Dijkstra's Algorithm

Benedikt Nordhoff   Peter Lammich

April 19, 2020

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**
    $\llbracket\ x{\in}S;\ \forall\, x'{\in}S.\ f\ x \le f\ x'\ \rrbracket \Longrightarrow x \in \mathit{least\text{-}map}\ f\ S$

  lemma *least-map-subset*: $\mathit{least\text{-}map}\ f\ S \subseteq S$
    ⟨*proof*⟩

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

  lemma *least-map-leD*:
    assumes $x \in \mathit{least\text{-}map}\ f\ S$
    assumes $y{\in}S$
    shows $f\ x \le f\ y$
    ⟨*proof*⟩

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

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

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

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

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

lemma *inf-absorb-less*[*simp*]:
$a < b \Longrightarrow$ *inf* $a\ b = a$
$a < b \Longrightarrow$ *inf* $b\ a = a$
⟨*proof*⟩
end

end

3 Graphs

theory *Graph*
imports *Main*
begin

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

3.1 Definitions

A graph is represented by a record.

record $('v,'w)$ *graph* =
  nodes :: $'v$ *set*
  edges :: $('v \times 'w \times 'v)$ *set*

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

locale *valid-graph* =
  fixes G :: $('v,'w)$ *graph*
  assumes *E-valid*: *fst*‘*edges* $G \subseteq$ *nodes* G
      snd‘*snd*‘*edges* $G \subseteq$ *nodes* G
begin
  abbreviation $V \equiv$ *nodes* G
  abbreviation $E \equiv$ *edges* G

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

end

3.2 Basic operations on Graphs

The empty graph.

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

Adds a node to a graph.

definition *add-node* **where**
  add-node v g $\equiv$ (| *nodes* = *insert* v (*nodes* g), *edges*=*edges* g|)

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

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

Adds an edge to a graph.

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

Deletes an edge from a graph.

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

Successors of a node.

definition *succ* :: $('v,'w)$ *graph* $\Rightarrow$ $'v \Rightarrow ('w\times'v)$ *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*
  ⟨*proof*⟩
lemma *add-node-valid*[*simp*]: **assumes** *valid-graph g*
  shows *valid-graph* (*add-node v g*)
⟨*proof*⟩
lemma *delete-node-valid*[*simp*]: **assumes** *valid-graph g*
  shows *valid-graph* (*delete-node v g*)
⟨*proof*⟩
lemma *add-edge-valid*[*simp*]: **assumes** *valid-graph g*
  shows *valid-graph* (*add-edge v e v′ g*)
⟨*proof*⟩
lemma *delete-edge-valid*[*simp*]: **assumes** *valid-graph g*
  shows *valid-graph* (*delete-edge v e v′ g*)
⟨*proof*⟩

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

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

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

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

3.3 Paths

A path is represented by a list of adjacent edges.

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

context *valid-graph*

begin

The following predicate describes a valid path:

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

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

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

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

lemma *is-path-split'*[*simp*]:
is-path v $(p1@(u,w,u')\#p2)$ v'
$\longleftrightarrow$ *is-path* v $p1$ $u \wedge (u,w,u')\in E \wedge$ *is-path* u' $p2$ v'
⟨*proof*⟩
end

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

definition *int-vertices* :: $('v,'w)$ *path* $\Rightarrow 'v$ *set* **where**
int-vertices $p \equiv$ *set* (*map fst* p)

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

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

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

3.3.1 Splitting Paths

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

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

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

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

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

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

3.4 Weighted Graphs

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

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

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

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

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

end

4 Weights for Dijkstra's Algorithm

theory *Weight*
imports *Complex-Main*
begin

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

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

4.1 Type Classes Setup

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

lemma *add-nonneg-nonneg* [*simp*]:
  assumes $0 \leq a$ **and** $0 \leq b$ **shows** $0 \leq a + b$
⟨*proof*⟩

lemma *add-nonpos-nonpos*[*simp*]:
  assumes $a \leq 0$ **and** $b \leq 0$ **shows** $a + b \leq 0$
⟨*proof*⟩

lemma *add-nonneg-eq-0-iff* :
  assumes *x*: $0 \leq x$ **and** *y*: $0 \leq y$
  shows $x + y = 0 \longleftrightarrow x = 0 \wedge y = 0$
  ⟨*proof*⟩

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

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

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

end

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

term *top*

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

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

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

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

end

4.2 Adding Infinity

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

datatype *'a infty* = *Infty* | *Num 'a*

primrec *val* **where** *val* (*Num d*) = *d*

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

type-synonym *NatB* = *nat infty*

instantiation *infty* :: (*weight*) *top-weight*
begin
  definition (*0::'a infty*) == *Num 0*
  definition *top* ≡ *Infty*

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

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

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

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

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

lemma [*simp*]: *plus Infty a* = *Infty* ⟨*proof*⟩

instance
⟨*proof*⟩
end

4.2.1 Unboxing

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

lemma *infty-inf-unbox*:
Num a ≠ *top*
top ≠ *Num a*
Infty = *top*
⟨*proof*⟩

lemma *infty-ord-unbox*:
Num a ≤ *Num b* ⟷ $a \le b$
Num a < *Num b* ⟷ $a < b$
⟨*proof*⟩

lemma *infty-plus-unbox*:
Num a + *Num b* = *Num* (a+b)
⟨*proof*⟩

lemma *infty-zero-unbox*:
Num a = *0* ⟷ $a = 0$
Num 0 = *0*
⟨*proof*⟩

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

lemma *inf-not-zero*[*simp*]:
top≠(*0*::- *infty*) (*0*::- *infty*)≠*top*
⟨*proof*⟩

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

lemma *infty-neE*:
⟦*a*≠*Infty*; ⋀*d*. *a*=*Num d* ⟹ *P*⟧ ⟹ *P*
⟦*a*≠*top*; ⋀*d*. *a*=*Num d* ⟹ *P*⟧ ⟹ *P*

⟨*proof*⟩

end

5 Dijkstra's Algorithm

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

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

fun *path-weight'* **where**
  path-weight' None = *top* |
  path-weight' (*Some p*) = *Num* (*path-weight p*)

5.3 Dijkstra's Algorithm

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

type-synonym $('V,'W)$ *state* $= ('V\ set) \times ('V \rightharpoonup ('V,'W)\ path)$

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

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

Paths have non-negative weights.

lemma *path-nonneg-weight*: *is-path v p v'* $\Longrightarrow 0 \leq$ *path-weight p*
  ⟨*proof*⟩

Invariant of the main loop:

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

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

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

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

$\langle proof \rangle$

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

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

The initial state satisfies the invariant.

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

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

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

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

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

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

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

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

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

Finally, we define Dijkstra's algorithm:

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

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

theorem *dijkstra-correct*: $dijkstra \leq SPEC\ (is\text{-}shortest\text{-}path\text{-}map\ v0)$
$\langle proof \rangle$

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

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

lemma *update'-refines*:
assumes $v'=v$ **and** $\sigma'=\sigma$

shows $update'\ v'\ \sigma' \le \Downarrow Id\ (update\ v\ \sigma)$
$\langle proof \rangle$

We integrate the new update function into the main algorithm:

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

lemma $dijkstra'\text{-}refines$: $dijkstra' \le \Downarrow Id\ dijkstra$
$\langle proof \rangle$
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**
$mpath'\ None = None\ |$
$mpath'\ (Some\ (p,w)) = Some\ p$

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

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

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

⟨*proof*⟩

lemma *mpath′-correct*: $[\![\mathit{dinvarm}\ (\mathit{wl},\mathit{res})]\!] \Longrightarrow$
$\mathit{mpath}'\ (\mathit{res}\ v) = \mathit{map\text{-}option}\ \mathit{rev}\ (\alpha r\ \mathit{res}\ v)$
⟨*proof*⟩

lemma *wl-weight-correct*:
assumes *INV*: $\mathit{dinvarm}\ (\mathit{wl},\mathit{res})$
assumes *WLV*: $\mathit{wl}\ v = \mathit{Some}\ w$
shows $\mathit{path\text{-}weight}'\ (\alpha r\ \mathit{res}\ v) = w$
⟨*proof*⟩

The initial state is constructed using an iterator:

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

lemma *mdinit-refines*: $\mathit{mdinit} \le \Downarrow(\mathit{build\text{-}rel}\ \alpha s\ \mathit{dinvarm})\ \mathit{dinit}$
⟨*proof*⟩

The new pop function:

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

lemma *mpop-min-refines*:
$[\![\ (\sigma,\sigma') \in \mathit{build\text{-}rel}\ \alpha s\ \mathit{dinvarm}\ ]\!] \Longrightarrow$
$\quad \mathit{mpop\text{-}min}\ \sigma \le$

$\Downarrow$(*build-rel*
  (λ(v,w,σ). (v,αs σ))
  (λ(v,w,σ). *dinvarm* σ $\wedge$ w = *mpath-weight'* (*snd* σ v)))
(*pop-min* σ')
— The two algorithms are structurally different, so we use the nofail/inres method to prove refinement.
⟨*proof*⟩

The new update function:

definition *uinvarm v wl res it* $\sigma \equiv$
  uinvar v wl res it (αs σ) $\wedge$ *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* (αs σ) $\wedge$ *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*) (λ(w',v') (*wl*,*res*).
    if (*wv* + *Num* w' < *mpath-weight'* (*res* v')) *then do* {
      ASSERT ($v'$$\in$*dom wl* $\wedge$ *pv* $\neq$ *None*);
      ASSERT (*wv* $\neq$ *Infty*);
      RETURN (*wl*($v'$$\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*: (σ,σ')$\in$*build-rel* αs *dinvarm*
  assumes *WV*: *wv* = *mpath-weight'* (*snd* σ v)
  assumes *VV'*: v'=v
  shows *mupdate v wv* σ $\leq$ $\Downarrow$(*build-rel* αs *dinvarm*) (*update'* v' σ')
⟨*proof*⟩

Finally, we assemble the refined algorithm:

definition *mdijkstra* **where**
mdijkstra $\equiv$ *do* {
  $\sigma 0$ $\leftarrow$ *mdinit*;
  (-,*res*) $\leftarrow$ $WHILE_T{}^{dinvarm}$ (λ(*wl*,-). *dom wl*$\neq${})
    ($\lambda\sigma$. *do* { (v,*wv*,σ') $\leftarrow$ *mpop-min* σ; *mupdate v wv* σ' })
    $\sigma 0$;
  RETURN res
}

lemma *mdijkstra-refines*: *mdijkstra* $\leq$ $\Downarrow$(*build-rel* αr *res-invarm*) *dijkstra'*
⟨*proof*⟩

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 $\alpha :: 'G \Rightarrow ('V,'W)$ *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 $\alpha :: 'G \Rightarrow ('V,'W)$ *graph*
fixes *empty* :: *unit* $\Rightarrow 'G$
assumes *empty-correct*:
α (*empty* ()) = *Graph.empty*
invar (*empty* ())

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

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

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* ($\alpha\ g$))
begin
  lemma *nodes-it-correct*:
    invar $g \Longrightarrow$ *set-iterator* (*nodes-it* g) (*Graph.nodes* ($\alpha\ g$))
    ⟨*proof*⟩

  lemma *pi-nodes-it*[*icf-proper-iteratorI*]:
    proper-it (*nodes-it* S) (*nodes-it* S)
    ⟨*proof*⟩

  lemma *nodes-it-proper*[*proper-it*]:
    proper-it' *nodes-it* *nodes-it*
    ⟨*proof*⟩

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* $g \equiv$ *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))
    ⟨*proof*⟩

  lemma *pi-edges-it*[*icf-proper-iteratorI*]:
    proper-it (*edges-it* S) (*edges-it* S)
    ⟨*proof*⟩

  lemma *edges-it-proper*[*proper-it*]:
    proper-it' *edges-it* *edges-it*
    ⟨*proof*⟩

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 $v \equiv$ *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)
    ⟨*proof*⟩

  lemma *pi-succ-it*[*icf-proper-iteratorI*]:
    proper-it (*succ-it* S v) (*succ-it* S v)
    ⟨*proof*⟩

  lemma *succ-it-proper*[*proper-it*]:

$proper\text{-}it'$ $(\lambda S.\ succ\text{-}it\ S\ v)$ $(\lambda S.\ succ\text{-}it\ S\ v)$
⟨*proof*⟩

end

6.1 Adjacency Lists

type-synonym $('V,'W)\ adj\text{-}list = 'V\ list \times ('V\times'W\times'V)\ list$

definition $adjl\text{-}\alpha :: ('V,'W)\ adj\text{-}list \Rightarrow ('V,'W)\ graph$ **where**
$adjl\text{-}\alpha\ l \equiv let\ (nl,el) = l\ in\ (\!|$
$nodes = set\ nl \cup fst`set\ el \cup snd`snd`set\ el,$
$edges = set\ el$
$|\!)$

lemma *adjl-is-graph*: $graph\ adjl\text{-}\alpha\ (\lambda\text{-}.\ True)$
⟨*proof*⟩

type-synonym $('V,'W,'G)\ graph\text{-}from\text{-}list = ('V,'W)\ adj\text{-}list \Rightarrow 'G$
locale *graph-from-list* = *graph* +
constrains $\alpha :: 'G \Rightarrow ('V,'W)\ graph$
fixes $from\text{-}list :: ('V,'W)\ adj\text{-}list \Rightarrow 'G$
assumes *from-list-correct*:
$invar\ (from\text{-}list\ l)$
$\alpha\ (from\text{-}list\ l) = adjl\text{-}\alpha\ l$

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

6.2 Record Based Interface

record $('V,'W,'G)\ graph\text{-}ops =$
$gop\text{-}\alpha :: ('V,'W,'G)\ graph\text{-}\alpha$
$gop\text{-}invar :: 'G \Rightarrow bool$
$gop\text{-}empty :: ('V,'W,'G)\ graph\text{-}empty$
$gop\text{-}add\text{-}node :: ('V,'W,'G)\ graph\text{-}add\text{-}node$
$gop\text{-}delete\text{-}node :: ('V,'W,'G)\ graph\text{-}delete\text{-}node$
$gop\text{-}add\text{-}edge :: ('V,'W,'G)\ graph\text{-}add\text{-}edge$
$gop\text{-}delete\text{-}edge :: ('V,'W,'G)\ graph\text{-}delete\text{-}edge$
$gop\text{-}from\text{-}list :: ('V,'W,'G)\ graph\text{-}from\text{-}list$
$gop\text{-}to\text{-}list :: ('V,'W,'G)\ graph\text{-}to\text{-}list$
$gop\text{-}nodes\text{-}list\text{-}it :: 'G \Rightarrow ('V,'V\ list)\ set\text{-}iterator$
$gop\text{-}edges\text{-}list\text{-}it :: ('V,'W,('V\times'W\times'V)\ list,'G)\ graph\text{-}edges\text{-}it$
$gop\text{-}succ\text{-}list\text{-}it :: 'G \Rightarrow 'V \Rightarrow ('W\times'V,('W\times'V)\ list)\ set\text{-}iterator$

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*))
⟨*proof*⟩

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

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

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

end

7 Generic Algorithms for Graphs

theory *GraphGA*
imports
  GraphSpec
begin

definition *gga-from-list* ::
  ('*V*,'*W*,'*G*) *graph-empty* $\Rightarrow$ ('*V*,'*W*,'*G*) *graph-add-node*
    $\Rightarrow$ ('*V*,'*W*,'*G*) *graph-add-edge*
  $\Rightarrow$ ('*V*,'*W*,'*G*) *graph-from-list*
  where
  gga-from-list e a u l $\equiv$
    let (*nl*,*el*) = *l*;
      g1 = *foldl* (λ*g v*. *a v g*) (*e* ()) *nl*
    in foldl (λ*g* (*v*,*e*,*v'*). *u v e v' g*) *g1 el*

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

term *map-iterator-product*

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* $\equiv$ *set-iterator-product*
      (*nodes-it G*) (*succ-it G*)

⟨*ML*⟩
end
⟨*ML*⟩

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

locale *gga-to-list-defs-loc* =
graph-nodes-it-defs nodes-list-it
+ *graph-edges-it-defs edges-list-it*
for *nodes-list-it* :: $('V,'W,'V$ *list*$,'G)$ *graph-nodes-it*
and *edges-list-it* :: $('V,'W,('V\times'W\times'V)$ *list*$,'G)$ *graph-edges-it*
begin
definition *gga-to-list* ::
$('V,'W,'G)$ *graph-to-list*
where
gga-to-list g $\equiv$
(*nodes-it g* (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 :: $'G \Rightarrow ('V,'W)$ *graph* **and** *invar*
and *nodes-list-it* :: $('V,'W,'V$ *list*$,'G)$ *graph-nodes-it*
and *edges-list-it* :: $('V,'W,('V\times'W\times'V)$ *list*$,'G)$ *graph-edges-it*
begin

lemma *gga-to-list-correct*:
shows *graph-to-list* α *invar gga-to-list*
⟨*proof*⟩

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*) (λ*x*.
  F2 (*the* (*M1 x*)))
  ⟨*proof*⟩

lemma *ranE*:
  assumes *v*∈*ran m*
  obtains *k* **where** *m k* = *Some v*
  ⟨*proof*⟩
lemma *option-bind-alt*:
  Option.bind x f = (*case x of None* ⇒ *None* | *Some v* ⇒ *f v*)
  ⟨*proof*⟩

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*)
⟨*ML*⟩

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*)))

⟨*ML*⟩

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*))

⟨*ML*⟩

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*))
⟨*proof*⟩

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*)))))

⟨*proof*⟩

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*))))

⟨*proof*⟩

end

sublocale *GraphByMapDefs* < *graph-nodes-it-defs gbm-nodes-list-it* ⟨*proof*⟩

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

context GraphByMapDefs
begin

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

locale GraphByMap = GraphByMapDefs m1-ops m2-ops s3-ops m1-mvif +
  m1: StdMap m1-ops +
  m2: StdMap m2-ops +
  s3: StdSet s3-ops +
  m1: map-value-image-filter m1.α m1.invar m1.α m1.invar m1-mvif
  for m1-ops::('V,'m2,'m1,-) map-ops-scheme
  and m2-ops::('V,'s3,'m2,-) map-ops-scheme
  and s3-ops::('W,'s3,-) set-ops-scheme
  and m1-mvif :: ('V ⇒ 'm2 ⇀ 'm2) ⇒ 'm1 ⇒ 'm1
begin
  lemma gbm-invar-split:
    assumes gbm-invar g
    shows
    m1.invar g
    ⋀v m2. m1.α g v = Some m2 ⟹ m2.invar m2
    ⋀v m2 v' s3. m1.α g v = Some m2 ⟹ m2.α m2 v' = Some s3 ⟹ s3.invar
s3
    valid-graph (gbm-α g)
    ⟨proof⟩

end

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

context *GraphByMap*
begin

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

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

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

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

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

lemma *gbm-nodes-list-it-impl*:
shows *graph-nodes-it gbm-α gbm-invar gbm-nodes-list-it*
⟨*proof*⟩

lemma *gbm-edges-list-it-impl*:
shows *graph-edges-it gbm-α gbm-invar gbm-edges-list-it*
⟨*proof*⟩

lemma *gbm-succ-list-it-impl*:
shows *graph-succ-it gbm-α gbm-invar gbm-succ-list-it*
⟨*proof*⟩

lemma *gbm-from-list-impl*:
shows *graph-from-list gbm-α gbm-invar gbm-from-list*
⟨*proof*⟩

end

sublocale *GraphByMap* $<$ *graph-nodes-it gbm-α gbm-invar gbm-nodes-list-it*
⟨*proof*⟩
sublocale *GraphByMap* $<$ *graph-edges-it gbm-α gbm-invar gbm-edges-list-it*
⟨*proof*⟩
sublocale *GraphByMap* $<$ *graph-succ-it gbm-α gbm-invar gbm-succ-list-it*
⟨*proof*⟩

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

context *GraphByMap*
begin
lemma *gbm-to-list-impl*: *graph-to-list gbm-α gbm-invar gga-to-list*
⟨*proof*⟩

lemma *gbm-ops-impl*: *StdGraph gbm-ops*
⟨*proof*⟩
end

⟨*ML*⟩

end

9 Graphs by Hashmaps

theory *HashGraphImpl*
imports
GraphByMap
begin

Abbreviation: hlg

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

⟨*ML*⟩
interpretation *hh-mvif*: *g-value-image-filter-loc hm-ops hm-ops*
⟨*proof*⟩
interpretation *hlg-gbm*: *GraphByMap hm-ops hm-ops ls-ops*
hh-mvif.g-value-image-filter
⟨*proof*⟩
⟨*ML*⟩

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

⟨*ML*⟩
interpretation *hlg*: *StdGraph hlg-ops*
⟨*proof*⟩
⟨*ML*⟩

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

definition *test-codegen* **where** *test-codegen* ≡ (
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* $\wedge$ *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 {
    σ0 ← cdinit g v0;
    (-,res) ← WHILE_T (λ(wl,-). ¬ qw.isEmpty wl)
          (λσ. do { (v,wv,σ') ← cpop-min σ; cupdate g v wv σ' } )
          σ0;
    RETURN res
  }

end

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

schematic-goal *cdinit-refines*:
notes [*refine*] = *inj-on-id*
shows *cdinit g v0* $\leq\Downarrow$*?R mdinit*
⟨*proof*⟩

schematic-goal *cpop-min-refines*:
(σ,σ') $\in$ *build-rel* α*sc dinvarC-add*
$\Longrightarrow$ *cpop-min* σ $\leq$ $\Downarrow$*?R* (*mpop-min* σ')
⟨*proof*⟩

schematic-goal *cupdate-refines*:
notes [*refine*] = *inj-on-id*
shows (σ,σ')$\in$*build-rel* α*sc dinvarC-add* $\Longrightarrow$ *v*=*v'* $\Longrightarrow$ *wv*=*wv'* $\Longrightarrow$
cupdate g v wv σ $\leq$ $\Downarrow$*?R* (*mupdate v' wv'* σ')
⟨*proof*⟩

lemma *cdijkstra-refines*:
cdijkstra g v0 $\leq$ $\Downarrow$(*build-rel mr*.α *mr*.*invar*) *mdijkstra*
⟨*proof*⟩
end

context *dijkstraC*
begin

thm *g.nodes-it-is-iterator*

schematic-goal *idijkstra-refines-aux*:
assumes *g.invar g*
shows *RETURN ?f* $\leq$ *cdijkstra g v0*
⟨*proof*⟩

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

lemma *idijkstra-refines*:
assumes *g.invar g*
shows *RETURN* (*idijkstra g v0*) $\leq$ *cdijkstra g v0*
⟨*proof*⟩

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*⟩

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*⟩

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

⟨*ML*⟩
interpretation *hrf*: *dijkstraC hlg-ops rm-ops aluprioi-ops*
 ⟨*proof*⟩
⟨*ML*⟩

definition *hrf-dijkstra* ≡ *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*

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

⟨*ML*⟩

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-rel R $\equiv$ {(*Num a*, *Num a'*)| *a a'*. (*a*,*a'*)∈*R*} ∪ {(*Infty*,*Infty*)}
lemma *infty-rel-def*[*refine-rel-defs*]:
  ⟨*R*⟩*infty-rel* = {(*Num a*, *Num a'*)| *a a'*. (*a*,*a'*)∈*R*} ∪ {(*Infty*,*Infty*)}
  ⟨*proof*⟩

lemma *infty-relI*:
  (*Infty*,*Infty*)∈⟨*R*⟩*infty-rel*
  (*a*,*a'*)∈*R* ⟹ (*Num a*, *Num a'*)∈⟨*R*⟩*infty-rel*
  ⟨*proof*⟩

lemma *infty-relE*:
  assumes (*x*,*x'*)∈⟨*R*⟩*infty-rel*
  obtains *x*=*Infty* **and** *x'*=*Infty*
  | *a a'* **where** *x*=*Num a* **and** *x'*=*Num a'* **and** (*a*,*a'*)∈*R*
  ⟨*proof*⟩

lemma *infty-rel-simps*[*simp*]:
  (*Infty*,*x'*)∈⟨*R*⟩*infty-rel* ⟷ *x'*=*Infty*
  (*x*,*Infty*)∈⟨*R*⟩*infty-rel* ⟷ *x*=*Infty*
  (*Num a*, *Num a'*)∈⟨*R*⟩*infty-rel* ⟷ (*a*,*a'*)∈*R*
  ⟨*proof*⟩

lemma *infty-rel-sv*[*relator-props*]:
  single-valued R ⟹ *single-valued* (⟨*R*⟩*infty-rel*)
  ⟨*proof*⟩

lemma *infty-rel-id*[*simp*, *relator-props*]: ⟨*Id*⟩*infty-rel* = *Id*
⟨*proof*⟩

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

lemma *autoref-infty*[*param*,*autoref-rules*]:
(*Infty*,*Infty*)∈⟨*R*⟩*infty-rel*
(*Num*,*Num*)∈*R*→⟨*R*⟩*infty-rel*
(*case-infty*,*case-infty*)∈*Rr*→(*R*→*Rr*)→⟨*R*⟩*infty-rel*→*Rr*
(*rec-infty*,*rec-infty*)∈*Rr*→(*R*→*Rr*)→⟨*R*⟩*infty-rel*→*Rr*
⟨*proof*⟩

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

context begin interpretation *autoref-syn* ⟨*proof*⟩
lemma *pat-is-Infty*[*autoref-op-pat*]:
x=*Infty* ≡ (*OP is-Infty* $:::_i$ $\langle I\rangle_i$*i-infty* $\rightarrow_i$ *i-bool*)$*x*
Infty=*x* ≡ (*OP is-Infty* $:::_i$ $\langle I\rangle_i$*i-infty* $\rightarrow_i$ *i-bool*)$*x*
⟨*proof*⟩
end

lemma *autoref-is-Infty*[*autoref-rules*]:
(*is-Infty*, *is-Infty*)∈⟨*R*⟩*infty-rel* → *bool-rel*
⟨*proof*⟩

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-OP eq* (=) (*R*→*R*→*bool-rel*) ⟧
⟹ (*infty-eq eq*,(=))∈⟨*R*⟩*infty-rel*→⟨*R*⟩*infty-rel*→*bool-rel*
⟨*proof*⟩

lemma *infty-eq-expand*[*autoref-struct-expand*]: (=) = *infty-eq* (=)
⟨*proof*⟩

context begin interpretation *autoref-syn* ⟨*proof*⟩
lemma *infty-val-autoref*[*autoref-rules*]:
⟦*SIDE-PRECOND* (*x*≠*Infty*); (*xi*,*x*)∈⟨*R*⟩*infty-rel*⟧
⟹ (*val xi*,(*OP val* ::: ⟨*R*⟩*infty-rel* → *R*) $ *x*)∈*R*
⟨*proof*⟩
end

definition *infty-plus* **where**

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

lemma *infty-plus-param*[*param*]:
$(infty\text{-}plus,infty\text{-}plus) \in (R{\to}R{\to}R) \to \langle R\rangle infty\text{-}rel \to \langle R\rangle infty\text{-}rel \to \langle R\rangle infty\text{-}rel$
⟨*proof*⟩

lemma *infty-plus-eq-plus*: $infty\text{-}plus\ (+) = (+)$
⟨*proof*⟩

lemma *infty-plus-autoref*[*autoref-rules*]:
$GEN\text{-}OP\ pl\ (+)\ (R{\to}R{\to}R)$
$\Longrightarrow (infty\text{-}plus\ pl,(+)) \in \langle R\rangle infty\text{-}rel \to \langle R\rangle infty\text{-}rel \to \langle R\rangle infty\text{-}rel$
⟨*proof*⟩

11.1.2 Graph

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

definition *graph-more-rel-internal-def*:
$graph\text{-}more\text{-}rel\ Rm\ Rv\ Rw \equiv \{\ (g,g').$
$(graph.nodes\ g,\ graph.nodes\ g') \in \langle Rv\rangle set\text{-}rel$
$\wedge\ (graph.edges\ g,\ graph.edges\ g') \in \langle\langle Rv,\langle Rw,Rv\rangle prod\text{-}rel\rangle prod\text{-}rel\rangle set\text{-}rel$
$\wedge\ (graph.more\ g,\ graph.more\ g') \in Rm\}$

lemma *graph-more-rel-def*[*refine-rel-defs*]:
$\langle Rm,Rv,Rw\rangle graph\text{-}more\text{-}rel \equiv \{\ (g,g').$
$(graph.nodes\ g,\ graph.nodes\ g') \in \langle Rv\rangle set\text{-}rel$
$\wedge\ (graph.edges\ g,\ graph.edges\ g') \in \langle\langle Rv,\langle Rw,Rv\rangle prod\text{-}rel\rangle prod\text{-}rel\rangle set\text{-}rel$
$\wedge\ (graph.more\ g,\ graph.more\ g') \in Rm\}$
⟨*proof*⟩

abbreviation $graph\text{-}rel \equiv \langle unit\text{-}rel\rangle graph\text{-}more\text{-}rel$
lemmas *graph-rel-def* = *graph-more-rel-def*[**where** $Rm{=}unit\text{-}rel$, *simplified*]

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

lemma *graph-more-rel-sv*[*relator-props*]:
$[\![single\text{-}valued\ Rm;\ single\text{-}valued\ Rv;\ single\text{-}valued\ Rw]\!]$
$\Longrightarrow single\text{-}valued\ (\langle Rm,Rv,Rw\rangle graph\text{-}more\text{-}rel)$
⟨*proof*⟩

lemma [*autoref-itype*]:
$graph.nodes ::_i \langle Iv,Iw\rangle_i i\text{-}graph \to_i \langle Iv\rangle_i i\text{-}set$
⟨*proof*⟩

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* $::_i$ ⟨*Iv*,*Iw*⟩$_i$*i-graph* $\rightarrow_i$ ⟨⟨*Iv*⟩$_i$*i-list*⟩$_i$*i-nres*
⟨*proof*⟩
lemma *nodes-to-list-pat*[*autoref-op-pat*]: *it-to-sorted-list* (λ- -. *True*) (*graph.nodes g*) ≡ *nodes-to-list g*
⟨*proof*⟩

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 $::_i$ ⟨*Iv*,*Iw*⟩$_i$*i-graph* $\rightarrow_i$ *Iv* $\rightarrow_i$ ⟨⟨⟨*Iw*,*Iv*⟩$_i$*i-prod*⟩$_i$*i-list*⟩$_i$*i-nres* ⟨*proof*⟩
lemma *succ-to-list-pat*[*autoref-op-pat*]: *it-to-sorted-list* (λ- -. *True*) (*Graph.succ g v*) ≡ *succ-to-list g v*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *rel-id*[*simp*]: ⟨*Id*,*Id*⟩*rel* = *br* α *invar* ⟨*proof*⟩

lemma *rel-sv*[*relator-props*]:
⟦*single-valued Rv*; *single-valued Rw*⟧ ⟹ *single-valued* (⟨*Rv*,*Rw*⟩*rel*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
    ⟨*proof*⟩

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

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

  term *mpath-weight'*
  lemma [*autoref-rules*]:
    (*mpath-weight'*,*mpath-weight'*)
      ∈ ⟨⟨*v-rel*$\times_r$*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*
    ⟨*proof*⟩

  term *mdinit*

  lemmas [*autoref-tyrel*] =
    ty-REL[**where** *R*=*v-rel*]
    ty-REL[**where** *R*=*w-rel*]
    ty-REL[**where** *R*=⟨*w-rel*⟩*infty-rel*]
    ty-REL[**where** *R*=⟨*v-rel*,⟨*w-rel*⟩*infty-rel*⟩*qw*.*rel*]
    ty-REL[**where** *R*=⟨*v-rel*,⟨*v-rel*$\times_r$*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*
  ⟨*proof*⟩

end

context *dijkstraC*
begin

  concrete-definition *cdijkstra* **for** *g ?v0.0*
    uses *dijkstraC-fixg.cdijkstra-refines-aux*
    [*of g-ops mr-ops qw-ops*]

    term *cdijkstra*
end

context *dijkstraC-fixg*
begin

  term *cdijkstra*
  term *mdijkstra*

  lemma *cdijkstra-refines*:
    RETURN (*cdijkstra g v0*) ≤ ⇓(*build-rel mr.α mr.invar*) *mdijkstra*
    ⟨*proof*⟩

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

end

theorem (**in** *dijkstraC*) *cdijkstra-correct*:
  assumes *INV*: *g.invar g*
  assumes *V0*: *v0* ∈ *nodes* (*g.α g*)
  assumes *nonneg-weights*: ⋀*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*⟩

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

⟨*ML*⟩
interpretation *hrf*: *dijkstraC hlg-ops rm-ops aluprioi-ops*
⟨*proof*⟩
⟨*ML*⟩

definition *hrf-dijkstra* ≡ *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*)

$(\lambda\ (g',v',s').\ ((v,s',v')\#g',v'+1,\mathit{lcg\text{-}next}\ s'))$
$(g,0,s))$
in $(g'',v+1,s''))$
$([],0,\mathit{lcg\text{-}next}\ \mathit{seed})))$

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

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

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

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

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

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

⟨*ML*⟩

end

References

[1] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik 1*, pages 269–271, 1959.

[2] F. Haftmann. *Code Generation from Specifications in Higher Order Logic*. PhD thesis, Technische Universität München, 2009.

[3] F. Haftmann and T. Nipkow. Code generation via higher-order rewrite systems. In *Functional and Logic Programming (FLOPS 2010)*, LNCS. Springer, 2010.

[4] P. Lammich. Collections framework. In G. Klein, T. Nipkow, and L. Paulson, editors, *Archive of Formal Proofs*. http://isa-afp.org/entries/collections.shtml, Dec. 2009. Formal proof development.

[5] P. Lammich. Refinement for monadic programs. 2011. Submitted to AFP.

[6] P. Lammich and A. Lochbihler. The Isabelle collections framework. In M. Kaufmann and L. Paulson, editors, *Interactive Theorem Prov-*

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

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