *nodes* (*hlg-α g*)
assumes *nonneg-weights*: ⋀*v w v'*. (*v*,*w*,*v'*)∈*edges* (*hlg-α g*) ⟹ *0*≤*w*
shows
weighted-graph.is-shortest-path-map (*hlg-α g*) *v0*
(*Dijkstra.αr* (*rm-α* (*dijkstra-impl g v0*))) (**is** *?G1*)
and *Dijkstra.res-invarm* (*rm-α* (*dijkstra-impl g v0*)) (**is** *?G2*)
⟨*proof*⟩

lemmas [*code*] = *dijkstraC-def.cdijkstra-def*

Code can be exported to all available target languages:

export-code *dijkstra-impl* **in** *SML* **file** –
export-code *dijkstra-impl* **in** *OCaml* **file** –
export-code *dijkstra-impl* **in** *Haskell* **file** –
export-code *dijkstra-impl* **in** *Scala* **file** –

end

12 Performance Test

theory *Test*
imports *Dijkstra-Impl*
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*) *mod 268435456*

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

definition *ran-graph* :: *nat* ⇒ *nat* ⇒ (*nat list*×(*nat* × *nat* × *nat*) *list*) **where**
ran-graph vertices seed ==
([*0*::*nat*..<*vertices*],*fst*
(*while* (λ (*g*,*v*,*s*). *v* < *vertices*)
(λ (*g*,*v*,*s*).
let (*g''*,*v''*,*s''*) = (*while* (λ (*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 ≡ *dijkstra-impl*

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* (*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://afp.sf.net/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 Proving*, 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://afp.sf.net/entries/Tree-Automata.shtml, Oct. 2010. Formal proof development.