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

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

  lemma *least-map-subset*: $least\text{-}map\ f\ S \subseteq S$
    by (*auto elim*: *least-map.cases*)

  lemmas *least-map-elemD* = *subsetD*[*OF least-map-subset*]

  lemma *least-map-leD*:
    assumes $x \in least\text{-}map\ f\ S$
    assumes $y{\in}S$
    shows $f\ x \le f\ y$
    using *assms*

by (*auto elim*: *least-map.cases*)

lemma *least-map-empty*[*simp*]: *least-map f* {} = {}
by (*auto elim*: *least-map.cases*)

lemma *least-map-singleton*[*simp*]: *least-map* (*f*::'*a*⇒'*b*::*order*) {*x*} = {*x*}
by (*auto elim*: *least-map.cases intro*!: *least-map.intros simp*: *refl*)

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*)
using *assms* **by** (*auto intro*: *least-map.intros*)

lemma *least-map-insert-nmin*:
⟦ *x*∈*least-map f S*; *f x* ≤ *f a* ⟧ ⟹ *x*∈*least-map f* (*insert a S*)
by (*auto elim*: *least-map.cases intro*: *least-map.intros*)

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*
apply (*metis le-iff-inf less-imp-le*)
by (*metis inf-commute le-iff-inf less-imp-le*)
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
    apply –
    apply (rule subsetD[OF E-valid(1)])
    using assms apply force
    apply (rule subsetD[OF E-valid(2)])
    using assms apply force
    done

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

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*
unfolding *empty-def* **by** *unfold-locales auto*
lemma *add-node-valid*[*simp*]: **assumes** *valid-graph g*
shows *valid-graph* (*add-node v g*)
proof −
interpret *valid-graph g* **by** *fact*
show *?thesis*
unfolding *add-node-def*
by *unfold-locales* (*auto dest*: *E-validD*)
qed
lemma *delete-node-valid*[*simp*]: **assumes** *valid-graph g*
shows *valid-graph* (*delete-node v g*)
proof −
interpret *valid-graph g* **by** *fact*
show *?thesis*
unfolding *delete-node-def*
by *unfold-locales* (*auto dest*: *E-validD*)
qed
lemma *add-edge-valid*[*simp*]: **assumes** *valid-graph g*
shows *valid-graph* (*add-edge v e v′ g*)
proof −
interpret *valid-graph g* **by** *fact*
show *?thesis*
unfolding *add-edge-def*
by *unfold-locales* (*auto dest*: *E-validD*)
qed
lemma *delete-edge-valid*[*simp*]: **assumes** *valid-graph g*
shows *valid-graph* (*delete-edge v e v′ g*)
proof −
interpret *valid-graph g* **by** *fact*
show *?thesis*
unfolding *delete-edge-def*
by *unfold-locales* (*auto dest*: *E-validD*)
qed

lemma *succ-finite*[*simp*, *intro*]: $finite\ (edges\ G) \Longrightarrow finite\ (succ\ G\ v)$
unfolding *succ-def*
by (*rule finite-subset*[**where** $B=snd`edges\ G$]) *force+*

lemma *nodes-empty*[*simp*]: *nodes empty* = {} **unfolding** *empty-def* **by** *simp*
lemma *edges-empty*[*simp*]: *edges empty* = {} **unfolding** *empty-def* **by** *simp*
lemma *succ-empty*[*simp*]: *succ empty v* = {} **unfolding** *empty-def succ-def* **by** *auto*

lemma *nodes-add-node*[*simp*]: *nodes* (*add-node v g*) = *insert v* (*nodes g*)
by (*simp add*: *add-node-def*)
lemma *nodes-add-edge*[*simp*]:
nodes (*add-edge v e v′ g*) = *insert v* (*insert v′* (*nodes g*))
by (*simp add*: *add-edge-def*)
lemma *edges-add-edge*[*simp*]:
edges (*add-edge v e v′ g*) = *insert* (*v*,*e*,*v′*) (*edges g*)
by (*simp add*: *add-edge-def*)
lemma *edges-add-node*[*simp*]:
edges (*add-node v g*) = *edges g*
by (*simp add*: *add-node-def*)

lemma (**in** *valid-graph*) *succ-subset*: *succ G v* ⊆ *UNIV* × *V*
unfolding *succ-def* **using** *E-valid*
by (*force*)

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* ⇒ (′*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*
by (*auto dest*: *E-validD*)

lemma *is-path-memb*[*simp*]:
is-path v p v′ ⟹ *v*∈*V* ∧ *v′*∈*V*
apply (*induct p arbitrary*: *v*)
apply (*auto dest*: *E-validD*)
done

lemma *is-path-split*:
is-path v (*p1*@*p2*) *v′* ⟷ (∃ *u*. *is-path v p1 u* ∧ *is-path u p2 v′*)
by (*induct p1 arbitrary*: *v*) *auto*

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'*
by (*auto simp add*: *is-path-split*)
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*
by (*auto simp add*: *int-vertices-def*)

lemma (**in** *valid-graph*) *int-vertices-subset*:
is-path v p v' ⟹ *int-vertices p* ⊆ *V*
apply (*induct p arbitrary*: *v*)
apply (*simp*)
apply (*force dest*: *E-validD*)
done

lemma *int-vertices-empty*[*simp*]: *int-vertices p* = {} ⟷ *p*=[]
by (*cases p*) *auto*

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*∈*W* **and** *v'*∉*W*
obtains *p1 p2 u w u'* **where**
p=*p1*@(*u*,*w*,*u'*)#*p2* **and**
int-vertices p1 ⊆ *W* **and** *u*∈*W* **and** *u'*∉*W*
using *assms*
proof (*induct p arbitrary*: *v thesis*)
case *Nil* **thus** *?case* **by** *auto*
next
case (*Cons vv p*)
note [*simp*, *intro*!] = ‹*v*∈*W*› ‹*v'*∉*W*›
from *Cons.prems* **obtain** *w u'* **where**
[*simp*]: *vv*=(*v*,*w*,*u'*) **and**
REST: *is-path u' p v'*
by (*cases vv*) *auto*

Distinguish wether the second node u' of the path is in W. If yes, the proposition follows by the induction hypothesis, otherwise it is straightforward, as the split takes place at the first edge of the path.

```
    {
      assume A [simp, intro!]: u'∈W
      from Cons.hyps[OF - REST] obtain p1 uu ww uu' p2 where
        p=p1@(uu,ww,uu')#p2 int-vertices p1 ⊆ W uu ∈ W uu' ∉ W
        by blast
      with Cons.prems(1)[of vv#p1 uu ww uu' p2] have thesis by auto
    } moreover {
      assume u'∉W
      with Cons.prems(1)[of [] v w u' p] have thesis by auto
    } ultimately show thesis by blast
  qed
```

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'∈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 ⊆ −W and u∈W
    using assms
  proof (cases v∈W)
    case True with that[of [] p] assms show ?thesis
      by auto
  next
    case False with assms that show ?thesis
    proof (induct p arbitrary: v thesis)
      case Nil thus ?case by auto
    next
      case (Cons vv p)
      note [simp, intro!] = ‹v'∈W› ‹v∉W›
      from Cons.prems obtain w u' where
        [simp]: vv=(v,w,u') and [simp]: (v,w,u')∈E and
          REST: is-path u' p v'
        by (cases vv) auto
```

Distinguish wether the second node u' of the path is in W. If yes, the proposition is straightforward, otherwise, it follows by the induction hypothesis.

```
      {
        assume A [simp, intro!]: u'∈W
        from Cons.prems(3)[of [vv] p u'] REST have ?case by auto
      } moreover {
        assume [simp, intro!]: u'∉W
        from Cons.hyps[OF REST] obtain p1 p2 u'' where
          [simp]: p=p1@p2 and
            is-path u' p1 u'' and
            is-path u'' p2 v' and
```

int-vertices p1 $\subseteq -W$ **and**
$u'' \in W$ **by** *blast*
with *Cons.prems*(*3*)[*of vv#p1*] **have** *?case* **by** *auto*
} **ultimately show** *?case* **by** *blast*
qed
qed

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∈int-vertices p*
obtains *p1 p2* **where**
p=p1@p2 **and**
is-path v p1 u **and**
u ∉ int-vertices p1
using *assms*
proof (*induct p arbitrary*: *v thesis*)
case *Nil* **thus** *?case* **by** *auto*
next
case (*Cons vv p*)
from *Cons.prems* **obtain** *w u′* **where**
[*simp*]: *vv=*(*v*,*w*,*u′*) *v∈V* (*v*,*w*,*u′*)*∈E* **and**
REST: *is-path u′ p v′*
by (*cases vv*) *auto*

{
assume *u=v*
with *Cons.prems*(*1*)[*of* [] *vv#p*] **have** *thesis* **by** *auto*
} **moreover** **{**
assume [*simp*]: *u≠v*
with *Cons.hyps*(*1*)[*OF - REST*] *Cons.prems*(*3*) **obtain** *p1 p2* **where**
p=p1@p2 is-path u′ p1 u u∉int-vertices p1
by *auto*
with *Cons.prems*(*1*)[*of vv#p1 p2*] **have** *thesis*
by *auto*
} **ultimately show** *?case* **by** *blast*
qed

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* ≡ *sum-list* (*map* (*fst* ∘ *snd*) *p*)

lemma *path-weight-split*[*simp*]:
(*path-weight* (*p1@p2*)::*′w*::*monoid-add*) = *path-weight p1* + *path-weight p2*
unfolding *path-weight-def*

by (*auto*)

lemma *path-weight-empty*[*simp*]: *path-weight* [] = *0*
unfolding *path-weight-def*
by *auto*

lemma *path-weight-cons*[*simp*]:
(*path-weight* (*e#p*)::'*w*::*monoid-add*) = *fst* (*snd e*) + *path-weight p*
unfolding *path-weight-def*
by (*auto*)

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 −
have $0 + 0 \leq a + b$
using *assms* **by** (*rule add-mono*)
then show *?thesis* **by** *simp*
qed

lemma *add-nonpos-nonpos*[*simp*]:
assumes $a \leq 0$ **and** $b \leq 0$ **shows** $a + b \leq 0$
proof −
have $a + b \leq 0 + 0$
using *assms* **by** (*rule add-mono*)
then show *?thesis* **by** *simp*
qed

lemma *add-nonneg-eq-0-iff*:
assumes *x*: $0 \leq x$ **and** *y*: $0 \leq y$

shows $x + y = 0 \longleftrightarrow x = 0 \wedge y = 0$
by (*metis local.add-0-left local.add-0-right local.add-left-mono local.antisym-conv x y*)

lemma *add-incr*: $0 \le b \Longrightarrow a \le a+b$
by (*metis add.comm-neutral add-left-mono*)

lemma *add-incr-left*[*simp*, *intro*!]: $0 \le b \Longrightarrow a \le b + a$
by (*metis add-incr add.commute*)

lemma *sum-not-less*[*simp*, *intro*!]:
$0 \le b \Longrightarrow \neg (a+b < a)$
$0 \le a \Longrightarrow \neg (a+b < b)$
apply (*metis add-incr less-le-not-le*)
apply (*metis add-incr-left less-le-not-le*)
done

end

instance *nat* :: *weight* **..**
instance *int* :: *weight* **..**
instance *rat* :: *weight* **..**
instance *real* :: *weight* **..**

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*
by (*metis add.commute inf-add-right*)

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

lemma *not-less-inf*[*simp*]:
¬ (*a* < *top*) ⟷ *a*=*top*
by *simp*

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*]: $e \neq Infty \Longrightarrow Num\ (val\ e) = e$ **by** (*cases e*) *auto*

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*]: $Infty \leq a \longleftrightarrow a = Infty$
    by (*cases a*) *auto*

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

  lemma [*simp*]: *less a Infty* ⟷ $a \neq Infty$
    by (*cases a*) *auto*

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

  lemma [*simp*]: *plus Infty a* = *Infty* **by** (*cases a*) *simp-all*

  instance
    apply (*intro-classes*)
    apply (*case-tac* [!] *x*) [*4*]
    apply *simp-all*
    apply (*case-tac* [!] *y*) [*3*]
    apply (*simp-all add*: *less-le-not-le*)
    apply (*case-tac z*)
    apply (*simp-all add*: *top-infty-def zero-infty-def*)
    apply (*case-tac* [!] *a*) [*4*]
    apply *simp-all*
    apply (*case-tac* [!] *b*) [*3*]
    apply (*simp-all add*: *ac-simps*)
    apply (*case-tac* [!] *c*) [*2*]
    apply (*simp-all add*: *ac-simps add-right-mono*)
    apply (*case-tac* (*x*,*y*) *rule*: *less-eq-infty.cases*)
    apply (*simp-all add*: *linear*)

done
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 $\neq$ *top*
  top $\neq$ *Num a*
  Infty = *top*
  by (*auto simp add*: *top-infty-def*)

lemma *infty-ord-unbox*:
  Num a $\leq$ *Num b* $\longleftrightarrow$ *a* $\leq$ *b*
  Num a < *Num b* $\longleftrightarrow$ *a* < *b*
  by *auto*

lemma *infty-plus-unbox*:
  Num a + *Num b* = *Num* (*a*+*b*)
  by (*auto*)

lemma *infty-zero-unbox*:
  Num a = *0* $\longleftrightarrow$ *a* = *0*
  Num 0 = *0*
  by (*auto simp*: *zero-infty-def*)

lemmas *infty-unbox* =
  infty-inf-unbox infty-zero-unbox infty-ord-unbox infty-plus-unbox

lemma *inf-not-zero*[*simp*]:
  top$\neq$(*0*::- *infty*) (*0*::- *infty*)$\neq$*top*
  apply (*unfold zero-infty-def top-infty-def*)
  apply *auto*
  done

lemma *num-val-iff'*[*simp*]: *e*$\neq$*top* $\Longrightarrow$ *Num* (*val e*) = *e*
  by (*cases e*) (*auto simp add*: *infty-unbox*)

lemma *infty-neE*:
  $[\![$*a*$\neq$*Infty*; $\bigwedge$*d*. *a*=*Num d* $\Longrightarrow$ *P*$]\!]$ $\Longrightarrow$ *P*
  $[\![$*a*$\neq$*top*; $\bigwedge$*d*. *a*=*Num d* $\Longrightarrow$ *P*$]\!]$ $\Longrightarrow$ *P*
  by (*case-tac* [!] *a*) (*auto simp add*: *infty-unbox*)

end

5 Dijkstra's Algorithm

theory *Dijkstra*
  imports
  Graph
  Dijkstra-Misc
  Collections.Refine-Dflt-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* $:: 'V \Rightarrow ('V \rightharpoonup ('V,'W)\ path) \Rightarrow bool$
  where
  $is\text{-}shortest\text{-}path\text{-}map\ v0\ res \equiv \forall\, v{\in}V.\ (case\ res\ v\ of$
    $None \Rightarrow \neg(\exists\, p.\ is\text{-}path\ v0\ p\ v)\ |$
    $Some\ p \Rightarrow is\text{-}path\ v0\ p\ v$
      $\wedge\ (\forall\, p'.\ is\text{-}path\ v0\ p'\ v \longrightarrow path\text{-}weight\ p \leq path\text{-}weight\ p')$
  $)$
end

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

fun *path-weight'* **where**
  $path\text{-}weight'\ None = top\ |$
  $path\text{-}weight'\ (Some\ p) = Num\ (path\text{-}weight\ p)$

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
  by (*induct rule*: *is-path.induct*) *auto*

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* $(\{\},res) \Longrightarrow$ *is-shortest-path-map v0 res*

unfolding *dinvar-def is-shortest-path-map-def*
by (*auto simp*: *infty-unbox split*: *option.split*)

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*
unfolding *dinit-def*
apply (*intro refine-vcg*)
apply (*force simp*: *dinvar-def split*: *option.split*)
done

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

Finally, we define Dijkstra's algorithm:

definition *dijkstra* **where**
dijkstra $\equiv$ *do* {
$\sigma 0 \leftarrow$ *dinit*;
(-,*res*) $\leftarrow$ $WHILE_T{}^{dinvar}$ (λ(*wl*,-). *wl*$\neq${})
($\lambda\sigma$.
do { (*v*,σ') $\leftarrow$ *pop-min* σ; *update v* σ' }
)
$\sigma 0$;
RETURN res }

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

theorem *dijkstra-correct*: *dijkstra* $\leq$ *SPEC* (*is-shortest-path-map v0*)
unfolding *dijkstra-def*
unfolding *dinit-def*
unfolding *pop-min-def update-def* [*abs-def*]
thm *refine-vcg*

apply (*refine-rcg*
WHILEIT-rule[**where** *R*=*inv-image* {(*x*,*y*). *x*<*y*} (*card* $\circ$ *fst*)]
refine-vcg
)

supply [[*simproc del*: *defined-all*]]
apply (*simp-all split*: *prod.split-asm*)
apply (*tactic* ‹
ALLGOALS ((*REPEAT-DETERM o Hypsubst.bound-hyp-subst-tac* @{*context*})
THEN′ asm-full-simp-tac @{*context*}
)›)

proof −
fix *wl res v*
assume *INV*: *dinvar* (*wl*,*res*)
and *LM*: *v*$\in$*least-map* (*path-weight′* $\circ$ *res*) *wl*
hence *v*$\in$*V* **unfolding** *dinvar-def* **by** (*auto dest*: *least-map-elemD*)
moreover
from *INV* **have** $\forall v'\in V - (wl-\{v\}).\ v' \neq v \longrightarrow$
($\forall$ *p*. *is-path v0 p v′* $\longrightarrow$ *path-weight′* (*res v′*) $\leq$ *Num* (*path-weight p*))
by (*auto simp*: *dinvar-def*)
moreover from *INV* **have** $\forall v'\in V.\ \forall p.$ *res v′*=*Some p* $\longrightarrow$ *is-path v0 p v′*
by (*auto simp*: *dinvar-def*)
ultimately show *update-pre v* (*wl*−{*v*},*res*) **by** (*auto simp*: *update-pre-def*)
next

```
  fix res
  assume dinvar ({}, res)
  thus is-shortest-path-map v0 res
    by (rule invar-imp-correct)
next
  show wf (inv-image {(x, y). x < y} (card ∘ fst))
    by (blast intro: wf-less)
next
  fix wl res v σ''
  assume
    LM: v∈least-map (path-weight' ∘ res) wl and
    UD: update-spec v (wl−{v},res) σ'' and
    INV: dinvar (wl,res)

  from LM have v∈wl by (auto dest: least-map-elemD)
  moreover from UD have fst σ'' = wl−{v} by (auto elim: update-spec.cases)
  moreover from INV have finite wl
    unfolding dinvar-def by (auto dest: finite-subset)
  ultimately show card (fst σ'') < card wl
    apply simp
    by (metis card-gt-0-iff diff-Suc-less empty-iff)
next
  fix a and res :: 'V ⇀ ('V,'W) path
  assume a = V ∧ res v0 = Some [] ∧ (∀ v∈V−{v0}. res v = None)
  thus dinvar (V,res)
    by (force simp: dinvar-def split: option.split)
next
  fix wl res
  assume INV: dinvar (wl,res)
  hence
    WL-SUBSET: wl ⊆ V and
    PATH-VALID: ∀ v∈V. ∀ p. res v = Some p
      ⟶ is-path v0 p v ∧ int-vertices p ⊆ V − wl and
    NWL-MIN: ∀ v∈V − wl. ∀ p. is-path v0 p v
      ⟶ path-weight' (res v) ≤ Num (path-weight p) and
    WL-MIN: ∀ v∈wl. ∀ p. is-path v0 p v ∧ int-vertices p ⊆ V − wl
      ⟶ path-weight' (res v) ≤ Num (path-weight p)
    unfolding dinvar-def by auto

  fix v σ''
  assume V-LEAST: v∈least-map (path-weight' o res) wl
    and update-spec v (wl−{v},res) σ''
  then obtain res' where
    [simp]: σ''=(wl−{v},res')
    and CONSIDERED-NEW-PATHS: ∀ v'∈V. res' v' ∈ least-map path-weight'
      (insert (res v')
          ({ Some (p@[(v,w,v')]) | p w. res v = Some p ∧ (v,w,v')∈E }))
    by (auto elim!: update-spec.cases)
```

from *V-LEAST* **have** *V-MEM*: $v \in wl$ **by** (*blast intro*: *least-map-elemD*)

show *dinvar* σ''
  apply (*unfold dinvar-def*, *simp*)
  apply (*intro conjI*)
proof −
  from *WL-SUBSET* **show** $wl - \{v\} \subseteq V$ **by** *auto*

  show $\forall\, va \in V.\ \forall\, p.\ res'\ va = Some\ p$
    $\longrightarrow is\text{-}path\ v0\ p\ va \wedge int\text{-}vertices\ p \subseteq V - (wl - \{v\})$
  proof (*intro ballI conjI impI allI*)
    fix v' *p*
    assume *V'-MEM*: $v' \in V$ **and** [*simp*]: $res'\ v' = Some\ p$

The new paths that we have added are valid and only use intermediate vertices outside the workset.

This proof works as follows: A path $res'\ v'$ is either the old path, or has been assembled as a path over node *v*. In the former case the proposition follows straightforwardly from the invariant for the old state. In the latter case we get, by the invariant for the old state, that the path over node *v* is valid. Then, we observe that appending an edge to a valid path yields a valid path again. Also, adding *v* as intermediate node is legal, as we just removed *v* from the workset.

    with *CONSIDERED-NEW-PATHS* **have** $res'\ v' \in (insert\ (res\ v')$
      $(\{\ Some\ (p@[(v,w,v')])\ |\ p\ w.\ res\ v = Some\ p \wedge (v,w,v') \in E\ \}))$
      by (*rule-tac least-map-elemD*) *blast*
    moreover {
      assume [*symmetric*,*simp*]: $res'\ v' = res\ v'$
      from *V'-MEM PATH-VALID* **have**
        $is\text{-}path\ v0\ p\ v'$
        $int\text{-}vertices\ p \subseteq V - (wl - \{v\})$
        by *force+*
    } moreover {
      fix *pv w*
      assume $res'\ v' = Some\ (pv@[(v,w,v')])$
        and [*simp*]: $res\ v = Some\ pv$
        and *EDGE*: $(v,w,v') \in E$
      hence [*simp*]: $p = pv@[(v,w,v')]$ **by** *simp*

        from *bspec*[*OF PATH-VALID rev-subsetD*[*OF V-MEM WL-SUBSET*]]
have
        PATHV: $is\text{-}path\ v0\ pv\ v$ **and** *IVV*: $int\text{-}vertices\ pv \subseteq V - wl$ **by** *auto*
      hence
        $is\text{-}path\ v0\ p\ v'$
        $int\text{-}vertices\ p \subseteq V - (wl - \{v\})$
        by (*auto simp*: *EDGE V'-MEM*)
    }
    ultimately show
      $is\text{-}path\ v0\ p\ v'$
      $int\text{-}vertices\ p \subseteq V - (wl - \{v\})$

```
    by blast+
  qed
```

We show that already the *original* result stores the minimal path for all vertices not in the *new* workset. For vertices also not in the original workset, this follows straightforwardly from the invariant.

For the vertex v, that has been removed from the workset, we split a path p' to v at the point u where it first enters the original workset.

As we chose v to be the vertex in the workset with the minimal weight, its weight is less than the current weight of u. As the vertices of the prefix of p' up to u are not in the workset, the current weight of u is less than the weight of the prefix of p', and thus less than the weight of p'. Together, the current weight of v is less than the weight of p'.

```
  have RES-MIN: ∀ v∈V − (wl − {v}). ∀ p. is-path v0 p v
    ⟶ path-weight' (res v) ≤ Num (path-weight p)
  proof (intro ballI allI impI)
    fix v' p'
    assume NOT-IN-WL: v' ∈ V − (wl − {v})
      and PATH: is-path v0 p' v'
    hence [simp, intro!]: v'∈V by auto

    show path-weight' (res v') ≤ Num (path-weight p')
    proof (cases v' = v)
      assume NE[simp]: v'≠v
      from bspec[OF NWL-MIN, of v'] NOT-IN-WL PATH show
        path-weight' (res v') ≤ Num (path-weight p') by auto
    next
      assume EQ[simp]: v'=v

      from path-split-set'[OF PATH, of wl] V-MEM obtain p1 p2 u where
        [simp]: p'=p1@p2
          and P1: is-path v0 p1 u
          and P2: is-path u p2 v'
          and P1V: int-vertices p1 ⊆ −wl
          and [simp]: u∈wl
        by auto

      from least-map-leD[OF V-LEAST]
      have path-weight' (res v') ≤ path-weight' (res u)by auto
      also from bspec[OF WL-MIN, of u] P1 P1V int-vertices-subset[OF P1]
      have path-weight' (res u) ≤ Num (path-weight p1) by auto
      also have ... ≤ Num (path-weight p')
        using path-nonneg-weight[OF P2]
        apply (auto simp: infty-unbox )
        by (metis add-0-right add-left-mono)
      finally show ?thesis .
    qed
  qed
```

With the previous statement, we easily show the third part of the invariant, as the new paths are not longer than the old ones.

```
show ∀ v∈V − (wl − {v}). ∀ p. is-path v0 p v
  ⟶ path-weight′ (res′ v) ≤ Num (path-weight p)
proof (intro allI ballI impI)
  fix v′ p
  assume NOT-IN-WL: v′ ∈ V − (wl − {v})
    and PATH: is-path v0 p v′
  hence [simp, intro!]: v′∈V by auto
  from bspec[OF CONSIDERED-NEW-PATHS, of v′]
  have path-weight′ (res′ v′) ≤ path-weight′ (res v′)
    by (auto dest: least-map-leD)
  also from bspec[OF RES-MIN NOT-IN-WL] PATH
  have path-weight′ (res v′) ≤ Num (path-weight p) by blast
  finally show path-weight′ (res′ v′) ≤ Num (path-weight p) .
qed
```

Finally, we have to show that for nodes on the worklist, the stored paths are not longer than any path using only nodes not on the worklist. Compared to the situation before the step, those path may also use the node v.

```
show ∀ va∈wl − {v}. ∀ p.
  is-path v0 p va ∧ int-vertices p ⊆ V − (wl − {v})
  ⟶ path-weight′ (res′ va) ≤ Num (path-weight p)
proof (intro allI impI ballI, elim conjE)
  fix v′ p
  assume IWS: v′∈wl − {v}
    and PATH: is-path v0 p v′
    and VERTICES: int-vertices p ⊆ V − (wl − {v})
  from IWS WL-SUBSET have [simp, intro!]: v′∈V by auto

  {
```

If the path is empty, the proposition follows easily from the invariant for the original states, as no intermediate nodes are used at all.

```
    assume [simp]: p=[]
    from bspec[OF CONSIDERED-NEW-PATHS, of v′] have
      path-weight′ (res′ v′) ≤ path-weight′ (res v′)
      using IWS WL-SUBSET by (auto dest: least-map-leD)
    also have int-vertices p ⊆ V−wl by auto
    with WL-MIN IWS PATH
    have path-weight′ (res v′) ≤ Num (path-weight p)
      by (auto simp del: path-weight-empty)
    finally have path-weight′ (res′ v′) ≤ Num (path-weight p) .
  } moreover {
    fix p1 u w
    assume [simp]: p = p1@[(u,w,v′)]
```

If the path is not empty, we pick the last but one vertex, and call it u.

from *PATH* **have** *PATH1*: *is-path v0 p1 u* **and** *EDGE*: $(u,w,v')\in E$ **by** *auto*
from *VERTICES* **have** *NIV*: $u\in V - (wl-\{v\})$ **by** *simp*
hence *U-MEM*[*simp*]: $u\in V$ **by** *auto*

From *RES-MIN*, we know that *res u* holds the shortest path to u. Thus p is longer than the path that is constructed by replacing the prefix of p by term "res u"

from *NIV RES-MIN PATH1*
have *G*: *Num* (*path-weight p1*) $\geq$ *path-weight'* (*res u*) **by** *simp*
then obtain *pu* **where** [*simp*]: *res u* = *Some pu*
  by (*cases res u*) (*auto simp*: *infty-unbox*)
from *G* **have** *Num* (*path-weight p*) $\geq$ *path-weight'* (*res u*) + *Num w*
  by (*auto simp*: *infty-unbox add-right-mono*)
also
have *path-weight'* (*res u*) + *Num w* $\geq$ *path-weight'* (*res' v'*)

The remaining argument depends on wether u equals v. In the case $u \neq v$, all vertices of *res u* are outside the original workset. Thus, appending the edge (u, w, v') to *res u* yields a path to v over intermediate nodes only outside the workset. By the invariant for the original state, *res v'* is shorter than this path. As a step does not replace paths by longer ones, also *res' v'* is shorter.

In the case $u = v$, the step has considered the path to v' over v, and thus the result path is not longer.

proof (*cases* $u=v$)
  assume $u\neq v$
  with *NIV* **have** *NIV'*: $u\in V-wl$ **by** *auto*
  from *bspec*[*OF PATH-VALID U-MEM*] *NIV'*
  have *is-path v0 pu u* **and** *VU*: *int-vertices* (pu@$[(u,w,v')]$) $\subseteq V-wl$
    by *auto*
  with *EDGE* **have** *PV'*: *is-path v0* (pu@$[(u,w,v')]$) v' **by** *auto*
  with *bspec*[*OF WL-MIN*, *of v'*] *IWS VU* **have**
    path-weight' (*res v'*) $\leq$ *Num* (*path-weight* (pu@$[(u,w,v')]$))
    by *blast*
  hence *path-weight'* (*res u*) + *Num w* $\geq$ *path-weight'* (*res v'*)
    by (*auto simp*: *infty-unbox*)
  also from *CONSIDERED-NEW-PATHS* **have**
    path-weight' (*res v'*) $\geq$ *path-weight'* (*res' v'*)
    by (*auto dest*: *least-map-leD*)
  finally (*order-trans*[*rotated*]) **show** *?thesis* **.**
next
  assume [*symmetric*,*simp*]: $u=v$
  from *CONSIDERED-NEW-PATHS EDGE* **have**
    path-weight' (*res' v'*) $\leq$ *path-weight'* (*Some* (pu@$[(v,w,v')]$))
    by (*rule-tac least-map-leD*) *auto*
  thus *?thesis* **by** (*auto simp*: *infty-unbox*)
qed
finally (*order-trans*[*rotated*]) **have**
  path-weight' (*res' v'*) $\leq$ *Num* (*path-weight p*) **.**
} **ultimately show** *path-weight'* (*res' v'*) $\leq$ *Num* (*path-weight p*)

using *PATH* **apply** (*cases p rule*: *rev-cases*) **by** *auto*
qed
qed
qed

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*
:: $'V \Rightarrow 'V\ set \Rightarrow$ - $\Rightarrow ('W \times 'V)\ set \Rightarrow ('V,'W)\ state \Rightarrow bool$ **where**
$uinvar\ v\ wl\ res\ it\ \sigma \equiv let\ (wl',res')=\sigma\ in\ wl'=wl$
$\wedge\ (\forall v' \in V.$
$res'\ v' \in least\text{-}map\ path\text{-}weight'\ ($
$\{\ res\ v'\ \} \cup \{\ Some\ (p@[(v,w,v')])\ |\ p\ w.\ res\ v = Some\ p$
$\wedge\ (w,v') \in succ\ G\ v - it\ \}$
$))$
$\wedge\ (\forall v' \in V.\ \forall p.\ res'\ v' = Some\ p \longrightarrow is\text{-}path\ v0\ p\ v')$
$\wedge\ res'\ v = res\ v$

definition *update'* :: $'V \Rightarrow ('V,'W)\ state \Rightarrow ('V,'W)\ state\ nres$ **where**
$update'\ v\ \sigma \equiv do\ \{$
$ASSERT\ (update\text{-}pre\ v\ \sigma);$
$let\ (wl,res) = \sigma;$
$let\ wv = path\text{-}weight'\ (res\ v);$
$let\ pv = res\ v;$
$FOREACH^{uinvar\ v\ wl\ res}\ (succ\ G\ v)\ (\lambda(w',v')\ (wl,res).$
$if\ (wv + Num\ w' < path\text{-}weight'\ (res\ v'))\ then\ do\ \{$
$ASSERT\ (v' \in wl \wedge pv \neq None);$
$RETURN\ (wl,res(v' \mapsto the\ pv@[(v,w',v')]))$
$\}\ else\ RETURN\ (wl,res)$
$)\ (wl,res)\}$

lemma *update'-refines*:
assumes $v'=v$ **and** $\sigma'=\sigma$
shows $update'\ v'\ \sigma' \le \Downarrow Id\ (update\ v\ \sigma)$
apply (*simp only*: *assms*)
unfolding *update'-def update-def*
apply (*refine-rcg refine-vcg*)

apply (*simp-all only*: *singleton-iff*)
proof −
fix *wl res*
assume *update-pre v* (*wl*,*res*)

```
  thus uinvar v wl res (succ G v) (wl,res)
    by (simp add: uinvar-def update-pre-def)
next

  fix wl res it wl' res' v' w'
  assume PRE: update-pre v (wl,res)
  assume INV: uinvar v wl res it (wl',res')
  assume MEM: (w',v')∈it
  assume IT-SS: it⊆ succ G v
  assume LESS: path-weight' (res v) + Num w' < path-weight' (res' v')

  from PRE have [simp, intro!]: v∈V by (simp add: update-pre-def)

  from MEM IT-SS have [simp,intro!]: v'∈V using succ-subset
    by auto

  from LESS obtain pv where [simp]: res v = Some pv
    by (cases res v) auto

  thus res v ≠ None by simp

  have [simp]: wl'=wl and [simp]: res' v = res v
    using INV unfolding uinvar-def by auto

  from MEM IT-SS have EDGE[simp]: (v,w',v')∈E
    unfolding succ-def by auto
  with INV have [simp]: is-path v0 pv v
    unfolding uinvar-def by auto

  have 0≤w' by (rule nonneg-weights[OF EDGE])
  hence [simp]: v'≠v using LESS
    by auto
  hence [simp]: v≠v' by blast

  show [simp]: v'∈wl' proof (rule ccontr)
    assume [simp]: v'∉wl'
    hence [simp]: v'∈V−wl and [simp]: v'∉wl by auto
    note LESS
    also
    from INV have path-weight' (res' v') ≤ path-weight' (res v')
      unfolding uinvar-def by (auto dest: least-map-leD)
    also
    from PRE have PW: ⋀p. is-path v0 p v' ⟹
      path-weight' (res v') ≤ path-weight' (Some p)
      unfolding update-pre-def
      by auto
    have P: is-path v0 (pv@[(v,w',v')]) v' by simp
    from PW[OF P] have
      path-weight' (res v') ≤ Num (path-weight (pv@[(v,w',v')]))
```

by *auto*
finally show *False* **by** (*simp add*: *infty-unbox*)
qed

show *uinvar v wl res* (*it*−{(*w′*,*v′*)}) (*wl′*,*res′*(*v′*↦*the* (*res v*)@[(*v*,*w′*,*v′*)]))
proof −
have (*res′*(*v′*↦*the* (*res v*)@[(*v*,*w′*,*v′*)])) *v* = *res′ v* **by** *simp*
moreover {
fix *v′′* **assume** *VMEM*: *v′′*∈*V*
have (*res′*(*v′*↦*the* (*res v*)@[(*v*,*w′*,*v′*)])) *v′′* ∈ *least-map path-weight′* (
{ *res v′′* } ∪ { *Some* (*p*@[(*v*,*w*,*v′′*)]) | *p w*. *res v* = *Some p*
∧ (*w*,*v′′*) ∈ *succ G v* − (*it* − {(*w′*,*v′*)}) }
) ∧ (∀ *p*. (*res′*(*v′*↦*the* (*res v*)@[(*v*,*w′*,*v′*)])) *v′′* = *Some p*
⟶ *is-path v0 p v′′*)
proof (*cases v′′*=*v′*)
case [*simp*]: *False*
have { *Some* (*p*@[(*v*,*w*,*v′′*)]) | *p w*. *res v* = *Some p*
∧ (*w*,*v′′*) ∈ *succ G v* − (*it* − {(*w′*,*v′*)}) } =
{ *Some* (*p*@[(*v*,*w*,*v′′*)]) | *p w*. *res v* = *Some p*
∧ (*w*,*v′′*) ∈ *succ G v* − *it* }
by *auto*
with *INV VMEM* **show** *?thesis* **unfolding** *uinvar-def*
by *simp*
next
case [*simp*]: *True*
have *EQ*: { *res v′′* } ∪ { *Some* (*p*@[(*v*,*w*,*v′′*)]) | *p w*. *res v* = *Some p*
∧ (*w*,*v′′*) ∈ *succ G v* − (*it* − {(*w′*,*v′*)}) } =
insert (*Some* (*pv*@[(*v*,*w′*,*v′*)])) (
{ *res v′′* } ∪ { *Some* (*p*@[(*v*,*w*,*v′′*)]) | *p w*. *res v* = *Some p*
∧ (*w*,*v′′*) ∈ *succ G v* − *it* })
using *MEM IT-SS*
by *auto*
show *?thesis*
apply (*subst EQ*)
apply *simp*
apply (*rule least-map-insert-min*)
apply (*rule ballI*)
proof −
fix *r′*
assume *A*:
r′ ∈ *insert* (*res v′*)
{*Some* (*pv* @ [(*v*, *w*, *v′*)]) |*w*. (*w*, *v′*) ∈ *succ G v* ∧ (*w*, *v′*) ∉ *it*}

from *LESS* **have**
path-weight′ (*Some* (*pv* @ [(*v*, *w′*, *v′*)])) < *path-weight′* (*res′ v′*)
by (*auto simp*: *infty-unbox*)
also from *INV*[*unfolded uinvar-def*] **have**
res′ v′ ∈ *least-map path-weight′* (
insert (*res v′*)

```
          {Some (pv @ [(v, w, v′)]) |w. (w, v′) ∈ succ G v ∧ (w, v′) ∉ it}
          )
          by auto
        with A have path-weight′ (res′ v′) ≤ path-weight′ r′
          by (auto dest: least-map-leD)
        finally show
          path-weight′ (Some (pv @ [(v, w′, v′)])) ≤ path-weight′ r′
          by simp
      qed
    qed
  }
  ultimately show ?thesis
    unfolding uinvar-def Let-def
    by auto
qed
next
  fix wl res it w′ v′ wl′ res′
  assume INV: uinvar v wl res it (wl′,res′)
  and NLESS: ¬ path-weight′ (res v) + Num w′ < path-weight′ (res′ v′)
  and IN-IT: (w′,v′)∈it
  and IT-SS: it ⊆ succ G v

  from IN-IT IT-SS have [simp, intro!]: (w′,v′)∈succ G v by auto
  hence [simp,intro!]: v′∈V using succ-subset
    by auto

  show uinvar v wl res (it − {(w′,v′)}) (wl′,res′)
  proof (cases res v)
    case [simp]: None
    from INV show ?thesis
      unfolding uinvar-def by auto
  next
    case [simp]: (Some p)
    {
      fix v′′
      assume [simp, intro!]: v′′∈V
      have 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 − {(w′,v′)}) }
        ) (is - ∈ least-map path-weight′ ?S)
      proof (cases v′′=v′)
        case False with INV show ?thesis
          unfolding uinvar-def by auto
      next
        case [simp]: True

        have EQ: ?S = insert (Some (p@[(v,w′,v′)])) (
          { res v′ } ∪ { Some (p@[(v,w,v′′)]) | p w. res v = Some p
                        ∧ (w,v′′) ∈ succ G v − it }
```

```
        )
        by auto
      from NLESS have
        path-weight' (res' v') ≤ path-weight' (Some (p@[(v,w',v')]))
        by (auto simp: infty-unbox)
      thus ?thesis
        apply (subst EQ)
        apply (rule least-map-insert-nmin)
        using INV unfolding uinvar-def apply auto []
        apply simp
        done
    qed
  } with INV
  show ?thesis
    unfolding uinvar-def by auto
 qed
next
 fix wl res σ'

 assume uinvar v wl res {} σ'
 thus update-spec v (wl,res) σ'
   unfolding uinvar-def
   apply (cases σ')
   apply (auto intro: update-spec.intros simp: succ-def)
   done
next
 show finite (succ G v) by simp
qed
```

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 −
  note [refine] = update'-refines
  have [refine]: ⋀σ σ'. σ=σ' ⟹ pop-min σ ≤ ⇓Id (pop-min σ') by simp
  show ?thesis
    unfolding dijkstra-def dijkstra'-def
    apply (refine-rcg)
    apply simp-all
    done
```

qed
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* $= ('V \rightharpoonup 'W\ infty)$
type-synonym $('V,'W)$ *mres* $= ('V \rightharpoonup (('V,'W)\ path \times 'W))$
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'* $:: (('V,'W)\ path \times 'W)\ option \rightharpoonup ('V,'W)\ path$ **where**
  $mpath'\ None = None \mid$
  $mpath'\ (Some\ (p,w)) = Some\ p$

fun *mpath-weight'* $:: (('V,'W)\ path \times 'W)\ option \Rightarrow ('W{::}weight)\ infty$ **where**
  $mpath\text{-}weight'\ None = top \mid$
  $mpath\text{-}weight'\ (Some\ (p,w)) = Num\ w$

context *Dijkstra*
begin
  definition $\alpha w{::}('V,'W)\ mwl \Rightarrow 'V\ set$ **where** $\alpha w \equiv dom$
  definition $\alpha r{::}('V,'W)\ mres \Rightarrow 'V \rightharpoonup ('V,'W)\ path$ **where**
    $\alpha r \equiv \lambda res\ v.\ case\ res\ v\ of\ None \Rightarrow None \mid Some\ (p,w) \Rightarrow Some\ (rev\ p)$
  definition $\alpha s{::}\ ('V,'W)\ mstate \Rightarrow ('V,'W)\ state$ **where**
    $\alpha s \equiv map\text{-}prod\ \alpha w\ \alpha r$

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

  definition *res-invarm* $:: ('V \rightharpoonup (('V,'W)\ path \times 'W)) \Rightarrow bool$ **where**
    $res\text{-}invarm\ res \equiv (\forall\ v.\ case\ res\ v\ of$
        $None \Rightarrow True \mid$
        $Some\ (p,w) \Rightarrow w = path\text{-}weight\ (rev\ p))$
  definition *dinvarm* $:: ('V,'W)\ mstate \Rightarrow bool$ **where**
    $dinvarm\ \sigma \equiv let\ (wl,res) = \sigma\ in$
      $(\forall\ v{\in}dom\ wl.\ the\ (wl\ v) = mpath\text{-}weight'\ (res\ v)) \wedge res\text{-}invarm\ res$

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

    unfolding *dinvarm-def res-invarm-def* αr*-def*

by (*auto split*: *option.split option.split-asm*)

lemma *mpath′-correct*: ⟦*dinvarm* (*wl*,*res*)⟧ ⟹
mpath′ (*res v*) = *map-option rev* (*αr res v*)
unfolding *dinvarm-def αr-def*
by (*auto split*: *option.split option.split-asm*)

lemma *wl-weight-correct*:
assumes *INV*: *dinvarm* (*wl*,*res*)
assumes *WLV*: *wl v* = *Some w*
shows *path-weight′* (*αr res v*) = *w*
proof −
from *INV WLV* **have** *w* = *mpath-weight′* (*res v*)
unfolding *dinvarm-def* **by** *force*
also from *mpath-weight′-correct*[*OF INV*] **have**
. . . = *path-weight′* (*αr res v*) **.**
finally show *?thesis* **by** *simp*
qed

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*
unfolding *mdinit-def dinit-def*
apply (*rule build-rel-SPEC*)
apply (*intro FOREACH-rule*[**where** *I*=*λit wl*. (∀ *v*∈*V*−*it*. *wl v* = *Some Infty*)
∧
dom wl = *V*−*it*]
refine-vcg)
apply (*auto*
simp: *αs-def αw-def αr-def dinvarm-def res-invarm-def infty-unbox*
split: *if-split-asm*
)
done

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*:
  ⟦ $(\sigma,\sigma') \in$ *build-rel* αs *dinvarm* ⟧ $\Longrightarrow$
    mpop-min $\sigma \leq$
     $\Downarrow$(*build-rel*
        $(\lambda(v,w,\sigma).\ (v,\alpha s\ \sigma))$
        $(\lambda(v,w,\sigma).$ *dinvarm* $\sigma \land w =$ *mpath-weight'* (*snd* σ v)))
    (*pop-min* σ')
  — The two algorithms are structurally different, so we use the nofail/inres method to prove refinement.
  unfolding *mpop-min-def pop-min-def prio-pop-min-def*

  apply (*rule pw-ref-I*)
  apply *rule*
  apply (*auto simp add*: *refine-pw-simps* αs*-def* αw*-def refine-rel-defs*
    split: *prod.split prod.split-asm*)

  apply (*auto simp*: *dinvarm-def*) []

  apply (*auto simp*: *mpath-weight'-correct wl-weight-correct*) []

  apply (*auto*
    simp: *wl-weight-correct*
    intro!: *least-map.intros*
) []
  apply (*metis restrict-map-eq*(*2*))

  done

The new update function:

definition *uinvarm v wl res it* $\sigma \equiv$
  uinvar v wl res it $(\alpha s\ \sigma) \land$ *dinvarm* σ

definition *mupdate* :: $'V \Rightarrow 'W$ *infty* $\Rightarrow ('V,'W)$ *mstate* $\Rightarrow ('V,'W)$ *mstate nres*
 where
  mupdate v wv $\sigma \equiv$ *do* {
    ASSERT (*update-pre v* $(\alpha s\ \sigma) \land$ *wv*=*mpath-weight'* (*snd* σ v));
    let (*wl*,*res*) = σ;
    let pv = *mpath'* (*res v*);
    FOREACH$^{uinvarm\ v\ (\alpha w\ wl)\ (\alpha r\ res)}$ (*succ G v*) $(\lambda(w',v')$ (*wl*,*res*).
      if (*wv* + *Num* $w' <$ *mpath-weight'* (*res* v')) *then do* {
        ASSERT ($v' \in$*dom wl* $\land$ *pv* $\neq$ *None*);
        ASSERT (*wv* $\neq$ *Infty*);
        RETURN (*wl*($v' \mapsto$*wv* + *Num* w'),
                res($v' \mapsto ((v,w',v')\#$*the pv*,*val wv* + w')))
      } *else RETURN* (*wl*,*res*)
) (*wl*,*res*)
  }

lemma *mupdate-refines*:

assumes *SREF*: $(\sigma,\sigma')\in$*build-rel* αs *dinvarm*
assumes *WV*: *wv* = *mpath-weight'* (*snd* σ *v*)
assumes *VV'*: *v'*=*v*
shows *mupdate v wv* $\sigma \leq \Downarrow$(*build-rel* αs *dinvarm*) (*update' v'* σ')
proof (*simp only*: *VV'*)
{

Show that IF-condition is a refinement:

fix *wl res wl' res' it w' v'*
assume *uinvarm v* (αw *wl*) (αr *res*) *it* (*wl'*,*res'*)
and *dinvarm* (*wl*,*res*)
hence *mpath-weight'* (*res v*) + *Num w'* < *mpath-weight'* (*res' v'*) $\longleftrightarrow$
path-weight' (αr *res v*) + *Num w'* < *path-weight'* (αr *res' v'*)
unfolding *uinvarm-def*
by (*auto simp add*: *mpath-weight'-correct*)
} **note** *COND-refine*=*this*

{

THEN-case:

fix *wl res wl' res' it w' v'*
assume *UINV*: *uinvarm v* (αw *wl*) (αr *res*) *it* (*wl'*,*res'*)
and *DINV*: *dinvarm* (*wl*,*res*)
and *mpath-weight'* (*res v*) + *Num w'* < *mpath-weight'* (*res' v'*)
and *path-weight'* (αr *res v*) + *Num w'* < *path-weight'* (αr *res' v'*)
and *V'MEM*: *v'*$\in\alpha w$ *wl'*
and *NN*: αr *res v* $\neq$ *None*

from *NN* **obtain** *pv wv* **where**
ARV: αr *res v* = *Some* (*rev pv*) **and**
RV: *res v* = *Some* (*pv*,*wv*)
unfolding αr*-def* **by** (*auto split*: *option.split-asm*)

with *DINV* **have** [*simp*]: *wv* = *path-weight* (*rev pv*)
unfolding *dinvarm-def res-invarm-def* **by** (*auto split*: *option.split-asm*)

note [*simp*] = *ARV RV*

from *V'MEM NN* **have** *v'*$\in$*dom wl'* (**is** *?G1*)
and *mpath'* (*res v*) $\neq$ *None* (**is** *?G2*)
unfolding αw*-def* αr*-def* **by** (*auto split*: *option.split-asm*)

hence $\bigwedge x$. αw *wl'* = αw (*wl'*(*v'*$\mapsto$*x*)) **by** (*auto simp*: αw*-def*)
moreover have *mpath'* (*res v*) = *map-option rev* (αr *res v*) **using** *DINV*
by (*simp add*: *mpath'-correct*)
ultimately have
αw *wl'* = αw (*wl'*(*v'* $\mapsto$ *mpath-weight'* (*res v*) + *Num w'*))
$\wedge$ (αr *res'*)(*v'* $\mapsto$ *the* (αr *res v*)@[(*v*, *w'*, *v'*)])
= αr (*res'*(*v'* $\mapsto$ ((*v*, *w'*, *v'*)#*the* (*mpath'* (*res v*)),

val (*mpath-weight′* (*res v*)) + *w′*))) (**is** *?G3*)
by (*auto simp add*: *αr-def intro*!: *ext*)
have
(*dinvarm* (*wl′*(*v′*↦*mpath-weight′* (*res v*) + *Num w′*),
res′(*v′* ↦ ((*v*,*w′*,*v′*) # *the* (*mpath′* (*res v*)),
val (*mpath-weight′* (*res v*)) + *w′*
)))) (**is** *?G4*)
using *UINV* **unfolding** *uinvarm-def dinvarm-def res-invarm-def*
by (*auto simp*: *infty-unbox split*: *option.split option.split-asm*)
note ‹*?G1*› ‹*?G2*› ‹*?G3*› ‹*?G4*›
} **note** *THEN-refine=this*

note [*refine2*] = *inj-on-id*

note [*simp*] = *refine-rel-defs*

show *mupdate v wv σ* ≤ ⇓(*build-rel αs dinvarm*) (*update′ v σ′*)
using *SREF WV*
unfolding *mupdate-def update′-def*
apply −

apply (*refine-rcg*)

apply *simp-all* [*3*]
apply (*simp add*: *αs-def uinvarm-def*)
apply (*simp-all add*: *αs-def COND-refine THEN-refine*(*1*−*2*)) [*3*]
apply (*rule ccontr*,*simp*)
using *THEN-refine*(*3*,*4*)
apply (*auto simp*: *αs-def*) []

The ELSE-case is trivial:

apply *simp*
done
qed

Finally, we assemble the refined algorithm:

definition *mdijkstra* **where**
mdijkstra ≡ *do* {
σ0 ← *mdinit*;
(-,*res*) ← $WHILE_T^{dinvarm}$ (λ(*wl*,-). *dom wl*≠{})
(λσ. *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 −
note [*refine*] = *mdinit-refines mpop-min-refines mupdate-refines*

show *?thesis*
unfolding *mdijkstra-def dijkstra'-def*
apply (*refine-rcg*)
apply (*simp-all split*: *prod.split*
add: *αs-def αw-def dinvarm-def refine-rel-defs*)
done
qed

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

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

type-synonym ($'V$,$'W$,$'G$) *graph-empty* = *unit* $\Rightarrow$ $'G$
locale *graph-empty* = *graph* +
constrains α :: $'G \Rightarrow$ ($'V$,$'W$) *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 α :: $'G \Rightarrow$ ($'V$,$'W$) *graph*
fixes *add-node* :: $'V \Rightarrow 'G \Rightarrow 'G$
assumes *add-node-correct*:
invar g $\Longrightarrow$ *invar* (*add-node v g*)
invar g $\Longrightarrow$ α (*add-node v g*) = *Graph.add-node v* (α *g*)

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

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

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

type-synonym $('V,'W,'G)$ *graph-delete-edge* = $'V \Rightarrow 'W \Rightarrow 'V \Rightarrow 'G \Rightarrow 'G$
locale *graph-delete-edge* = *graph* +
  constrains α :: $'G \Rightarrow ('V,'W)$ *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 $\Longrightarrow$ α (*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-defs* =
  fixes *nodes-list-it* :: $'G \Rightarrow ('V,'V$ *list*) *set-iterator*
begin
  definition *nodes-it g* $\equiv$ *it-to-it* (*nodes-list-it g*)
end

locale *graph-nodes-it* = *graph* α *invar* + *graph-nodes-it-defs nodes-list-it*
  for α :: $'G \Rightarrow ('V,'W)$ *graph* **and** *invar* **and**
  nodes-list-it :: $'G \Rightarrow ('V,'V$ *list*) *set-iterator*
  +
  assumes *nodes-list-it-correct*:
    invar g $\Longrightarrow$ *set-iterator* (*nodes-list-it g*) (*Graph.nodes* (α *g*))
begin
  lemma *nodes-it-correct*:
    invar g $\Longrightarrow$ *set-iterator* (*nodes-it g*) (*Graph.nodes* (α *g*))
    unfolding *nodes-it-def*
    apply (*rule it-to-it-correct*)
    by (*rule nodes-list-it-correct*)

  lemma *pi-nodes-it*[*icf-proper-iteratorI*]:
    proper-it (*nodes-it S*) (*nodes-it S*)
    unfolding *nodes-it-def*
    by (*intro icf-proper-iteratorI*)

lemma *nodes-it-proper*[*proper-it*]:
proper-it′ nodes-it nodes-it
apply (*rule proper-it′I*)
by (*rule pi-nodes-it*)

end

type-synonym ($'V$,$'W$,$'\sigma$,$'G$) *graph-edges-it*
= $'G \Rightarrow$ (($'V \times 'W \times 'V$),$'\sigma$) *set-iterator*

locale *graph-edges-it-defs* =
fixes *edges-list-it* :: ($'V$,$'W$,($'V \times 'W \times 'V$) *list*,$'G$) *graph-edges-it*
begin
definition *edges-it g* ≡ *it-to-it* (*edges-list-it g*)
end

locale *graph-edges-it* = *graph* α *invar* + *graph-edges-it-defs edges-list-it*
for α :: $'G \Rightarrow$ ($'V$,$'W$) *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* (α *g*))
begin
lemma *edges-it-correct*:
invar g $\Longrightarrow$ *set-iterator* (*edges-it g*) (*Graph.edges* (α *g*))
unfolding *edges-it-def*
apply (*rule it-to-it-correct*)
by (*rule edges-list-it-correct*)

lemma *pi-edges-it*[*icf-proper-iteratorI*]:
proper-it (*edges-it S*) (*edges-it S*)
unfolding *edges-it-def*
by (*intro icf-proper-iteratorI*)

lemma *edges-it-proper*[*proper-it*]:
proper-it′ edges-it edges-it
apply (*rule proper-it′I*)
by (*rule pi-edges-it*)

end

type-synonym ($'V$,$'W$,$'\sigma$,$'G$) *graph-succ-it* =
$'G \Rightarrow 'V \Rightarrow$ ($'W \times 'V$,$'\sigma$) *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* ≡ *it-to-it* (*succ-list-it g v*)
end

locale *graph-succ-it* = *graph* α *invar* + *graph-succ-it-defs* *succ-list-it*
for α :: $'G \Rightarrow ('V,'W)$ *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* (α *g*) *v*)
begin
lemma *succ-it-correct*:
invar g $\Longrightarrow$ *set-iterator* (*succ-it g v*) (*Graph.succ* (α *g*) *v*)
unfolding *succ-it-def*
apply (*rule it-to-it-correct*)
by (*rule succ-list-it-correct*)

lemma *pi-succ-it*[*icf-proper-iteratorI*]:
proper-it (*succ-it S v*) (*succ-it S v*)
unfolding *succ-it-def*
by (*intro icf-proper-iteratorI*)

lemma *succ-it-proper*[*proper-it*]:
proper-it' ($\lambda S.$ *succ-it S v*) ($\lambda S.$ *succ-it S v*)
apply (*rule proper-it'I*)
by (*rule pi-succ-it*)

end

6.1 Adjacency Lists

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

definition *adjl-*α :: $('V,'W)$ *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*)
apply (*unfold-locales*)
unfolding *adjl-*α*-def*
by *force*+

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

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*
gop-nodes-list-it $:: 'G \Rightarrow ('V,'V$ *list*$)$ *set-iterator*
gop-edges-list-it $:: ('V,'W,('V \times 'W \times 'V)$ *list*$,'G)$ *graph-edges-it*
gop-succ-list-it $:: 'G \Rightarrow 'V \Rightarrow ('W \times 'V,('W \times 'V)$ *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 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*
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 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* +
graph-nodes-it α *invar nodes-list-it* +
graph-edges-it α *invar edges-list-it* +
graph-succ-it α *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 $\Longrightarrow$ *set-iterator* (*nodes-it g*) (*nodes* (α *g*))
by (*rule nodes-it-correct*)

lemma (**in** *graph-edges-it*) *edges-it-is-iterator*[*refine-transfer*]:
invar g $\Longrightarrow$ *set-iterator* (*edges-it g*) (*edges* (α *g*))
by (*rule edges-it-correct*)

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*)
by (*rule succ-it-correct*)

lemma (**in** *graph*) *drh*[*refine-dref-RELATES*]: *RELATES* (*build-rel* α *invar*)
by (*simp add*: *RELATES-def*)

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* (λ*g v. a v g*) (*e* ()) *nl*

```
    in foldl (λg (v,e,v'). u v e v' g) g1 el

lemma gga-from-list-correct:
  fixes α :: 'G ⇒ ('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 −
  interpret
    graph-empty α invar e +
    graph-add-node α invar a +
    graph-add-edge α invar u
    by fact+

  {
    fix nl el
    define g1 where g1 = foldl (λg v. a v g) (e ()) nl
    define g2 where g2 = foldl (λg (v,e,v'). u v e v' g) g1 el
    have invar g1 ∧ α g1 = (| nodes = set nl, edges = {} |)
      unfolding g1-def
      by (induct nl rule: rev-induct)
        (auto simp: empty-correct add-node-correct empty-def add-node-def)
    hence invar g2
      ∧ α g2 = (| nodes = set nl ∪ fst`set el ∪ snd`snd`set el,
                 edges = set el |)
      unfolding g2-def
      by (induct el rule: rev-induct) (auto simp: add-edge-correct add-edge-def)
    hence invar g2 ∧ adjl-α (nl,el) = α g2
      unfolding adjl-α-def by auto
  }
  thus ?thesis
    unfolding gga-from-list-def [abs-def]
    apply unfold-locales
    apply auto
    done
qed

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×'V) list,'G) graph-succ-it
begin
  definition gga-edges-list-it ::
    ('V,'W,('V×'W×'V) list,'G) graph-edges-it
```

```
    where gga-edges-list-it G ≡ set-iterator-product
      (nodes-it G) (succ-it G)
  local-setup ‹Locale-Code.lc-decl-del @{term gga-edges-list-it}›
end
setup ‹
  (Record-Intf.add-unf-thms-global @{thms
    gga-edges-it-defs.gga-edges-list-it-def[abs-def]
  })
›

locale gga-edges-it = gga-edges-it-defs nodes-list-it succ-list-it
  + graph α invar
  + graph-nodes-it α invar nodes-list-it
  + graph-succ-it α invar succ-list-it
  for α :: 'G ⇒ ('V,'W) graph
  and invar
  and nodes-list-it :: ('V,'W,'V list,'G) graph-nodes-it
  and succ-list-it :: ('V,'W,('W×'V) list,'G) graph-succ-it
begin
  lemma gga-edges-list-it-impl:
    shows graph-edges-it α invar gga-edges-list-it
  proof
    fix g
    assume INV: invar g

    from set-iterator-product-correct[OF
      nodes-it-correct[OF INV] succ-it-correct[OF INV]]
    have set-iterator (set-iterator-product (nodes-it g) (λv. succ-it g v))
      (SIGMA v:nodes (α g). succ (α g) v)
      .
    also have (SIGMA v:nodes (α g). succ (α g) v) = edges (α g)
      unfolding succ-def
      by (auto dest: valid-graph.E-validD[OF valid[OF INV]])

    finally show set-iterator (gga-edges-list-it g) (edges (α g))
      unfolding gga-edges-list-it-def .
  qed
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 list,'G) graph-nodes-it
  and edges-list-it :: ('V,'W,('V×'W×'V) list,'G) graph-edges-it
begin
  definition gga-to-list ::
    ('V,'W,'G) graph-to-list
    where
    gga-to-list g ≡
```

```
      (nodes-it g (λ-. True) (#) [], edges-it g (λ-. True) (#) [])

  end

  locale gga-to-list-loc = gga-to-list-defs-loc nodes-list-it edges-list-it +
    graph α invar
    + graph-nodes-it α invar nodes-list-it
    + graph-edges-it α invar edges-list-it
    for α :: 'G ⇒ ('V,'W) graph and invar
    and nodes-list-it :: ('V,'W,'V list,'G) graph-nodes-it
    and edges-list-it :: ('V,'W,('V×'W×'V) list,'G) graph-edges-it
  begin

    lemma gga-to-list-correct:
      shows graph-to-list α invar gga-to-list
    proof
      fix g
      assume [simp, intro!]: invar g
      then interpret valid-graph α g by (rule valid)

      have set (nodes-it g (λ-. True) (#) []) = V
        apply (rule-tac I=λit σ. set σ = V − it
          in set-iterator-rule-P[OF nodes-it-correct])
        by auto
      moreover have set (edges-it g (λ-. True) (#) []) = E
        apply (rule-tac I=λit σ. set σ = E − it
          in set-iterator-rule-P[OF edges-it-correct])
        by auto
      ultimately show adjl-α (gga-to-list g) = α g
        unfolding adjl-α-def gga-to-list-def
        apply simp
        apply (rule graph.equality)
        apply (auto intro: E-validD)
        done
    qed

  end

end
```

8 Implementing Graphs by Maps

```
theory GraphByMap
imports
  GraphSpec
  GraphGA
begin

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)))*
  unfolding *map-Sigma-def*
  by *auto*

lemma *ranE*:
  assumes *v∈ran m*
  obtains *k* **where** *m k = Some v*
  using *assms*
by (*metis ran-restrictD restrict-map-self*)
lemma *option-bind-alt*:
  Option.bind x f = (case x of None ⇒ None | Some v ⇒ f v)
  by (*auto split*: *option.split*)

locale *GraphByMapDefs* =
  m1: *StdMapDefs m1-ops* +
  m2: *StdMapDefs m2-ops* +
  s3: *StdSetDefs s3-ops*
  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)*

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

  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*

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*

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*)

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*
)

definition *gbm-nodes-list-it*
:: (′*V*,′*W*,′*V list*,′*m1*) *graph-nodes-it*
where
gbm-nodes-list-it g ≡ *map-iterator-dom* (*m1.iteratei g*)
local-setup ‹*Locale-Code.lc-decl-del* @{*term gbm-nodes-list-it*}›

definition *gbm-edges-list-it*
:: (′*V*,′*W*,(′*V*×′*W*×′*V*) *list*,′*m1*) *graph-edges-it*
where
gbm-edges-list-it g ≡ *set-iterator-image*
(λ((*v1*,*m1*),(*v2*,*m2*),*w*). (*v1*,*w*,*v2*))
(*set-iterator-product* (*m1.iteratei g*)
(λ(*v*,*m2*). *set-iterator-product*
(*m2.iteratei m2*) (λ(*w*,*s3*). *s3.iteratei s3*)))

local-setup ‹*Locale-Code.lc-decl-del* @{*term gbm-edges-list-it*}›

definition *gbm-succ-list-it* ::
(′*V*,′*W*,(′*W*×′*V*) *list*,′*m1*) *graph-succ-it*
where

gbm-succ-list-it 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* (*m2.iteratei m2*) (λ(*v'*,*s*). *s3.iteratei s*))

local-setup ‹*Locale-Code.lc-decl-del* @{*term gbm-succ-list-it*}›

definition
  gbm-from-list ≡ *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*)
  ≡ *map-iterator-dom* (*it-to-it* (*m1.list-it g*))
  apply (*rule eq-reflection*)
  apply (*rule it-to-it-fold*)
  unfolding *gbm-nodes-list-it-def m1.iteratei-def*
  by (*intro icf-proper-iteratorI*)

lemma *gbm-edges-list-it-unf*:
  it-to-it (*gbm-edges-list-it g*)
  ≡ *set-iterator-image*
    (λ((*v1*,*m1*),(*v2*,*m2*),*w*). (*v1*,*w*,*v2*))
    (*set-iterator-product* (*it-to-it* (*m1.list-it g*))
      (λ(*v*,*m2*). *set-iterator-product*
        (*it-to-it* (*m2.list-it m2*)) (λ(*w*,*s3*). (*it-to-it* (*s3.list-it s3*)))))

  apply (*rule eq-reflection*)
  apply (*rule it-to-it-fold*)
  unfolding *gbm-edges-list-it-def*
    m1.iteratei-def m2.iteratei-def s3.iteratei-def
  apply (*intro icf-proper-iteratorI allI impI*, (*simp split*: *prod.split*)*?*)+
  done

lemma *gbm-succ-list-it-unf*:
it-to-it (*gbm-succ-list-it 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* (*it-to-it* (*m2.list-it m2*))
          (λ(*v'*,*s*). (*it-to-it* (*s3.list-it s*))))

  apply (*rule eq-reflection*)
  apply (*rule it-to-it-fold*)
  unfolding *gbm-succ-list-it-def*
    m2.iteratei-def s3.iteratei-def
  apply (*simp split*: *prod.split option.split*)

```
    apply (intro icf-proper-iteratorI allI impI conjI,
      (simp split: prod.split option.split)?)+
    done

end

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

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
    ⦈
  local-setup ‹Locale-Code.lc-decl-del @{term gbm-ops}›
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)
    using assms unfolding gbm-invar-def
    by (auto intro: ranI)

end

sublocale GraphByMap < graph gbm-α gbm-invar
proof
  fix g
  assume INV: gbm-invar g
  then interpret vg: valid-graph (gbm-α g) by (simp add: gbm-invar-def)

  from vg.E-valid
  show fst ` edges (gbm-α g) ⊆ nodes (gbm-α g) and
    snd ` snd ` edges (gbm-α g) ⊆ nodes (gbm-α g) .

  from INV show finite (nodes (gbm-α g))
    unfolding gbm-invar-def gbm-α-def by auto

  note [simp] = gbm-invar-split[OF INV]

  show finite (edges (gbm-α g))
    apply (rule finite-imageD[where f=λ(v,e,v'). (v,v',e)])
    apply (rule finite-subset[where B=
      map-Sigma (m1.α g) (λm2. map-Sigma (m2.α m2) (s3.α))])
    apply (auto simp add: map-Sigma-def gbm-α-def) []
    apply (unfold map-Sigma-alt)
    apply (auto intro!: finite-SigmaI inj-onI)
    done
qed

context GraphByMap
begin

  lemma gbm-empty-impl:
    graph-empty gbm-α gbm-invar gbm-empty
    apply (unfold-locales)
    unfolding gbm-α-def gbm-invar-def gbm-empty-def
    apply (auto simp: m1.correct Graph.empty-def)
    apply (unfold-locales)
    apply auto
    done

  lemma gbm-add-node-impl:
    graph-add-node gbm-α gbm-invar gbm-add-node
  proof
    fix g v
    assume INV: gbm-invar g
```

note [*simp*]= *gbm-invar-split*[*OF INV*]
show *gbm-α* (*gbm-add-node v g*) = *add-node v* (*gbm-α g*)
unfolding *gbm-α-def gbm-add-node-def*
by (*auto simp*: *m1.correct m2.correct s3.correct add-node-def*
split: *option.split if-split-asm*)

thus *gbm-invar* (*gbm-add-node v g*)
unfolding *gbm-invar-def*
apply (*simp*)
unfolding *gbm-α-def gbm-add-node-def add-node-def*
apply (*auto simp*: *m1.correct m2.correct s3.correct add-node-def*
split: *option.split if-split-asm elim*!: *ranE*)
done
qed

lemma *gbm-delete-node-impl*:
graph-delete-node gbm-α gbm-invar gbm-delete-node
proof
fix *g v*
assume *INV*: *gbm-invar g*
note [*simp*]= *gbm-invar-split*[*OF INV*]
show *gbm-α* (*gbm-delete-node v g*) = *delete-node v* (*gbm-α g*)
unfolding *gbm-α-def gbm-delete-node-def*
by (*auto simp*: *restrict-map-def option-bind-alt*
m1.correct m2.correct s3.correct m1.map-value-image-filter-correct
delete-node-def
split: *option.split if-split-asm option.split-asm*)

thus *gbm-invar* (*gbm-delete-node v g*)
unfolding *gbm-invar-def*
apply (*simp*)
unfolding *gbm-α-def gbm-delete-node-def delete-node-def*
apply (*auto simp*: *restrict-map-def option-bind-alt*
m1.correct m2.correct s3.correct m1.map-value-image-filter-correct
split: *option.split if-split-asm option.split-asm elim*!: *ranE*)
done
qed

lemma *gbm-add-edge-impl*:
graph-add-edge gbm-α gbm-invar gbm-add-edge
proof
fix *g v e v'*
assume *INV*: *gbm-invar g*
note [*simp*]= *gbm-invar-split*[*OF INV*]
show *gbm-α* (*gbm-add-edge v e v' g*) = *add-edge v e v'* (*gbm-α g*)
unfolding *gbm-α-def gbm-add-edge-def*
apply (*auto simp*: *m1.correct m2.correct s3.correct*
Let-def
split: *option.split if-split-asm*)

unfolding *add-edge-def*

apply (*fastforce split*: *if-split-asm*
simp: *m1.correct m2.correct s3.correct*
)+
done

thus *gbm-invar* (*gbm-add-edge v e v′ g*)
unfolding *gbm-invar-def*
apply (*simp*)
unfolding *gbm-α-def gbm-add-edge-def*
apply (*force simp*: *m1.correct m2.correct s3.correct*
Let-def
split: *option.split if-split-asm elim*!: *ranE*)
done
qed

lemma *gbm-delete-edge-impl*:
graph-delete-edge gbm-α gbm-invar gbm-delete-edge
proof
fix *g v e v′*
assume *INV*: *gbm-invar g*
note [*simp*]= *gbm-invar-split*[*OF INV*]
show *gbm-α* (*gbm-delete-edge v e v′ g*) = *delete-edge v e v′* (*gbm-α g*)
unfolding *gbm-α-def gbm-delete-edge-def delete-edge-def*
apply (*auto simp*: *m1.correct m2.correct s3.correct*
Let-def
split: *option.split if-split-asm*)
done

thus *gbm-invar* (*gbm-delete-edge v e v′ g*)
unfolding *gbm-invar-def*
apply (*simp*)
unfolding *gbm-α-def gbm-delete-edge-def*
apply (*auto simp*: *m1.correct m2.correct s3.correct*
Let-def
split: *option.split if-split-asm elim*!: *ranE*)
done
qed

lemma *gbm-nodes-list-it-impl*:
shows *graph-nodes-it gbm-α gbm-invar gbm-nodes-list-it*
proof
fix *g*
assume *gbm-invar g*
hence *MINV*: *map-op-invar m1-ops g* **unfolding** *gbm-invar-def* **by** *auto*
from *map-iterator-dom-correct*[*OF m1.iteratei-correct*[*OF MINV*]]
show *set-iterator* (*gbm-nodes-list-it g*) (*nodes* (*gbm-α g*))
unfolding *gbm-nodes-list-it-def gbm-α-def* **by** *simp*

```
qed

lemma gbm-edges-list-it-impl:
  shows graph-edges-it gbm-α gbm-invar gbm-edges-list-it
proof
  fix g
  assume INV: gbm-invar g

  from INV have I1: m1.invar g unfolding gbm-invar-def by auto
  from INV have I2: ⋀v m2. (v,m2)∈map-to-set (m1.α g) ⟹ m2.invar m2
    unfolding gbm-invar-def map-to-set-def
    by (auto simp: ran-def)
  from INV have I3: ⋀v m2 v' s. ⟦
    (v,m2)∈map-to-set (m1.α g);
    (v',s)∈map-to-set (m2.α m2)⟧
    ⟹ s3.invar s
    unfolding gbm-invar-def map-to-set-def
    by (auto simp: ran-def)

  show set-iterator (gbm-edges-list-it g) (edges (gbm-α g))
    unfolding gbm-edges-list-it-def
    apply (rule set-iterator-image-correct)
    apply (rule set-iterator-product-correct)
    apply (rule m1.iteratei-correct)
    apply (rule I1)
    apply (case-tac a)
    apply clarsimp
    apply (rule set-iterator-product-correct)
    apply (drule I2)
    apply (subgoal-tac map-iterator (m2.iteratei ba)
      (map-op-α m2-ops (snd (aa,ba))))
    apply assumption
    apply (simp add: m2.iteratei-correct)

    apply (case-tac a)
    apply clarsimp
    apply (subgoal-tac set-iterator (s3.iteratei bb)
      (s3.α (snd (ab,bb))))
    apply assumption
    apply (simp add: s3.iteratei-correct I3)

    apply (auto simp: inj-on-def map-to-set-def) []

    apply (force simp: gbm-α-def map-to-set-def) []
    done
  qed

lemma gbm-succ-list-it-impl:
  shows graph-succ-it gbm-α gbm-invar gbm-succ-list-it
```

proof
  fix *g v*
  assume *INV*: *gbm-invar g*
  hence *I1*[*simp*]: *m1.invar g* **unfolding** *gbm-invar-def* **by** *auto*

  show *set-iterator* (*gbm-succ-list-it g v*) (*succ* (*gbm-α g*) *v*)
  proof (*cases m1.lookup v g*)
    case *None* **hence** (*succ* (*gbm-α g*) *v*) = {}
      unfolding *succ-def gbm-α-def* **by** (*auto simp*: *m1.lookup-correct*)
    with *None* **show** *?thesis* **unfolding** *gbm-succ-list-it-def*
      by (*auto simp*: *set-iterator-emp-correct*)
  next
    case (*Some m2*)
    hence [*simp*]: *m2.invar m2* **using** *gbm-invar-split*[*OF INV*]
      by (*simp add*: *m1.lookup-correct*)

    from *INV Some* **have**
      I2: $\bigwedge v'\ s.\ (v',\ s) \in$ *map-to-set* (*map-op-α m2-ops m2*) $\Longrightarrow$ *s3.invar s*
      unfolding *gbm-invar-def*
      by (*auto simp*: *map-to-set-def ran-def m1.lookup-correct*)

    from *Some* **show** *?thesis*
      unfolding *gbm-succ-list-it-def* **apply** *simp*
      apply (*rule set-iterator-image-correct*)
      apply (*rule set-iterator-product-correct*)
      apply (*rule m2.iteratei-correct*)
      apply *simp*
      apply (*case-tac a*, *simp*)
      apply (*subgoal-tac set-iterator* (*s3.iteratei b*) (*s3.α* (*snd* (*aa*, *b*))))
      apply *assumption*
      apply *simp*
      apply (*rule s3.iteratei-correct*)
      apply (*simp add*: *I2*)

      apply (*auto simp*: *inj-on-def map-to-set-def*) []

      apply (*force simp*: *succ-def gbm-α-def map-to-set-def m1.lookup-correct*)
      done
  qed
qed

lemma *gbm-from-list-impl*:
  shows *graph-from-list gbm-α gbm-invar gbm-from-list*
  unfolding *gbm-from-list-def*
  apply (*rule gga-from-list-correct*)
  apply (*rule gbm-empty-impl gbm-add-node-impl gbm-add-edge-impl*)+
  done

end

```
sublocale GraphByMap < graph-nodes-it gbm-α gbm-invar gbm-nodes-list-it
  by (rule gbm-nodes-list-it-impl)
sublocale GraphByMap < graph-edges-it gbm-α gbm-invar gbm-edges-list-it
  by (rule gbm-edges-list-it-impl)
sublocale GraphByMap < graph-succ-it gbm-α gbm-invar gbm-succ-list-it
  by (rule gbm-succ-list-it-impl)

sublocale GraphByMap
  < gga-to-list-loc gbm-α gbm-invar gbm-nodes-list-it gbm-edges-list-it
  by unfold-locales

context GraphByMap
begin
  lemma gbm-to-list-impl: graph-to-list gbm-α gbm-invar gga-to-list
    by (rule gga-to-list-correct)

  lemma gbm-ops-impl: StdGraph gbm-ops
    apply (rule StdGraph.intro)
    apply (simp-all add: icf-rec-unf)
    apply icf-locales
    apply (rule gbm-empty-impl gbm-add-node-impl gbm-delete-node-impl
      gbm-add-edge-impl gbm-delete-edge-impl gbm-from-list-impl
      gbm-to-list-impl)+
    done
end

setup ‹
  (Record-Intf.add-unf-thms-global @{thms
    GraphByMapDefs.gbm-nodes-list-it-unf
    GraphByMapDefs.gbm-edges-list-it-unf
    GraphByMapDefs.gbm-succ-list-it-unf
  })
›

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
```

setup *Locale-Code.open-block*
interpretation *hh-mvif*: *g-value-image-filter-loc hm-ops hm-ops*
  by *unfold-locales*
interpretation *hlg-gbm*: *GraphByMap hm-ops hm-ops ls-ops*
  hh-mvif.g-value-image-filter
  by *unfold-locales*
setup *Locale-Code.close-block*

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

setup *Locale-Code.open-block*
interpretation *hlg*: *StdGraph hlg-ops*
  unfolding *hlg-ops-def*
  by (*rule hlg-gbm.gbm-ops-impl*)
setup *Locale-Code.close-block*
setup ‹*ICF-Tools.revert-abbrevs HashGraphImpl.hlg*›

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

definition *test-codegen* **where** *test-codegen* ≡ (
  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 wv 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 (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 g v0 ≡ *do* {
$\sigma 0$ ← *cdinit g v0*;
(-,*res*) ← $WHILE_T$ (λ(*wl*,-). ¬ *qw.isEmpty wl*)
($\lambda\sigma$. *do* { (*v*,*wv*,σ') ← *cpop-min* σ; *cupdate g v wv* σ' })
$\sigma 0$;
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* ≤⇓?*R mdinit*
using *g-rel*
unfolding *cdinit-def mdinit-def*
apply (*refine-rcg*)
apply (*refine-dref-type*)
apply (*simp-all add*: α*sc-def dinvarC-add-def refine-rel-defs*
qw.correct mr.correct refine-hsimp)
done

schematic-goal *cpop-min-refines*:
(σ,σ') ∈ *build-rel* α*sc dinvarC-add*
⟹ *cpop-min* σ ≤ ⇓?*R* (*mpop-min* σ')
unfolding *cpop-min-def mpop-min-def*
apply (*refine-rcg*)
apply (*refine-dref-type*)
apply (*simp add*: α*sc-def dinvarC-add-def refine-hsimp refine-rel-defs*)
apply (*simp add*: α*sc-def dinvarC-add-def refine-hsimp refine-rel-defs*)
done

schematic-goal *cupdate-refines*:
notes [*refine*] = *inj-on-id*
shows (σ,σ')∈*build-rel* α*sc dinvarC-add* ⟹ *v*=*v'* ⟹ *wv*=*wv'* ⟹
cupdate g v wv σ ≤ ⇓?*R* (*mupdate v' wv'* σ')
unfolding *cupdate-def mupdate-def*
using *g-rel*

```
  apply (refine-rcg)
  apply (refine-dref-type)
  apply (simp-all add: αsc-def dinvarC-add-def refine-rel-defs
    qw.correct mr.correct refine-hsimp)
  done

lemma cdijkstra-refines:
  cdijkstra g v0 ≤ ⇓(build-rel mr.α mr.invar) mdijkstra
proof −
  note [refine] = cdinit-refines cpop-min-refines cupdate-refines
  show ?thesis
    unfolding cdijkstra-def mdijkstra-def
    using g-rel
    apply (refine-rcg)

    apply (auto
      split: prod.split prod.split-asm
      simp add: qw.correct mr.correct dinvarC-add-def αsc-def refine-hsimp
        refine-rel-defs)
    done
qed
end

context dijkstraC
begin

thm g.nodes-it-is-iterator

schematic-goal idijkstra-refines-aux:
  assumes g.invar g
  shows RETURN ?f ≤ cdijkstra g v0
  using assms
  unfolding cdijkstra-def cdinit-def cpop-min-def cupdate-def
  apply (refine-transfer)
  done

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

lemma idijkstra-refines:
  assumes g.invar g
  shows RETURN (idijkstra g v0) ≤ cdijkstra g v0
  using assms
  by (rule idijkstra.refine)

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 (αr (*mr*.α (*idijkstra g v0*)))
    (**is** *?G1*)
  and *mr.invar* (*idijkstra g v0*) (**is** *?G2*)
  and *Dijkstra.res-invarm* (*mr*.α (*idijkstra g v0*)) (**is** *?G3*)
proof –
  from *g-rel* **have** *I*: *g.invar g* **by** (*simp add*: *refine-rel-defs*)

  note *idijkstra-refines*[*OF I*]
  also note *cdijkstra-refines*
  also note *mdijkstra-refines*
  finally have *Z*: *RETURN* (*idijkstra g v0*) $\leq$
    $\Downarrow$(*build-rel* ($\alpha r \circ$ *mr*.α) ($\lambda m.$ *mr.invar m* $\wedge$ *res-invarm* (*mr*.α *m*)))
      dijkstra'
    apply (*subst* (*asm*) *conc-fun-chain*)
    apply (*simp only*: *br-chain*)
    done
  also note *dijkstra'-refines*[*simplified*]
  also note *dijkstra-correct*
  finally show *?G1 ?G2 ?G3*
    by (*auto elim*: *RETURN-ref-SPECD simp*: *refine-rel-defs*)
qed

theorem (**in** *dijkstraC*) *idijkstra-correct*:
  assumes *INV*: *g.invar g*
  assumes *V0*: *v0* $\in$ *nodes* (*g*.α *g*)
  assumes *nonneg-weights*: $\bigwedge v\ w\ v'.\ (v,w,v') \in$ *edges* (*g*.α *g*) $\Longrightarrow 0 \leq w$
  shows
  weighted-graph.is-shortest-path-map (*g*.α *g*) *v0*
      (*Dijkstra*.αr (*mr*.α (*idijkstra g v0*))) (**is** *?G1*)
  and *Dijkstra.res-invarm* (*mr*.α (*idijkstra g v0*)) (**is** *?G2*)
proof –
  interpret *gv*: *valid-graph g*.α *g* **using** *g.valid INV* **.**

  interpret *dcg*: *dijkstraC-fixg g-ops mr-ops qw-ops g*.α *g v0 g*
    apply (*rule dijkstraC-fixg.intro*)
    apply *unfold-locales*
    apply (*simp-all add*: *hlg.finite INV V0 hlg-ops-def*
      nonneg-weights refine-rel-defs)
    done

from *dcg.idijkstra-correct* **show** *?G1 ?G2* **by** *simp-all*
qed

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

setup *Locale-Code.open-block*
interpretation *hrf*: *dijkstraC hlg-ops rm-ops aluprioi-ops*
by *unfold-locales*
setup *Locale-Code.close-block*

definition *hrf-dijkstra* ≡ *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* ⇒ -
where *hrfn-dijkstra* ≡ *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*
≡ *rm.to-list*
(*hrfn-dijkstra* (*hlg.from-list* ([*0*..<*4*],[(*0*,*3*,*1*),(*0*,*4*,*2*),(*2*,*1*,*3*),(*1*,*4*,*3*)])) *0*)

ML-val ‹
@{*code test-hrfn-dijkstra*}

›

end

11 Implementation of Dijkstra's-Algorithm using Automatic Determinization

theory *Dijkstra-Impl-Adet*
imports

Dijkstra
GraphSpec
HashGraphImpl
Collections.Refine-Dflt-ICF
HOL−Library.Code-Target-Numeral
begin

11.1 Setup

11.1.1 Infinity

definition *infty-rel-internal-def*:
$infty\text{-}rel\ R \equiv \{(Num\ a,\ Num\ a')|\ a\ a'.\ (a,a')\in R\} \cup \{(Infty,Infty)\}$
lemma *infty-rel-def*[*refine-rel-defs*]:
$\langle R\rangle infty\text{-}rel = \{(Num\ a,\ Num\ a')|\ a\ a'.\ (a,a')\in R\} \cup \{(Infty,Infty)\}$
unfolding *infty-rel-internal-def relAPP-def* **by** *simp*

lemma *infty-relI*:
$(Infty,Infty)\in\langle R\rangle infty\text{-}rel$
$(a,a')\in R \Longrightarrow (Num\ a,\ Num\ a')\in\langle R\rangle infty\text{-}rel$
unfolding *infty-rel-def* **by** *auto*

lemma *infty-relE*:
assumes $(x,x')\in\langle R\rangle infty\text{-}rel$
obtains $x=Infty$ **and** $x'=Infty$
| $a\ a'$ **where** $x=Num\ a$ **and** $x'=Num\ a'$ **and** $(a,a')\in R$
using *assms*
unfolding *infty-rel-def*
by *auto*

lemma *infty-rel-simps*[*simp*]:
$(Infty,x')\in\langle R\rangle infty\text{-}rel \longleftrightarrow x'=Infty$
$(x,Infty)\in\langle R\rangle infty\text{-}rel \longleftrightarrow x=Infty$
$(Num\ a,\ Num\ a')\in\langle R\rangle infty\text{-}rel \longleftrightarrow (a,a')\in R$
unfolding *infty-rel-def* **by** *auto*

lemma *infty-rel-sv*[*relator-props*]:
$single\text{-}valued\ R \Longrightarrow single\text{-}valued\ (\langle R\rangle infty\text{-}rel)$
unfolding *infty-rel-def*
by (*auto intro*: *single-valuedI dest*: *single-valuedD*)

lemma *infty-rel-id*[*simp*, *relator-props*]: $\langle Id\rangle infty\text{-}rel = Id$
apply *rule*
apply (*auto elim*: *infty-relE*) []
apply *safe*
apply (*case-tac b*) **by** *auto*

consts *i-infty* :: *interface* ⇒ *interface*
lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of infty-rel i-infty*]

lemma *autoref-infty*[*param*,*autoref-rules*]:
  $(Infty,Infty) \in \langle R \rangle infty\text{-}rel$
  $(Num,Num) \in R \rightarrow \langle R \rangle infty\text{-}rel$
  $(case\text{-}infty,case\text{-}infty) \in Rr \rightarrow (R \rightarrow Rr) \rightarrow \langle R \rangle infty\text{-}rel \rightarrow Rr$
  $(rec\text{-}infty,rec\text{-}infty) \in Rr \rightarrow (R \rightarrow Rr) \rightarrow \langle R \rangle infty\text{-}rel \rightarrow Rr$
  unfolding *infty-rel-def*
  by (*auto dest*: *fun-relD*)

definition [*simp*]: *is-Infty x* ≡ *case x of Infty* ⇒ *True* | - ⇒ *False*

context begin interpretation *autoref-syn* **.**
lemma *pat-is-Infty*[*autoref-op-pat*]:
  $x{=}Infty \equiv (OP\ is\text{-}Infty :::_i \langle I \rangle_i i\text{-}infty \rightarrow_i i\text{-}bool)\$x$
  $Infty{=}x \equiv (OP\ is\text{-}Infty :::_i \langle I \rangle_i i\text{-}infty \rightarrow_i i\text{-}bool)\$x$
  by (*auto intro*!: *eq-reflection split*: *infty.splits*)
end

lemma *autoref-is-Infty*[*autoref-rules*]:
  $(is\text{-}Infty,\ is\text{-}Infty) \in \langle R \rangle infty\text{-}rel \rightarrow bool\text{-}rel$
  by (*auto split*: *infty.splits*)

definition *infty-eq eq v1 v2* ≡
  case (*v1*,*v2*) *of*
    (*Infty*,*Infty*) ⇒ *True*
  | (*Num a1*, *Num a2*) ⇒ *eq a1 a2*
  | - ⇒ *False*

lemma *infty-eq-autoref*[*autoref-rules* (**overloaded**)]:
  $[\![\ GEN\text{-}OP\ eq\ (=)\ (R \rightarrow R \rightarrow bool\text{-}rel)\ ]\!]$
  $\Longrightarrow (infty\text{-}eq\ eq,(=)) \in \langle R \rangle infty\text{-}rel \rightarrow \langle R \rangle infty\text{-}rel \rightarrow bool\text{-}rel$
  unfolding *infty-eq-def*[*abs-def*]
  by (*auto split*: *infty.splits dest*: *fun-relD elim*!: *infty-relE*)

lemma *infty-eq-expand*[*autoref-struct-expand*]: (=) = *infty-eq* (=)
  by (*auto intro*!: *ext simp*: *infty-eq-def split*: *infty.splits*)

context begin interpretation *autoref-syn* **.**
lemma *infty-val-autoref*[*autoref-rules*]:
  $[\![SIDE\text{-}PRECOND\ (x \neq Infty);\ (xi,x) \in \langle R \rangle infty\text{-}rel]\!]$
  $\Longrightarrow (val\ xi,(OP\ val ::: \langle R \rangle infty\text{-}rel \rightarrow R)\ \$\ x) \in R$
  apply (*cases x*)
  apply (*auto elim*: *infty-relE*)
  done
end

definition *infty-plus* **where**
  infty-plus pl a b ≡ *case* (*a*,*b*) *of* (*Num a*, *Num b*) ⇒ *Num* (*pl a b*) | - ⇒ *Infty*

lemma *infty-plus-param*[*param*]:

(*infty-plus*,*infty-plus*) ∈ (*R*→*R*→*R*) → ⟨*R*⟩*infty-rel* → ⟨*R*⟩*infty-rel* → ⟨*R*⟩*infty-rel*
unfolding *infty-plus-def*[*abs-def*]
by *parametricity*

lemma *infty-plus-eq-plus*: *infty-plus* (+) = (+)
unfolding *infty-plus-def*[*abs-def*]
by (*auto intro*!: *ext split*: *infty.split*)

lemma *infty-plus-autoref*[*autoref-rules*]:
GEN-OP pl (+) (*R*→*R*→*R*)
⟹ (*infty-plus pl*,(+)) ∈ ⟨*R*⟩*infty-rel* → ⟨*R*⟩*infty-rel* → ⟨*R*⟩*infty-rel*
apply (*fold infty-plus-eq-plus*)
apply *simp*
apply *parametricity*
done

11.1.2 Graph

consts *i-graph* :: *interface* ⇒ *interface* ⇒ *interface*

definition *graph-more-rel-internal-def*:
graph-more-rel Rm Rv Rw ≡ { (*g*,*g'*).
(*graph.nodes g*, *graph.nodes g'*)∈⟨*Rv*⟩*set-rel*
∧ (*graph.edges g*, *graph.edges g'*)∈⟨⟨*Rv*,⟨*Rw*,*Rv*⟩*prod-rel*⟩*prod-rel*⟩*set-rel*
∧ (*graph.more g*, *graph.more g'*)∈*Rm*}

lemma *graph-more-rel-def*[*refine-rel-defs*]:
⟨*Rm*,*Rv*,*Rw*⟩*graph-more-rel* ≡ { (*g*,*g'*).
(*graph.nodes g*, *graph.nodes g'*)∈⟨*Rv*⟩*set-rel*
∧ (*graph.edges g*, *graph.edges g'*)∈⟨⟨*Rv*,⟨*Rw*,*Rv*⟩*prod-rel*⟩*prod-rel*⟩*set-rel*
∧ (*graph.more g*, *graph.more g'*)∈*Rm*}
unfolding *relAPP-def graph-more-rel-internal-def* **by** *simp*

abbreviation *graph-rel* ≡ ⟨*unit-rel*⟩*graph-more-rel*
lemmas *graph-rel-def* = *graph-more-rel-def*[**where** *Rm*=*unit-rel*, *simplified*]

lemma *graph-rel-id*[*simp*]: ⟨*Id*,*Id*⟩*graph-rel* = *Id*
unfolding *graph-rel-def* **by** *auto*

lemma *graph-more-rel-sv*[*relator-props*]:
⟦*single-valued Rm*; *single-valued Rv*; *single-valued Rw*⟧
⟹ *single-valued* (⟨*Rm*,*Rv*,*Rw*⟩*graph-more-rel*)
unfolding *graph-more-rel-def*
apply (*rule single-valuedI*, *clarsimp*)
apply (*rule graph.equality*)
apply (*erule* (*1*) *single-valuedD*[*rotated*], *tagged-solver*)+
done

lemma [*autoref-itype*]:
graph.nodes $::_i$ $\langle Iv,Iw\rangle_i$*i-graph* $\rightarrow_i$ $\langle Iv\rangle_i$*i-set*
by *simp-all*

thm *is-map-to-sorted-list-def*

definition *nodes-to-list g* ≡ *it-to-sorted-list* (λ- -. *True*) (*graph.nodes g*)
lemma *nodes-to-list-itype*[*autoref-itype*]: *nodes-to-list* $::_i$ $\langle Iv,Iw\rangle_i$*i-graph* $\rightarrow_i$ $\langle\langle Iv\rangle_i$*i-list*$\rangle_i$*i-nres*
by *simp*
lemma *nodes-to-list-pat*[*autoref-op-pat*]: *it-to-sorted-list* (λ- -. *True*) (*graph.nodes g*) ≡ *nodes-to-list g*
unfolding *nodes-to-list-def* **by** *simp*

definition *succ-to-list g v* ≡ *it-to-sorted-list* (λ- -. *True*) (*Graph.succ g v*)
lemma *succ-to-list-itype*[*autoref-itype*]:
succ-to-list $::_i$ $\langle Iv,Iw\rangle_i$*i-graph* $\rightarrow_i$ *Iv* $\rightarrow_i$ $\langle\langle\langle Iw,Iv\rangle_i$*i-prod*$\rangle_i$*i-list*$\rangle_i$*i-nres* **by** *simp*
lemma *succ-to-list-pat*[*autoref-op-pat*]: *it-to-sorted-list* (λ- -. *True*) (*Graph.succ g v*) ≡ *succ-to-list g v*
unfolding *succ-to-list-def* **by** *simp*

context *graph* **begin**
definition *rel-def-internal*: *rel Rv Rw* ≡ *br* α *invar O* ⟨*Rv*,*Rw*⟩*graph-rel*
lemma *rel-def*: ⟨*Rv*,*Rw*⟩*rel* ≡ *br* α *invar O* ⟨*Rv*,*Rw*⟩*graph-rel*
unfolding *relAPP-def rel-def-internal* **by** *simp*

lemma *rel-id*[*simp*]: ⟨*Id*,*Id*⟩*rel* = *br* α *invar* **by** (*simp add*: *rel-def*)

lemma *rel-sv*[*relator-props*]:
⟦*single-valued Rv*; *single-valued Rw*⟧ ⟹ *single-valued* (⟨*Rv*,*Rw*⟩*rel*)
unfolding *rel-def*
by *tagged-solver*

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 (λ*s*. *RETURN* (*it-to-list nodes-it s*),*nodes-to-list*) ∈ ⟨*Rv*,*Rw*⟩*rel* → ⟨⟨*Rv*⟩*list-rel*⟩*nres-rel*
unfolding *nodes-to-list-def*[*abs-def*]
proof (*intro fun-relI nres-relI*)
fix *s s'*
from *ID* **have** [*simp*]: *Rv* = *Id* **by** *simp*

assume (*s*,*s'*)∈⟨*Rv*,*Rw*⟩*rel*
hence *INV*: *invar s* **and** [*simp*]: *nodes s'* = *nodes* (α *s*) **unfolding** *rel-def*
by (*auto simp add*: *br-def graph-rel-def*)

obtain *l* **where**
[*simp*]: *distinct l nodes* (α *s*) = *set l it-to-list nodes-it s* = *l*

unfolding *it-to-list-def*
by (*metis nodes-it-correct*[*OF INV*, *unfolded set-iterator-def set-iterator-genord-def*]

foldli-snoc-id self-append-conv2)

show *RETURN* (*it-to-list nodes-it s*)
$\leq \Downarrow$ (⟨*Rv*⟩*list-rel*) (*it-to-sorted-list* (λ- -. *True*) (*nodes s'*))
by (*simp add*: *it-to-sorted-list-def*)
qed

lemma (**in** *graph-succ-it*) *autoref-succ-it*[*autoref-rules*]:
assumes *ID*: *PREFER-id Rv PREFER-id Rw*
shows (λ*s v*. *RETURN* (*it-to-list* (λ*s*. *succ-it s v*) *s*),*succ-to-list*)
∈ ⟨*Rv*,*Rw*⟩*rel* → *Rv* → ⟨⟨⟨*Rw*,*Rv*⟩*prod-rel*⟩*list-rel*⟩*nres-rel*
unfolding *succ-to-list-def*[*abs-def*]
proof (*intro fun-relI nres-relI*)
fix *s s' v v'*
from *ID* **have** [*simp*]: *Rv* = *Id Rw*=*Id* **by** *simp-all*

assume (*v*,*v'*)∈*Rv* **hence** [*simp*]: *v'*=*v* **by** *simp*

assume (*s*,*s'*)∈⟨*Rv*,*Rw*⟩*rel*
hence *INV*: *invar s* **and** [*simp*]: *Graph.succ s'* = *Graph.succ* (α *s*) **unfolding** *rel-def*
by (*auto simp add*: *br-def graph-rel-def succ-def*)

obtain *l* **where**
[*simp*]: *distinct l succ* (α *s*) *v* = *set l it-to-list* (λ*s*. *succ-it s v*) *s* = *l*
unfolding *it-to-list-def*
by (*metis succ-it-correct*[*OF INV*, *unfolded set-iterator-def set-iterator-genord-def*]

foldli-snoc-id self-append-conv2)

show *RETURN* (*it-to-list* (λ*s*. *succ-it s v*) *s*)
$\leq \Downarrow$ (⟨⟨*Rw*,*Rv*⟩*prod-rel*⟩*list-rel*) (*it-to-sorted-list* (λ- -. *True*) (*succ s' v'*))
by (*simp add*: *it-to-sorted-list-def*)
qed

11.2 Refinement

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

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*
    by *simp-all*

  lemma [*autoref-rules*]: (*g*,*ga*)∈⟨*v-rel*,*w-rel*⟩*g.rel* **using** *ga-trans*
    by (*simp add*: *g.rel-def*)

  lemma [*autoref-rules*]: (*v0*,*v0*)∈*v-rel* **by** *simp*

  term *mpath-weight'*
  lemma [*autoref-rules*]:
    (*mpath-weight'*,*mpath-weight'*)
      ∈ ⟨⟨*v-rel*$\times_r$*w-rel*$\times_r$*v-rel*⟩*list-rel*$\times_r$*w-rel*⟩*option-rel* → ⟨*w-rel*⟩*infty-rel*
    (*mpath'*, *mpath'*)
      ∈ ⟨⟨*v-rel*$\times_r$*w-rel*$\times_r$*v-rel*⟩*list-rel*$\times_r$*w-rel*⟩*option-rel*
        → ⟨⟨*v-rel*$\times_r$*w-rel*$\times_r$*v-rel*⟩*list-rel*⟩*option-rel*
    by *auto*

  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*$\times_r$*w-rel*$\times_r$*v-rel*⟩*list-rel*$\times_r$*w-rel*⟩*mr.rel*]
    ty-REL[**where** *R*=⟨*v-rel*$\times_r$*w-rel*$\times_r$*v-rel*⟩*list-rel*]

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

schematic-goal *cdijkstra-refines-aux*:
  shows (*?c*::*?'c*,
    mdijkstra
) ∈ *?R*
  apply (*simp only*: *mdijkstra-def mdinit-def mpop-min-def*[*abs-def*] *mupdate-def*)

  using [[*goals-limit* = *1*]]

  apply (*fold op-map-empty-def*[**where** $'a='V$ **and** $'b = ('V \times 'W \times 'V)$ *list* $\times$ $'W$])
  apply (*fold op-uprio-empty-def*[**where** $'a='V$ **and** $'b = 'W$ *infty*])

  using [[*autoref-trace-failed-id*]]

  apply (*autoref-monadic* (*plain*,*trace*))
  done

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*
  apply (*rule cdijkstra.refine*[*THEN nres-relD*, *simplified*])
  apply *unfold-locales*
  done

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 −
  note *cdijkstra-refines*
  also note *mdijkstra-refines*
  finally have *Z*: *RETURN* (*cdijkstra g v0*) ≤
    ⇓(*build-rel* (*αr* ∘ *mr.α*) (*λm. mr.invar m* ∧ *res-invarm* (*mr.α m*)))
      dijkstra′
    apply (*subst* (*asm*) *conc-fun-chain*)
    apply (*simp only*: *br-chain*)
    done
  also note *dijkstra′-refines*[*simplified*]
  also note *dijkstra-correct*
  finally show *?G1 ?G2 ?G3*
    by (*auto elim*: *RETURN-ref-SPECD simp*: *refine-rel-defs*)
qed

end

theorem (**in** *dijkstraC*) *cdijkstra-correct*:
  assumes *INV*: *g.invar g*
  assumes *V0*: *v0* ∈ *nodes* (*g.α g*)
  assumes *nonneg-weights*: ⋀*v w v′*. (*v*,*w*,*v′*)∈*edges* (*g.α g*) ⟹ *0*≤*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.α* (*cdijkstra g v0*)) (**is** *?G2*)
proof −
  interpret *hlgv*: *valid-graph g.α g* **using** *g.valid INV* **.**

  interpret *dc*: *dijkstraC-fixg g-ops mr-ops qw-ops g.α g v0*
    apply *unfold-locales*
    apply (*simp-all*
      add: *hlg.finite INV V0 hlg-ops-def nonneg-weights refine-rel-defs*)
    done

  from *dc.cdijkstra-correct* **show** *?G1 ?G2* **by** *auto*
qed

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

setup *Locale-Code.open-block*
interpretation *hrf*: *dijkstraC hlg-ops rm-ops aluprioi-ops*
  by *unfold-locales*
setup *Locale-Code.close-block*

definition *hrf-dijkstra* ≡ *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* ⇒ -
  where *hrfn-dijkstra* ≡ *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*) *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 ≡ *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-val ‹

```
let
  (* Configuration of test: *)
  val vertices = @{code nat-of-integer} 1000; (* Number of vertices *)
  val seed = @{code nat-of-integer} 123454; (* Seed for random number generator
*)
  val cfg-print-paths = true; (* Whether to output complete paths *)
  val cfg-print-res = true; (* Whether to output result at all *)

  (* Internals *)
  fun string-of-edge (u,(w,v)) = let
    val u = @{code integer-of-nat} u;
    val w = @{code integer-of-nat} w;
    val v = @{code integer-of-nat} v;
  in
    ( ^ string-of-int u ^ , ^ string-of-int w ^ , ^ string-of-int v ^ )
  end

  fun print-entry (dest,(path,weight)) = let
    val dest = @{code integer-of-nat} dest;
    val weight = @{code integer-of-nat} weight;
  in
    writeln (string-of-int dest ^ :  ^ string-of-int weight ^
      ( if cfg-print-paths then
          via [ ^ commas (map string-of-edge (rev path)) ^ ]
        else
      )
    )
  end

  fun print-res [] = ()
    | print-res (a::l) = let val - = print-entry a in print-res l end;

  val start = Time.now();
  val graph = @{code hlg-from-list-nat} (@{code ran-graph} vertices seed);
  val rt1 = Time.toMilliseconds (Time.now() − start);

  val start = Time.now();
  val res = @{code nat-dijkstra} graph (@{code nat-of-integer} 0);
```

```
  val rt2 = Time.toMilliseconds (Time.now() − start);
in
  writeln (string-of-int (@{code integer-of-nat} vertices) ^ vertices:
  ^ string-of-int rt2 ^ ms +
  ^ string-of-int rt1 ^ ms to create graph =
  ^ string-of-int (rt1+rt2) ^ ms);

  if cfg-print-res then
    print-res (@{code nat-res-to-list} res)
  else ()
end;
›
```

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 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://isa-afp.org/entries/Tree-Automata.shtml, Oct. 2010. Formal proof development.