Size-Change Termination

Alexander Krauss

November 22, 2007

1 Miscellaneous Tools for Size-Change Termination

theory *Misc-Tools*
imports *Main*
begin

1.1 Searching in lists

fun *index-of* :: $'a$ *list* $\Rightarrow$ $'a \Rightarrow$ *nat*
where
  index-of [] $c = 0$
| *index-of* $(x\#xs)$ c = (*if* $x = c$ *then* *0* *else* *Suc* (*index-of* *xs* *c*))

lemma *index-of-member*:
  $(x \in set\ l) \Longrightarrow (l\ !\ index\text{-}of\ l\ x = x)$
  by (*induct* *l*) *auto*

lemma *index-of-length*:
  $(x \in set\ l) = (index\text{-}of\ l\ x < length\ l)$
  by (*induct* *l*) *auto*

1.2 Some reasoning tools

lemma *three-cases*:
  assumes $a1 \Longrightarrow thesis$
  assumes $a2 \Longrightarrow thesis$
  assumes $a3 \Longrightarrow thesis$
  assumes $\bigwedge R.\ [\![a1 \Longrightarrow R;\ a2 \Longrightarrow R;\ a3 \Longrightarrow R]\!] \Longrightarrow R$
  shows *thesis*
  using *assms*
  by *auto*

1.3 Sequences

types
  $'a$ *sequence* = *nat* $\Rightarrow$ $'a$

1.3.1 Increasing sequences

definition
  increasing :: $(nat \Rightarrow nat) \Rightarrow bool$ **where**
  increasing $s = (\forall\, i\, j.\ i < j \longrightarrow s\ i < s\ j)$

lemma *increasing-strict*:
  assumes *increasing s*
  assumes $i < j$
  shows $s\ i < s\ j$
  using *assms*
  unfolding *increasing-def* **by** *simp*

lemma *increasing-weak*:
  assumes *increasing s*
  assumes $i \leq j$
  shows $s\ i \leq s\ j$
  using *assms increasing-strict*[*of s i j*]
  by (*cases* $i < j$) *auto*

lemma *increasing-inc*:
  assumes *increasing s*
  shows $n \leq s\ n$
proof (*induct n*)
  case *0* **then show** *?case* **by** *simp*
next
  case (*Suc n*)
  with *increasing-strict* [*OF* ‹*increasing s*›, *of n Suc n*]
  show *?case* **by** *auto*
qed

lemma *increasing-bij*:
  assumes [*simp*]: *increasing s*
  shows $(s\ i < s\ j) = (i < j)$
proof
  assume $s\ i < s\ j$
  show $i < j$
  proof (*rule classical*)
    assume ¬ *?thesis*
    hence $j \leq i$ **by** *arith*
    with *increasing-weak* **have** $s\ j \leq s\ i$ **by** *simp*
    with ‹$s\ i < s\ j$› **show** *?thesis* **by** *simp*
  qed
qed (*simp add*:*increasing-strict*)

1.3.2 Sections induced by an increasing sequence

abbreviation
  section $s\ i == \{s\ i\ ..< s\ (Suc\ i)\}$

```
definition
  section-of s n = (LEAST i. n < s (Suc i))

lemma section-help:
  assumes increasing s
  shows ∃ i. n < s (Suc i)
proof −
  have n ≤ s n
    using ‹increasing s› by (rule increasing-inc)
  also have . . . < s (Suc n)
    using ‹increasing s› increasing-strict by simp
  finally show ?thesis ..
qed

lemma section-of2:
  assumes increasing s
  shows n < s (Suc (section-of s n))
  unfolding section-of-def
  by (rule LeastI-ex) (rule section-help [OF ‹increasing s›])

lemma section-of1:
  assumes [simp, intro]: increasing s
  assumes s i ≤ n
  shows s (section-of s n) ≤ n
proof (rule classical)
  let ?m = section-of s n

  assume ¬ ?thesis
  hence a: n < s ?m by simp

  have nonzero: ?m ≠ 0
  proof
    assume ?m = 0
    from increasing-weak have s 0 ≤ s i by simp
    also note ‹. . . ≤ n›
    finally show False using ‹?m = 0› ‹n < s ?m› by simp
  qed
  with a have n < s (Suc (?m − 1)) by simp
  with Least-le have ?m ≤ ?m − 1
    unfolding section-of-def .
  with nonzero show ?thesis by simp
qed

lemma section-of-known:
  assumes [simp]: increasing s
  assumes in-sect: k ∈ section s i
  shows section-of s k = i (is ?s = i)
proof (rule classical)
  assume ¬ ?thesis
```

```
  hence ?s < i ∨ ?s > i by arith
  thus ?thesis
  proof
    assume ?s < i
    hence Suc ?s ≤ i by simp
    with increasing-weak have s (Suc ?s) ≤ s i by simp
    moreover have k < s (Suc ?s) using section-of2 by simp
    moreover from in-sect have s i ≤ k by simp
    ultimately show ?thesis by simp
  next
    assume i < ?s hence Suc i ≤ ?s by simp
    with increasing-weak have s (Suc i) ≤ s ?s by simp
    moreover
    from in-sect have s i ≤ k by simp
    with section-of1 have s ?s ≤ k by simp
    moreover from in-sect have k < s (Suc i) by simp
    ultimately show ?thesis by simp
  qed
qed

lemma in-section-of:
  assumes increasing s
  assumes s i ≤ k
  shows k ∈ section s (section-of s k)
  using assms
  by (auto intro:section-of1 section-of2)

end
```

2 Kleene Algebras

```
theory Kleene-Algebras
imports Main
begin
```

A type class of kleene algebras

```
class star = type +
  fixes star :: 'a ⇒ 'a

class idem-add = ab-semigroup-add +
  assumes add-idem [simp]: x + x = x

lemma add-idem2[simp]: (x::'a::idem-add) + (x + y) = x + y
  unfolding add-assoc[symmetric]
  by simp

class order-by-add = idem-add + ord +
```

assumes *order-def*: $a \leq b \longleftrightarrow a + b = b$
assumes *strict-order-def*: $a < b \longleftrightarrow a \leq b \wedge a \neq b$

lemma *ord-simp1* [*simp*]: $(x::'a::\mathit{order\text{-}by\text{-}add}) \leq y \Longrightarrow x + y = y$
unfolding *order-def* **.**
lemma *ord-simp2* [*simp*]: $(x::'a::\mathit{order\text{-}by\text{-}add}) \leq y \Longrightarrow y + x = y$
unfolding *order-def add-commute* **.**
lemma *ord-intro*: $(x::'a::\mathit{order\text{-}by\text{-}add}) + y = y \Longrightarrow x \leq y$
unfolding *order-def* **.**

instance *order-by-add* $\subseteq$ *order*
proof
fix $x\ y\ z :: 'a$
show $x \leq x$ **unfolding** *order-def* **by** *simp*

show $[\![x \leq y;\ y \leq z]\!] \Longrightarrow x \leq z$
proof (*rule ord-intro*)
assume $x \leq y$ $y \leq z$

have $x + z = x + y + z$ **by** (*simp add*:$\langle y \leq z\rangle$ *add-assoc*)
also have $\ldots = y + z$ **by** (*simp add*:$\langle x \leq y\rangle$)
also have $\ldots = z$ **by** (*simp add*:$\langle y \leq z\rangle$)
finally show $x + z = z$ **.**
qed

show $[\![x \leq y;\ y \leq x]\!] \Longrightarrow x = y$ **unfolding** *order-def*
by (*simp add*:*add-commute*)
show $x < y \longleftrightarrow x \leq y \wedge x \neq y$ **by** (*fact strict-order-def*)
qed

class *pre-kleene* = *semiring-1* + *order-by-add*

instance *pre-kleene* $\subseteq$ *pordered-semiring*
proof
fix $x\ y\ z :: 'a$

assume $x \leq y$

show $z + x \leq z + y$
proof (*rule ord-intro*)
have $z + x + (z + y) = x + y + z$ **by** (*simp add*:*add-ac*)
also have $\ldots = z + y$ **by** (*simp add*:$\langle x \leq y\rangle$ *add-ac*)
finally show $z + x + (z + y) = z + y$ **.**
qed

show $z * x \leq z * y$
proof (*rule ord-intro*)
from $\langle x \leq y\rangle$ **have** $z * (x + y) = z * y$ **by** *simp*

thus $z * x + z * y = z * y$ **by** (*simp add*:*right-distrib*)
qed

show $x * z \leq y * z$
proof (*rule ord-intro*)
from ‹$x \leq y$› **have** $(x + y) * z = y * z$ **by** *simp*
thus $x * z + y * z = y * z$ **by** (*simp add*:*left-distrib*)
qed
qed

class *kleene* = *pre-kleene* + *star* +
assumes *star1*: $1 + a * star\ a \leq star\ a$
and *star2*: $1 + star\ a * a \leq star\ a$
and *star3*: $a * x \leq x \Longrightarrow star\ a * x \leq x$
and *star4*: $x * a \leq x \Longrightarrow x * star\ a \leq x$

class *kleene-by-complete-lattice* = *pre-kleene*
+ *complete-lattice* + *recpower* + *star* +
assumes *star-cont*: $a * star\ b * c = SUPR\ UNIV\ (\lambda n.\ a * b\ \hat{}\ n * c)$

lemma *plus-leI*:
fixes x :: *'a* :: *order-by-add*
shows $x \leq z \Longrightarrow y \leq z \Longrightarrow x + y \leq z$
unfolding *order-def* **by** (*simp add*:*add-assoc*)

lemma *le-SUPI'*:
fixes l :: *'a* :: *complete-lattice*
assumes $l \leq M\ i$
shows $l \leq (SUP\ i.\ M\ i)$
using *assms* **by** (*rule order-trans*) (*rule le-SUPI* [*OF UNIV-I*])

lemma *zero-minimum*[*simp*]: $(0::'a::pre\text{-}kleene) \leq x$
unfolding *order-def* **by** *simp*

instance *kleene-by-complete-lattice* ⊆ *kleene*
proof

fix $a\ x$ :: *'a*

have [*simp*]: $1 \leq star\ a$
unfolding *star-cont*[*of 1 a 1*, *simplified*]
by (*subst power-0*[*symmetric*]) (*rule le-SUPI* [*OF UNIV-I*])

show $1 + a * star\ a \leq star\ a$
apply (*rule plus-leI*, *simp*)
apply (*simp add*:*star-cont*[*of a a 1*, *simplified*])
apply (*simp add*:*star-cont*[*of 1 a 1*, *simplified*])
apply (*subst power-Suc*[*symmetric*])
by (*intro SUP-leI le-SUPI UNIV-I*)

```
  show 1 + star a * a ≤ star a
    apply (rule plus-leI, simp)
    apply (simp add:star-cont[of 1 a a, simplified])
    apply (simp add:star-cont[of 1 a 1, simplified])
      by (auto intro: SUP-leI le-SUPI UNIV-I simp add: power-Suc[symmetric]
power-commutes)

  show a * x ≤ x ⟹ star a * x ≤ x
  proof -
    assume a: a * x ≤ x

    {
      fix n
      have a ^ (Suc n) * x ≤ a ^ n * x
      proof (induct n)
        case 0 thus ?case by (simp add:a power-Suc)
      next
        case (Suc n)
        hence a * (a ^ Suc n * x) ≤ a * (a ^ n * x)
          by (auto intro: mult-mono)
        thus ?case
          by (simp add:power-Suc mult-assoc)
      qed
    }
    note a = this

    {
      fix n have a ^ n * x ≤ x
      proof (induct n)
        case 0 show ?case by simp
      next
        case (Suc n) with a[of n]
        show ?case by simp
      qed
    }
    note b = this

    show star a * x ≤ x
      unfolding star-cont[of 1 a x, simplified]
      by (rule SUP-leI) (rule b)
  qed

  show x * a ≤ x ⟹ x * star a ≤ x
  proof -
    assume a: x * a ≤ x

    {
      fix n
```

```
      have x ∗ a ^ (Suc n) ≤ x ∗ a ^ n
      proof (induct n)
        case 0 thus ?case by (simp add:a power-Suc)
      next
        case (Suc n)
        hence (x ∗ a ^ Suc n) ∗ a  ≤ (x ∗ a ^ n) ∗ a
          by (auto intro: mult-mono)
        thus ?case
          by (simp add:power-Suc power-commutes mult-assoc)
      qed
    }
    note a = this

    {
      fix n have x ∗ a ^ n ≤ x
      proof (induct n)
        case 0 show ?case by simp
      next
        case (Suc n) with a[of n]
        show ?case by simp
      qed
    }
    note b = this

    show x ∗ star a ≤ x
      unfolding star-cont[of x a 1, simplified]
      by (rule SUP-leI) (rule b)
  qed
qed

lemma less-add[simp]:
  fixes a b :: 'a :: order-by-add
  shows a ≤ a + b
  and b ≤ a + b
  unfolding order-def
  by (auto simp:add-ac)

lemma add-est1:
  fixes a b c :: 'a :: order-by-add
  assumes a: a + b ≤ c
  shows a ≤ c
  using less-add(1) a
  by (rule order-trans)

lemma add-est2:
  fixes a b c :: 'a :: order-by-add
  assumes a: a + b ≤ c
  shows b ≤ c
  using less-add(2) a
```

by (*rule order-trans*)

lemma *star3'*:
fixes *a b x* :: *'a* :: *kleene*
assumes *a*: $b + a * x \leq x$
shows $star\ a * b \leq x$
proof (*rule order-trans*)
from *a* **have** $b \leq x$ **by** (*rule add-est1*)
show $star\ a * b \leq star\ a * x$
by (*rule mult-mono*) (*auto simp*:‹$b \leq x$›)

from *a* **have** $a * x \leq x$ **by** (*rule add-est2*)
with *star3* **show** $star\ a * x \leq x$ **.**
qed

lemma *star4'*:
fixes *a b x* :: *'a* :: *kleene*
assumes *a*: $b + x * a \leq x$
shows $b * star\ a \leq x$
proof (*rule order-trans*)
from *a* **have** $b \leq x$ **by** (*rule add-est1*)
show $b * star\ a \leq x * star\ a$
by (*rule mult-mono*) (*auto simp*:‹$b \leq x$›)

from *a* **have** $x * a \leq x$ **by** (*rule add-est2*)
with *star4* **show** $x * star\ a \leq x$ **.**
qed

lemma *star-idemp*:
fixes *x* :: *'a* :: *kleene*
shows $star\ (star\ x) = star\ x$
oops

lemma *star-unfold-left*:
fixes *a* :: *'a* :: *kleene*
shows $1 + a * star\ a = star\ a$
proof (*rule order-antisym*, *rule star1*)

have $1 + a * (1 + a * star\ a) \leq 1 + a * star\ a$
apply (*rule add-mono*, *rule*)
apply (*rule mult-mono*, *auto*)
apply (*rule star1*)
done

with *star3'* **have** $star\ a * 1 \leq 1 + a * star\ a$ **.**
thus $star\ a \leq 1 + a * star\ a$ **by** *simp*

qed

lemma *star-unfold-right*:
  fixes a :: *'a* :: *kleene*
  shows $1 + star\ a * a = star\ a$
proof (*rule order-antisym*, *rule star2*)

  have $1 + (1 + star\ a * a) * a \leq 1 + star\ a * a$
    apply (*rule add-mono*, *rule*)
    apply (*rule mult-mono*, *auto*)
    apply (*rule star2*)
    done

  with *star4'* **have** $1 * star\ a \leq 1 + star\ a * a$ **.**
  thus $star\ a \leq 1 + star\ a * a$ **by** *simp*
qed

lemma *star-zero*[*simp*]:
  shows *star* (*0*::*'a*::*kleene*) = *1*
  by (*rule star-unfold-left*[*of 0*, *simplified*])

lemma *star-commute*:
  fixes *a b x* :: *'a* :: *kleene*
  assumes *a*: $a * x = x * b$
  shows $star\ a * x = x * star\ b$
proof (*rule order-antisym*)

  show $star\ a * x \leq x * star\ b$
  proof (*rule star3'*, *rule order-trans*)

    from *a* **have** $a * x \leq x * b$ **by** *simp*
    hence $a * x * star\ b \leq x * b * star\ b$
      by (*rule mult-mono*) *auto*
    thus $x + a * (x * star\ b) \leq x + x * b * star\ b$
      using *add-mono* **by** (*auto simp*: *mult-assoc*)

    show $\ldots \leq x * star\ b$
    proof −
      have $x * (1 + b * star\ b) \leq x * star\ b$
        by (*rule mult-mono*[*OF - star1*]) *auto*
      thus *?thesis*
        by (*simp add*:*right-distrib mult-assoc*)
    qed
  qed

  show $x * star\ b \leq star\ a * x$
  proof (*rule star4'*, *rule order-trans*)

```
    from a have b: x * b ≤ a * x by simp
    have star a * x * b ≤ star a * a * x
      unfolding mult-assoc
      by (rule mult-mono[OF - b]) auto
    thus x + star a * x * b ≤ x + star a * a * x
      using add-mono by auto

    show ... ≤ star a * x
    proof −
      have (1 + star a * a) * x ≤ star a * x
        by (rule mult-mono[OF star2]) auto
      thus ?thesis
        by (simp add:left-distrib mult-assoc)
    qed
  qed
qed

lemma star-assoc:
  fixes c d :: 'a :: kleene
  shows star (c * d) * c = c * star (d * c)
  by (auto simp:mult-assoc star-commute)

lemma star-dist:
  fixes a b :: 'a :: kleene
  shows star (a + b) = star a * star (b * star a)
  oops

lemma star-one:
  fixes a p p' :: 'a :: kleene
  assumes p * p' = 1 and p' * p = 1
  shows p' * star a * p = star (p' * a * p)
proof −
  from assms
  have p' * star a * p = p' * star (p * p' * a) * p
    by simp
  also have ... = p' * p * star (p' * a * p)
    by (simp add: mult-assoc star-assoc)
  also have ... = star (p' * a * p)
    by (simp add: assms)
  finally show ?thesis .
qed

lemma star-mono:
  fixes x y :: 'a :: kleene
  assumes x ≤ y
  shows star x ≤ star y
  oops
```

lemma *x-less-star*[*simp*]:
  fixes x :: $'a$ :: *kleene*
  shows $x \le x * star\ a$
proof −
  have $x \le x * (1 + a * star\ a)$ **by** (*simp add*:*right-distrib*)
  also have ... $= x * star\ a$ **by** (*simp only*: *star-unfold-left*)
  finally show *?thesis* **.**
qed

2.1 Transitive Closure

definition
  $tcl\ (x::'a::kleene) = star\ x * x$

lemma *tcl-zero*:
  $tcl\ (0::'a::kleene) = 0$
  unfolding *tcl-def* **by** *simp*

lemma *tcl-unfold-right*: $tcl\ a = a + tcl\ a * a$
proof −
  from *star-unfold-right*[*of a*]
  have $a * (1 + star\ a * a) = a * star\ a$ **by** *simp*
  from *this*[*simplified right-distrib*, *simplified*]
  show *?thesis*
    by (*simp add*:*tcl-def star-commute mult-ac*)
qed

lemma *less-tcl*: $a \le tcl\ a$
proof −
  have $a \le a + tcl\ a * a$ **by** *simp*
  also have ... $= tcl\ a$ **by** (*rule tcl-unfold-right*[*symmetric*])
  finally show *?thesis* **.**
qed

2.2 Naive Algorithm to generate the transitive closure

function (*default* $\lambda x.\ 0$, *tailrec*, *domintros*)
  mk-tcl :: $('a::\{plus,times,ord,zero\}) \Rightarrow 'a \Rightarrow 'a$
where
  $mk\text{-}tcl\ A\ X = (if\ X * A \le X\ then\ X\ else\ mk\text{-}tcl\ A\ (X + X * A))$
  by *pat-completeness simp*

declare *mk-tcl.simps*[*simp del*]

lemma *mk-tcl-code*[*code*]:
  $mk\text{-}tcl\ A\ X =$

```
  (let XA = X * A
  in if XA ≤ X then X else mk-tcl A (X + XA))
  unfolding mk-tcl.simps[of A X] Let-def ..

lemma mk-tcl-lemma1:
  fixes X :: 'a :: kleene
  shows (X + X * A) * star A = X * star A
proof -
  have A * star A ≤ 1 + A * star A by simp
  also have ... = star A by (simp add:star-unfold-left)
  finally have star A + A * star A = star A by simp
  hence X * (star A + A * star A) = X * star A by simp
  thus ?thesis by (simp add:left-distrib right-distrib mult-ac)
qed

lemma mk-tcl-lemma2:
  fixes X :: 'a :: kleene
  shows X * A ≤ X ⟹ X * star A = X
  by (rule order-antisym) (auto simp:star4)

lemma mk-tcl-correctness:
  fixes A X :: 'a :: {kleene}
  assumes mk-tcl-dom (A, X)
  shows mk-tcl A X = X * star A
  using assms
  by induct (auto simp:mk-tcl-lemma1 mk-tcl-lemma2)

lemma graph-implies-dom: mk-tcl-graph x y ⟹ mk-tcl-dom x
  by (rule mk-tcl-graph.induct) (auto intro:accp.accI elim:mk-tcl-rel.cases)

lemma mk-tcl-default: ¬ mk-tcl-dom (a,x) ⟹ mk-tcl a x = 0
  unfolding mk-tcl-def
  by (rule fundef-default-value[OF mk-tcl-sum-def graph-implies-dom])
```

We can replace the dom-Condition of the correctness theorem with something executable

```
lemma mk-tcl-correctness2:
  fixes A X :: 'a :: {kleene}
  assumes mk-tcl A A ≠ 0
  shows mk-tcl A A = tcl A
  using assms mk-tcl-default mk-tcl-correctness
  unfolding tcl-def
  by (auto simp:star-commute)

end
```

3 General Graphs as Sets

theory *Graphs*
imports *Main Misc-Tools Kleene-Algebras*
begin

3.1 Basic types, Size Change Graphs

datatype $('a, 'b)$ *graph* =
  Graph $('a \times 'b \times 'a)$ *set*

fun *dest-graph* :: $('a, 'b)$ *graph* $\Rightarrow$ $('a \times 'b \times 'a)$ *set*
  where *dest-graph* (*Graph* G) = G

lemma *graph-dest-graph*[*simp*]:
  Graph (*dest-graph* G) = G
  by (*cases* G) *simp*

lemma *split-graph-all*:
  $(\bigwedge gr.\ PROP\ P\ gr) \equiv (\bigwedge set.\ PROP\ P\ (Graph\ set))$
proof
  fix *set*
  assume $\bigwedge gr.\ PROP\ P\ gr$
  then show $PROP\ P\ (Graph\ set)$ **.**
next
  fix *gr*
  assume $\bigwedge set.\ PROP\ P\ (Graph\ set)$
  then have $PROP\ P\ (Graph\ (dest\text{-}graph\ gr))$ **.**
  then show $PROP\ P\ gr$ **by** *simp*
qed

definition
  has-edge :: $('n,'e)$ *graph* $\Rightarrow$ $'n \Rightarrow 'e \Rightarrow 'n \Rightarrow bool$
$(\text{-} \vdash \text{-} \leadsto^{\text{-}} \text{-})$
where
  has-edge $G\ n\ e\ n' = ((n, e, n') \in dest\text{-}graph\ G)$

3.2 Graph composition

fun *grcomp* :: $('n, 'e::times)$ *graph* $\Rightarrow$ $('n, 'e)$ *graph* $\Rightarrow$ $('n, 'e)$ *graph*
where
  grcomp (*Graph* G) (*Graph* H) =
  Graph $\{(p,b,q) \mid p\ b\ q.$
  $(\exists k\ e\ e'.\ (p,e,k)\in G \wedge (k,e',q)\in H \wedge b = e * e')\}$

declare *grcomp.simps*[*code del*]

lemma *graph-ext*:

```
  assumes ⋀n e n'. has-edge G n e n' = has-edge H n e n'
  shows G = H
  using assms
  by (cases G, cases H) (auto simp:split-paired-all has-edge-def)


instance graph :: (type, type) {comm-monoid-add}
  graph-zero-def: 0 == Graph {}
  graph-plus-def: G + H == Graph (dest-graph G ∪ dest-graph H)
proof
  fix x y z :: ('a,'b) graph

  show x + y + z = x + (y + z)
   and x + y = y + x
   and 0 + x = x
  unfolding graph-plus-def graph-zero-def
  by auto
qed

lemmas [code func del] = graph-plus-def

instance graph :: (type, type) {distrib-lattice, complete-lattice}
  graph-leq-def: G ≤ H ≡ dest-graph G ⊆ dest-graph H
  graph-less-def: G < H ≡ dest-graph G ⊂ dest-graph H
  inf G H ≡ Graph (dest-graph G ∩ dest-graph H)
  sup G H ≡ G + H
  Inf-graph-def: Inf ≡ λGs. Graph (⋂(dest-graph ` Gs))
  Sup-graph-def: Sup ≡ λGs. Graph (⋃(dest-graph ` Gs))
proof
  fix x y z :: ('a,'b) graph
  fix A :: ('a, 'b) graph set

  show (x < y) = (x ≤ y ∧ x ≠ y)
    unfolding graph-leq-def graph-less-def
    by (cases x, cases y) auto

  show x ≤ x unfolding graph-leq-def ..

  { assume x ≤ y y ≤ z
    with order-trans show x ≤ z
      unfolding graph-leq-def . }

  { assume x ≤ y y ≤ x thus x = y
      unfolding graph-leq-def
      by (cases x, cases y) simp }

  show inf x y ≤ x inf x y ≤ y
    unfolding inf-graph-def graph-leq-def
    by auto
```

{ **assume** $x \leq y$ $x \leq z$ **thus** $x \leq inf\ y\ z$
  unfolding *inf-graph-def graph-leq-def*
  by *auto* }

show $x \leq sup\ x\ y$ $y \leq sup\ x\ y$
  unfolding *sup-graph-def graph-leq-def graph-plus-def* **by** *auto*

{ **assume** $y \leq x$ $z \leq x$ **thus** $sup\ y\ z \leq x$
  unfolding *sup-graph-def graph-leq-def graph-plus-def* **by** *auto* }

show $sup\ x\ (inf\ y\ z) = inf\ (sup\ x\ y)\ (sup\ x\ z)$
  unfolding *inf-graph-def sup-graph-def graph-leq-def graph-plus-def* **by** *auto*

{ **assume** $x \in A$ **thus** $Inf\ A \leq x$
  unfolding *Inf-graph-def graph-leq-def* **by** *auto* }

{ **assume** $\bigwedge x.\ x \in A \Longrightarrow z \leq x$ **thus** $z \leq Inf\ A$
  unfolding *Inf-graph-def graph-leq-def* **by** *auto* }

{ **assume** $x \in A$ **thus** $x \leq Sup\ A$
  unfolding *Sup-graph-def graph-leq-def* **by** *auto* }

{ **assume** $\bigwedge x.\ x \in A \Longrightarrow x \leq z$ **thus** $Sup\ A \leq z$
  unfolding *Sup-graph-def graph-leq-def* **by** *auto* }
qed

lemmas [*code func del*] = *graph-leq-def graph-less-def*
  inf-graph-def sup-graph-def Inf-graph-def Sup-graph-def

lemma *in-grplus*:
  $has\text{-}edge\ (G + H)\ p\ b\ q = (has\text{-}edge\ G\ p\ b\ q \vee has\text{-}edge\ H\ p\ b\ q)$
  by (*cases G*, *cases H*, *auto simp*:*has-edge-def graph-plus-def*)

lemma *in-grzero*:
  $has\text{-}edge\ 0\ p\ b\ q = False$
  by (*simp add*:*graph-zero-def has-edge-def*)

3.2.1 Multiplicative Structure

instance *graph* :: (*type*, *times*) *mult-zero*
  graph-mult-def: $G * H == grcomp\ G\ H$
proof
  fix $a :: ('a, 'b)\ graph$

  show $0 * a = 0$
    unfolding *graph-mult-def graph-zero-def*
    by (*cases a*) (*simp add*:*grcomp.simps*)
  show $a * 0 = 0$

```
    unfolding graph-mult-def graph-zero-def
    by (cases a) (simp add:grcomp.simps)
qed

lemmas [code func del] = graph-mult-def

instance graph :: (type, one) one
  graph-one-def: 1 == Graph { (x, 1, x) |x. True} ..

lemma in-grcomp:
  has-edge (G * H) p b q
  = (∃ k e e'. has-edge G p e k ∧ has-edge H k e' q ∧ b = e * e')
  by (cases G, cases H) (auto simp:graph-mult-def has-edge-def image-def)

lemma in-grunit:
  has-edge 1 p b q = (p = q ∧ b = 1)
  by (auto simp:graph-one-def has-edge-def)

instance graph :: (type, semigroup-mult) semigroup-mult
proof
  fix G1 G2 G3 :: ('a,'b) graph

  show G1 * G2 * G3 = G1 * (G2 * G3)
  proof (rule graph-ext, rule trans)
    fix p J q
    show has-edge ((G1 * G2) * G3) p J q =
      (∃ G i H j I.
      has-edge G1 p G i
      ∧ has-edge G2 i H j
      ∧ has-edge G3 j I q
      ∧ J = (G * H) * I)
      by (simp only:in-grcomp) blast
    show ... = has-edge (G1 * (G2 * G3)) p J q
      by (simp only:in-grcomp mult-assoc) blast
  qed
qed

fun grpow :: nat ⇒ ('a::type, 'b::monoid-mult) graph ⇒ ('a, 'b) graph
where
  grpow 0 A = 1
| grpow (Suc n) A = A * (grpow n A)

instance graph :: (type, monoid-mult)
  {semiring-1,idem-add,recpower,star}
  graph-pow-def: A ^ n == grpow n A
  graph-star-def: star G == (SUP n. G ^ n)
proof
  fix a b c :: ('a, 'b) graph
```

show $1 * a = a$
  by (*rule graph-ext*) (*auto simp*:*in-grcomp in-grunit*)
show $a * 1 = a$
  by (*rule graph-ext*) (*auto simp*:*in-grcomp in-grunit*)

show $(a + b) * c = a * c + b * c$
  by (*rule graph-ext*, *simp add*:*in-grcomp in-grplus*) *blast*

show $a * (b + c) = a * b + a * c$
  by (*rule graph-ext*, *simp add*:*in-grcomp in-grplus*) *blast*

show $(0::('a,'b)\ graph) \neq 1$ **unfolding** *graph-zero-def graph-one-def*
  by *simp*

show $a + a = a$ **unfolding** *graph-plus-def* **by** *simp*

show $a\ \hat{}\ 0 = 1$ $\bigwedge n.\ a\ \hat{}\ (Suc\ n) = a * a\ \hat{}\ n$
  unfolding *graph-pow-def* **by** *simp-all*
qed

lemma *graph-leqI*:
  assumes $\bigwedge n\ e\ n'.\ has\text{-}edge\ G\ n\ e\ n' \Longrightarrow has\text{-}edge\ H\ n\ e\ n'$
  shows $G \leq H$
  using *assms*
  unfolding *graph-leq-def has-edge-def*
  by *auto*

lemma *in-graph-plusE*:
  assumes $has\text{-}edge\ (G + H)\ n\ e\ n'$
  assumes $has\text{-}edge\ G\ n\ e\ n' \Longrightarrow P$
  assumes $has\text{-}edge\ H\ n\ e\ n' \Longrightarrow P$
  shows P
  using *assms*
  by (*auto simp*: *in-grplus*)

lemma *in-graph-compE*:
  assumes *GH*: $has\text{-}edge\ (G * H)\ n\ e\ n'$
  obtains *e1 k e2*
  where $has\text{-}edge\ G\ n\ e1\ k$ $has\text{-}edge\ H\ k\ e2\ n'$ $e = e1 * e2$
  using *GH*
  by (*auto simp*: *in-grcomp*)

lemma
  assumes $x \in S\ k$
  shows $x \in (\bigcup k.\ S\ k)$
  using *assms* **by** *blast*

lemma *graph-union-least*:
  assumes $\bigwedge n.\ Graph\ (G\ n) \leq C$

shows *Graph* ($\bigcup n.$ *G n*) $\leq$ *C*
using *assms* **unfolding** *graph-leq-def*
by *auto*

lemma *Sup-graph-eq*:
(*SUP n. Graph* (*G n*)) = *Graph* ($\bigcup n.$ *G n*)
proof (*rule order-antisym*)
show (*SUP n. Graph* (*G n*)) $\leq$ *Graph* ($\bigcup n.$ *G n*)
by (*rule SUP-leI*) (*auto simp add*: *graph-leq-def*)

show *Graph* ($\bigcup n.$ *G n*) $\leq$ (*SUP n. Graph* (*G n*))
by (*rule graph-union-least*, *rule le-SUPI'*, *rule*)
qed

lemma *has-edge-leq*: *has-edge G p b q* = (*Graph* {(*p*,*b*,*q*)} $\leq$ *G*)
unfolding *has-edge-def graph-leq-def*
by (*cases G*) *simp*

lemma *Sup-graph-eq2*:
(*SUP n. G n*) = *Graph* ($\bigcup n.$ *dest-graph* (*G n*))
using *Sup-graph-eq*[*of* $\lambda n.$ *dest-graph* (*G n*), *simplified*]
by *simp*

lemma *in-SUP*:
has-edge (*SUP x. Gs x*) *p b q* = ($\exists x.$ *has-edge* (*Gs x*) *p b q*)
unfolding *Sup-graph-eq2 has-edge-leq graph-leq-def*
by *simp*

instance *graph* :: (*type*, *monoid-mult*) *kleene-by-complete-lattice*
proof
fix *a b c* :: (*'a*, *'b*) *graph*

show $a \leq b \longleftrightarrow a + b = b$ **unfolding** *graph-leq-def graph-plus-def*
by (*cases a*, *cases b*) *auto*

from *order-less-le* **show** $a < b \longleftrightarrow a \leq b \wedge a \neq b$ **.**

show $a * star\ b * c = (SUP\ n.\ a * b\ \hat{}\ n * c)$
unfolding *graph-star-def*
by (*rule graph-ext*) (*force simp*:*in-SUP in-grcomp*)
qed

lemma *in-star*:
has-edge (*star G*) *a x b* = ($\exists n.$ *has-edge* (*G* ^ *n*) *a x b*)
by (*auto simp*:*graph-star-def in-SUP*)

lemma *tcl-is-SUP*:

tcl (*G*::(*'a*::*type*, *'b*::*monoid-mult*) *graph*) =
(*SUP n*. *G* ^ (*Suc n*))
unfolding *tcl-def*
using *star-cont*[*of 1 G G*]
by (*simp add*:*power-Suc power-commutes*)

lemma *in-tcl*:
has-edge (*tcl G*) *a x b* = ($\exists$ *n*>*0*. *has-edge* (*G* ^ *n*) *a x b*)
apply (*auto simp*: *tcl-is-SUP in-SUP*)
apply (*rule-tac x* = *n* − *1* **in** *exI*, *auto*)
done

3.3 Infinite Paths

types (*'n*, *'e*) *ipath* = (*'n* × *'e*) *sequence*

definition *has-ipath* :: (*'n*, *'e*) *graph* ⇒ (*'n*, *'e*) *ipath* ⇒ *bool*
where
has-ipath G p =
($\forall$ *i*. *has-edge G* (*fst* (*p i*)) (*snd* (*p i*)) (*fst* (*p* (*Suc i*))))

3.4 Finite Paths

types (*'n*, *'e*) *fpath* = (*'n* × (*'e* × *'n*) *list*)

inductive *has-fpath* :: (*'n*, *'e*) *graph* ⇒ (*'n*, *'e*) *fpath* ⇒ *bool*
for *G* :: (*'n*, *'e*) *graph*
where
has-fpath-empty: *has-fpath G* (*n*, [])
| *has-fpath-join*: ⟦*G* ⊢ *n* $\leadsto^{e}$ *n'*; *has-fpath G* (*n'*, *es*)⟧ ⟹ *has-fpath G* (*n*, (*e*, *n'*)#*es*)

definition
end-node p =
(*if snd p* = [] *then fst p else snd* (*snd p* ! (*length* (*snd p*) − *1*)))

definition *path-nth* :: (*'n*, *'e*) *fpath* ⇒ *nat* ⇒ (*'n* × *'e* × *'n*)
where
path-nth p k = (*if k* = *0 then fst p else snd* (*snd p* ! (*k* − *1*)), *snd p* ! *k*)

lemma *endnode-nth*:
assumes *length* (*snd p*) = *Suc k*
shows *end-node p* = *snd* (*snd* (*path-nth p k*))
using *assms* **unfolding** *end-node-def path-nth-def*
by *auto*

lemma *path-nth-graph*:
assumes *k* < *length* (*snd p*)
assumes *has-fpath G p*

shows $(\lambda(n,e,n')$. *has-edge* $G\ n\ e\ n')$ (*path-nth* $p\ k$)
using *assms*
proof (*induct* k *arbitrary*: p)
case *0* **thus** *?case*
unfolding *path-nth-def* **by** (*auto elim*:*has-fpath.cases*)
next
case (*Suc* $k\ p$)

from ⟨*has-fpath* $G\ p$⟩ **show** *?case*
proof (*rule has-fpath.cases*)
case *goal1* **with** *Suc* **show** *?case* **by** *simp*
next
fix $n\ e\ n'\ es$
assume *st*: $p = (n, (e, n') \# es)$
$G \vdash n \leadsto^{e} n'$
has-fpath $G\ (n', es)$
with *Suc*
have $(\lambda(n, b, a).\ G \vdash n \leadsto^{b} a)$ (*path-nth* $(n', es)\ k$) **by** *simp*
with *st* **show** *?thesis* **by** (*cases* k, *auto simp*:*path-nth-def*)
qed
qed

lemma *path-nth-connected*:
assumes *Suc* $k <$ *length* (*snd* p)
shows *fst* (*path-nth* p (*Suc* k)) = *snd* (*snd* (*path-nth* $p\ k$))
using *assms*
unfolding *path-nth-def*
by *auto*

definition *path-loop* :: $('n, 'e)$ *fpath* $\Rightarrow$ $('n, 'e)$ *ipath* (*omega*)
where
omega $p \equiv (\lambda i.\ (\lambda(n,e,n').\ (n,e))$ (*path-nth* p (i *mod* (*length* (*snd* p)))))

lemma *fst-p0*: *fst* (*path-nth* p *0*) = *fst* p
unfolding *path-nth-def* **by** *simp*

lemma *path-loop-connect*:
assumes *fst* p = *end-node* p
and *0* $<$ *length* (*snd* p) (**is** *0* $<$ *?l*)
shows *fst* (*path-nth* p (*Suc* i *mod* (*length* (*snd* p))))
= *snd* (*snd* (*path-nth* p (i *mod length* (*snd* p))))
(**is** ... = *snd* (*snd* (*path-nth* p *?k*)))
proof −
from ⟨*0* $<$ *?l*⟩ **have** i *mod ?l* $<$ *?l* (**is** *?k* $<$ *?l*)
by *simp*

show *?thesis*
proof (*cases Suc ?k* $<$ *?l*)
case *True*

```
    hence Suc ?k ≠ ?l by simp
    with path-nth-connected[OF True]
    show ?thesis
      by (simp add:mod-Suc)
  next
    case False
    with ‹?k < ?l› have wrap: Suc ?k = ?l by simp

    hence fst (path-nth p (Suc i mod ?l)) = fst (path-nth p 0)
      by (simp add: mod-Suc)
    also from fst-p0 have ... = fst p .
    also have ... = end-node p by fact
    also have ... = snd (snd (path-nth p ?k))
      by (auto simp: endnode-nth wrap)
    finally show ?thesis .
  qed
qed

lemma path-loop-graph:
  assumes has-fpath G p
  and loop: fst p = end-node p
  and nonempty: 0 < length (snd p) (is 0 < ?l)
  shows has-ipath G (omega p)
proof −
  {
    fix i
    from ‹0 < ?l› have i mod ?l < ?l (is ?k < ?l)
      by simp
    from this and ‹has-fpath G p›
    have pk-G: (λ(n,e,n'). has-edge G n e n') (path-nth p ?k)
      by (rule path-nth-graph)

    from path-loop-connect[OF loop nonempty] pk-G
    have has-edge G (fst (omega p i)) (snd (omega p i)) (fst (omega p (Suc i)))
      unfolding path-loop-def has-edge-def split-def
      by simp
  }
  then show ?thesis by (auto simp:has-ipath-def)
qed

definition prod :: ('n, 'e::monoid-mult) fpath ⇒ 'e
where
  prod p = foldr (op ∗) (map fst (snd p)) 1

lemma prod-simps[simp]:
  prod (n, []) = 1
  prod (n, (e,n')#es) = e ∗ (prod (n',es))
unfolding prod-def
by simp-all
```

lemma *power-induces-path*:
  assumes *a*: *has-edge* $(A\ \hat{}\ k)$ *n G m*
  obtains *p*
    where *has-fpath A p*
      and $n = fst\ p\ m = end\text{-}node\ p$
      and $G = prod\ p$
      and $k = length\ (snd\ p)$
  using *a*
proof (*induct k arbitrary*:*m n G thesis*)
  case (*0 m n G*)
  let $?p = (n, [])$
  from *0* **have** $has\text{-}fpath\ A\ ?p\ m = end\text{-}node\ ?p\ G = prod\ ?p$
    by (*auto simp*:*in-grunit end-node-def intro*:*has-fpath.intros*)
  thus *?case* **using** *0* **by** (*auto simp*:*end-node-def*)
next
  case (*Suc k m n G*)
  hence *has-edge* $(A * A\ \hat{}\ k)$ *n G m*
    by (*simp add*:*power-Suc power-commutes*)
  then obtain G' *H j* **where**
    a-A: *has-edge A n* G' *j*
    and *H-pow*: *has-edge* $(A\ \hat{}\ k)$ *j H m*
    and [*simp*]: $G = G' * H$
    by (*auto simp*:*in-grcomp*)

  from *H-pow* **and** *Suc*
  obtain *p*
    where *p-path*: *has-fpath A p*
    and [*simp*]: $j = fst\ p\ m = end\text{-}node\ p\ H = prod\ p$
    $k = length\ (snd\ p)$
    by *blast*

  let $?p' = (n, (G', j)\#snd\ p)$
  from *a-A* **and** *p-path*
  have $has\text{-}fpath\ A\ ?p'\ m = end\text{-}node\ ?p'\ G = prod\ ?p'$
    by (*auto simp*:*end-node-def nth.simps intro*:*has-fpath.intros split*:*nat.split*)
  thus *?case* **using** *Suc* **by** *auto*
qed

3.5 Sub-Paths

definition *sub-path* :: $('n, 'e)\ ipath \Rightarrow nat \Rightarrow nat \Rightarrow ('n, 'e)\ fpath$
$((\text{-}\langle\text{-},\text{-}\rangle))$
where
  $p\langle i,j\rangle =$
  $(fst\ (p\ i),\ map\ (\lambda k.\ (snd\ (p\ k),\ fst\ (p\ (Suc\ k))))\ [i\ ..{<}\ j])$

lemma *sub-path-is-path*:

assumes *ipath*: *has-ipath G p*
assumes *l*: $i \le j$
shows *has-fpath G* ($p\langle i,j\rangle$)
using *l*
proof (*induct i rule*:*inc-induct*)
case *base* **show** *?case* **by** (*auto simp*:*sub-path-def intro*:*has-fpath.intros*)
next
case (*step i*)
with *ipath upt-rec*[*of i j*]
show *?case*
by (*auto simp*:*sub-path-def has-ipath-def intro*:*has-fpath.intros*)
qed

lemma *sub-path-start*[*simp*]:
fst ($p\langle i,j\rangle$) = *fst* (*p i*)
by (*simp add*:*sub-path-def*)

lemma *nth-upto*[*simp*]: $k < j - i \Longrightarrow [i ..< j]\ !\ k = i + k$
by (*induct k*) *auto*

lemma *sub-path-end*[*simp*]:
$i < j \Longrightarrow$ *end-node* ($p\langle i,j\rangle$) = *fst* (*p j*)
by (*auto simp*:*sub-path-def end-node-def*)

lemma *foldr-map*: *foldr f* (*map g xs*) = *foldr* (*f o g*) *xs*
by (*induct xs*) *auto*

lemma *upto-append*[*simp*]:
assumes $i \le j$ $j \le k$
shows $[\ i ..< j\ ]$ @ $[j ..< k] = [i ..< k]$
using *assms* **and** *upt-add-eq-append*[*of i j k* $- j$]
by *simp*

lemma *foldr-monoid*: *foldr* (*op* $*$) *xs 1* $*$ *foldr* (*op* $*$) *ys 1*
= *foldr* (*op* $*$) (*xs* @ *ys*) (*1*::'a::*monoid-mult*)
by (*induct xs*) (*auto simp*:*mult-assoc*)

lemma *sub-path-prod*:
assumes $i < j$
assumes $j < k$
shows *prod* ($p\langle i,k\rangle$) = *prod* ($p\langle i,j\rangle$) $*$ *prod* ($p\langle j,k\rangle$)
using *assms*
unfolding *prod-def sub-path-def*
by (*simp add*:*map-compose*[*symmetric*] *comp-def*)
(*simp only*:*foldr-monoid map-append*[*symmetric*] *upto-append*)

lemma *path-acgpow-aux*:

  assumes *length es* = *l*
  assumes *has-fpath G* (*n*,*es*)
  shows *has-edge* (*G* ^ *l*) *n* (*prod* (*n*,*es*)) (*end-node* (*n*,*es*))
using *assms*
proof (*induct l arbitrary*:*n es*)
  case *0* **thus** *?case*
    by (*simp add*:*in-grunit end-node-def*)
next
  case (*Suc l n es*)
  hence *es* ≠ [] **by** *auto*
  let *?n′* = *snd* (*hd es*)
  let *?es′* = *tl es*
  let *?e* = *fst* (*hd es*)

  from *Suc* **have** *len*: *length ?es′* = *l* **by** *auto*

  from *Suc*
  have [*simp*]: *end-node* (*n*, *es*) = *end-node* (*?n′*, *?es′*)
    by (*cases es*) (*auto simp*:*end-node-def nth.simps split*:*nat.split*)

  from ‹*has-fpath G* (*n*,*es*)›
  have *has-fpath G* (*?n′*, *?es′*)
    by (*rule has-fpath.cases*) (*auto intro*:*has-fpath.intros*)
  with *Suc len*
  have *has-edge* (*G* ^ *l*) *?n′* (*prod* (*?n′*, *?es′*)) (*end-node* (*?n′*, *?es′*))
    by *auto*
  moreover
  from ‹*es* ≠ []›
  have *prod* (*n*, *es*) = *?e* ∗ (*prod* (*?n′*, *?es′*))
    by (*cases es*) *auto*
  moreover
  from ‹*has-fpath G* (*n*,*es*)› **have** *c*:*has-edge G n ?e ?n′*
    by (*rule has-fpath.cases*) (*insert* ‹*es* ≠ []›, *auto*)

  ultimately
  show *?case*
     unfolding *power-Suc*
     by (*auto simp*:*in-grcomp*)
qed

lemma *path-acgpow*:
   has-fpath G p
  ⟹ *has-edge* (*G* ^ *length* (*snd p*)) (*fst p*) (*prod p*) (*end-node p*)
by (*cases p*)
   (*rule path-acgpow-aux*[*of snd p length* (*snd p*) - *fst p*, *simplified*])

lemma *star-paths*:

has-edge (*star G*) *a x b* =
(∃ *p*. *has-fpath G p* ∧ *a* = *fst p* ∧ *b* = *end-node p* ∧ *x* = *prod p*)
proof
assume *has-edge* (*star G*) *a x b*
then obtain *n* **where** *pow*: *has-edge* (*G* ^ *n*) *a x b*
by (*auto simp*:*in-star*)

then obtain *p* **where**
has-fpath G p a = *fst p b* = *end-node p x* = *prod p*
by (*rule power-induces-path*)

thus ∃ *p*. *has-fpath G p* ∧ *a* = *fst p* ∧ *b* = *end-node p* ∧ *x* = *prod p*
by *blast*
next
assume ∃ *p*. *has-fpath G p* ∧ *a* = *fst p* ∧ *b* = *end-node p* ∧ *x* = *prod p*
then obtain *p* **where**
has-fpath G p a = *fst p b* = *end-node p x* = *prod p*
by *blast*

hence *has-edge* (*G* ^ *length* (*snd p*)) *a x b*
by (*auto intro*:*path-acgpow*)

thus *has-edge* (*star G*) *a x b*
by (*auto simp*:*in-star*)
qed

lemma *plus-paths*:
has-edge (*tcl G*) *a x b* =
(∃ *p*. *has-fpath G p* ∧ *a* = *fst p* ∧ *b* = *end-node p* ∧ *x* = *prod p* ∧ *0* < *length* (*snd p*))
proof
assume *has-edge* (*tcl G*) *a x b*

then obtain *n* **where** *pow*: *has-edge* (*G* ^ *n*) *a x b* **and** *0* < *n*
by (*auto simp*:*in-tcl*)

from *pow* **obtain** *p* **where**
has-fpath G p a = *fst p b* = *end-node p x* = *prod p*
n = *length* (*snd p*)
by (*rule power-induces-path*)

with ‹*0* < *n*›
show ∃ *p*. *has-fpath G p* ∧ *a* = *fst p* ∧ *b* = *end-node p* ∧ *x* = *prod p* ∧ *0* < *length* (*snd p*)
by *blast*
next
assume ∃ *p*. *has-fpath G p* ∧ *a* = *fst p* ∧ *b* = *end-node p* ∧ *x* = *prod p*
∧ *0* < *length* (*snd p*)

then obtain p **where**
has-fpath G p a = fst p b = end-node p x = prod p
0 < length (snd p)
by *blast*

hence *has-edge (G ^ length (snd p)) a x b*
by (*auto intro*:*path-acgpow*)

with ‹*0 < length (snd p)*›
show *has-edge (tcl G) a x b*
by (*auto simp*:*in-tcl*)
qed

definition
contract s p =
$(\lambda i.\ (fst\ (p\ (s\ i)),\ prod\ (p\langle s\ i,s\ (Suc\ i)\rangle)))$

lemma *ipath-contract*:
assumes [*simp*]: *increasing s*
assumes *ipath*: *has-ipath G p*
shows *has-ipath (tcl G) (contract s p)*
unfolding *has-ipath-def*
proof
fix i
let $?p = p\langle s\ i,s\ (Suc\ i)\rangle$

from *increasing-strict*
have *fst (p (s (Suc i))) = end-node ?p* **by** *simp*
moreover
from *increasing-strict*[*of s i Suc i*] **have** $snd\ ?p \neq []$
by (*simp add*:*sub-path-def*)
moreover
from *ipath increasing-weak*[*of s*] **have** *has-fpath G ?p*
by (*rule sub-path-is-path*) *auto*
ultimately
show *has-edge (tcl G)*
(*fst (contract s p i)*) (*snd (contract s p i)*) (*fst (contract s p (Suc i))*)
unfolding *contract-def plus-paths*
by (*intro exI*) *auto*
qed

lemma *prod-unfold*:
$i < j \Longrightarrow prod\ (p\langle i,j\rangle)$
$= snd\ (p\ i) * prod\ (p\langle Suc\ i,\ j\rangle)$
unfolding *prod-def*
by (*simp add*:*sub-path-def upt-rec*[*of i j*])

```
lemma sub-path-loop:
  assumes 0 < k
  assumes k: k = length (snd loop)
  assumes loop: fst loop = end-node loop
  shows (omega loop)⟨k * i,k * Suc i⟩ = loop (is ?ω = loop)
proof (rule prod-eqI)
  show fst ?ω = fst loop
    by (auto simp:path-loop-def path-nth-def split-def k)

  show snd ?ω = snd loop
  proof (rule nth-equalityI[rule-format])
    show leneq: length (snd ?ω) = length (snd loop)
      unfolding sub-path-def k by simp

    fix j assume j < length (snd (?ω))
    with leneq and k have j < k by simp

    have a: ⋀i. fst (path-nth loop (Suc i mod k))
      = snd (snd (path-nth loop (i mod k)))
      unfolding k
      apply (rule path-loop-connect[OF loop])
      using ‹0 < k› and k
      apply auto
      done

    from ‹j < k›
    show snd ?ω ! j = snd loop ! j
      unfolding sub-path-def
      apply (simp add:path-loop-def split-def add-ac)
      apply (simp add:a k[symmetric])
      apply (simp add:path-nth-def)
      done
  qed
qed

end
```

4 The Size-Change Principle (Definition)

```
theory Criterion
imports Graphs Infinite-Set
begin
```

4.1 Size-Change Graphs

```
datatype sedge =
    LESS (↓)
  | LEQ (⇓)
```

instance *sedge* :: *one*
one-sedge-def: $1 \equiv \Downarrow$ **..**

instance *sedge* :: *times*
mult-sedge-def: $a * b \equiv$ *if* $a = \downarrow$ *then* $\downarrow$ *else* b **..**

instance *sedge* :: *comm-monoid-mult*
proof
fix $a\ b\ c$ :: *sedge*
show $a * b * c = a * (b * c)$ **by** (*simp add*:*mult-sedge-def*)
show $1 * a = a$ **by** (*simp add*:*mult-sedge-def one-sedge-def*)
show $a * b = b * a$ **unfolding** *mult-sedge-def*
by (*cases a*, *simp*) (*cases b*, *auto*)
qed

lemma *sedge-UNIV*:
UNIV = { *LESS*, *LEQ* }
proof (*intro equalityI subsetI*)
fix x **show** $x \in$ { *LESS*, *LEQ* }
by (*cases x*) *auto*
qed *auto*

instance *sedge* :: *finite*
proof
show *finite* (*UNIV*::*sedge set*)
by (*simp add*: *sedge-UNIV*)
qed

lemmas [*code func*] = *sedge-UNIV*

types *'a scg* = (*'a*, *sedge*) *graph*
types *'a acg* = (*'a*, *'a scg*) *graph*

4.2 Size-Change Termination

abbreviation (*input*)
desc P Q == $((\exists n.\forall i \geq n.\ P\ i) \wedge (\exists_\infty i.\ Q\ i))$

abbreviation (*input*)
dsc G i j $\equiv$ *has-edge G i LESS j*

abbreviation (*input*)
eq G i j $\equiv$ *has-edge G i LEQ j*

abbreviation
eql :: *'a scg* $\Rightarrow$ *'a* $\Rightarrow$ *'a* $\Rightarrow$ *bool*
(- $\vdash$ - $\rightsquigarrow$ -)

where
$\quad eql\ G\ i\ j \equiv has\text{-}edge\ G\ i\ LESS\ j \vee has\text{-}edge\ G\ i\ LEQ\ j$

abbreviation $(input)\ descat :: ('a,\ 'a\ scg)\ ipath \Rightarrow 'a\ sequence \Rightarrow nat \Rightarrow bool$
where
$\quad descat\ p\ \vartheta\ i \equiv has\text{-}edge\ (snd\ (p\ i))\ (\vartheta\ i)\ LESS\ (\vartheta\ (Suc\ i))$

abbreviation $(input)\ eqat :: ('a,\ 'a\ scg)\ ipath \Rightarrow 'a\ sequence \Rightarrow nat \Rightarrow bool$
where
$\quad eqat\ p\ \vartheta\ i \equiv has\text{-}edge\ (snd\ (p\ i))\ (\vartheta\ i)\ LEQ\ (\vartheta\ (Suc\ i))$

abbreviation $(input)\ eqlat :: ('a,\ 'a\ scg)\ ipath \Rightarrow 'a\ sequence \Rightarrow nat \Rightarrow bool$
where
$\quad eqlat\ p\ \vartheta\ i \equiv (has\text{-}edge\ (snd\ (p\ i))\ (\vartheta\ i)\ LESS\ (\vartheta\ (Suc\ i))$
$\qquad\qquad \vee\ has\text{-}edge\ (snd\ (p\ i))\ (\vartheta\ i)\ LEQ\ (\vartheta\ (Suc\ i)))$

definition $is\text{-}desc\text{-}thread :: 'a\ sequence \Rightarrow ('a,\ 'a\ scg)\ ipath \Rightarrow bool$
where
$\quad is\text{-}desc\text{-}thread\ \vartheta\ p = ((\exists\, n.\forall\, i \geq n.\ eqlat\ p\ \vartheta\ i) \wedge (\exists_\infty i.\ descat\ p\ \vartheta\ i))$

definition $SCT :: 'a\ acg \Rightarrow bool$
where
$\quad SCT\ \mathcal{A} =$
$\quad (\forall\, p.\ has\text{-}ipath\ \mathcal{A}\ p \longrightarrow (\exists\, \vartheta.\ is\text{-}desc\text{-}thread\ \vartheta\ p))$

definition $no\text{-}bad\text{-}graphs :: 'a\ acg \Rightarrow bool$
where
$\quad no\text{-}bad\text{-}graphs\ A =$
$\quad (\forall\, n\ G.\ has\text{-}edge\ A\ n\ G\ n \wedge G * G = G$
$\quad \longrightarrow (\exists\, p.\ has\text{-}edge\ G\ p\ LESS\ p))$

definition $SCT' :: 'a\ acg \Rightarrow bool$
where
$\quad SCT'\ A = no\text{-}bad\text{-}graphs\ (tcl\ A)$

end

5 Proof of the Size-Change Principle

theory *Correctness*
imports *Main Ramsey Misc-Tools Criterion*
begin

5.1 Auxiliary definitions

definition *is-thread* :: *nat* ⇒ *'a sequence* ⇒ (*'a*, *'a scg*) *ipath* ⇒ *bool*
where
is-thread n ϑ p = (∀ *i*≥*n*. *eqlat p ϑ i*)

definition *is-fthread* ::
'a sequence ⇒ (*'a*, *'a scg*) *ipath* ⇒ *nat* ⇒ *nat* ⇒ *bool*
where
is-fthread ϑ mp i j = (∀ *k*∈{*i*..<*j*}. *eqlat mp ϑ k*)

definition *is-desc-fthread* ::
'a sequence ⇒ (*'a*, *'a scg*) *ipath* ⇒ *nat* ⇒ *nat* ⇒ *bool*
where
is-desc-fthread ϑ mp i j =
(*is-fthread ϑ mp i j* ∧
(∃ *k*∈{*i*..<*j*}. *descat mp ϑ k*))

definition
has-fth p i j n m =
(∃ *ϑ*. *is-fthread ϑ p i j* ∧ *ϑ i* = *n* ∧ *ϑ j* = *m*)

definition
has-desc-fth p i j n m =
(∃ *ϑ*. *is-desc-fthread ϑ p i j* ∧ *ϑ i* = *n* ∧ *ϑ j* = *m*)

5.2 Everything is finite

lemma *finite-range*:
fixes *f* :: *nat* ⇒ *'a*
assumes *fin*: *finite* (*range f*)
shows $\exists x.\ \exists_\infty i.\ f\ i = x$
proof (*rule classical*)
assume $\neg(\exists x.\ \exists_\infty i.\ f\ i = x)$
hence $\forall x.\ \exists j.\ \forall i{>}j.\ f\ i \neq x$
unfolding *INF-nat* **by** *blast*
with *choice*
have $\exists j.\ \forall x.\ \forall i{>}(j\ x).\ f\ i \neq x$ **.**
then obtain *j* **where**
neq: $\bigwedge x\ i.\ j\ x < i \Longrightarrow f\ i \neq x$ **by** *blast*

from *fin* **have** *finite* (*range* (*j o f*))
by (*auto simp*:*comp-def*)
with *finite-nat-bounded*
obtain *m* **where** *range* (*j o f*) ⊆ {..<*m*} **by** *blast*
hence *j* (*f m*) < *m* **unfolding** *comp-def* **by** *auto*

with *neq*[*of f m m*] **show** *?thesis* **by** *blast*
qed

lemma *finite-range-ignore-prefix*:
  fixes $f :: nat \Rightarrow 'a$
  assumes *fA*: *finite A*
  assumes *inA*: $\forall x \geq n.\ f\ x \in A$
  shows *finite* (*range f*)
proof −
  have *a*: $UNIV = \{0\ ..<\ (n{::}nat)\} \cup \{\ x.\ n \leq x\ \}$ **by** *auto*
  have *b*: $range\ f = f\ `\ \{0\ ..<\ n\} \cup f\ `\ \{\ x.\ n \leq x\ \}$
    (**is** $\ldots = ?A \cup ?B$)
    by (*unfold a*) (*simp add*:*image-Un*)

  have *finite ?A* **by** (*rule finite-imageI*) *simp*
  moreover
  from *inA* **have** $?B \subseteq A$ **by** *auto*
  from *this fA* **have** *finite ?B* **by** (*rule finite-subset*)
  ultimately show *?thesis* **using** *b* **by** *simp*
qed

definition
  finite-graph G = *finite* (*dest-graph* G)
definition
  all-finite $G = (\forall\, n\ H\ m.\ has\text{-}edge\ G\ n\ H\ m \longrightarrow finite\text{-}graph\ H)$
definition
  finite-acg $A = (finite\text{-}graph\ A \wedge all\text{-}finite\ A)$
definition
  nodes $G = fst\ `\ dest\text{-}graph\ G \cup snd\ `\ snd\ `\ dest\text{-}graph\ G$
definition
  edges $G = fst\ `\ snd\ `\ dest\text{-}graph\ G$
definition
  smallnodes $G = \bigcup (nodes\ `\ edges\ G)$

lemma *thread-image-nodes*:
  assumes *th*: *is-thread* $n\ \vartheta\ p$
  shows $\forall\, i \geq n.\ \vartheta\ i \in nodes\ (snd\ (p\ i))$
using *prems*
unfolding *is-thread-def has-edge-def nodes-def*
by *force*

lemma *finite-nodes*: $finite\text{-}graph\ G \Longrightarrow finite\ (nodes\ G)$
  unfolding *finite-graph-def nodes-def*
  by *auto*

lemma *nodes-subgraph*: $A \leq B \Longrightarrow nodes\ A \subseteq nodes\ B$
  unfolding *graph-leq-def nodes-def*
  by *auto*

lemma *finite-edges*: *finite-graph* $G \Longrightarrow$ *finite* (*edges* G)
  unfolding *finite-graph-def edges-def*
  by *auto*

lemma *edges-sum*[*simp*]: *edges* $(A + B) =$ *edges* $A \cup$ *edges* B
  unfolding *edges-def graph-plus-def*
  by *auto*

lemma *nodes-sum*[*simp*]: *nodes* $(A + B) =$ *nodes* $A \cup$ *nodes* B
  unfolding *nodes-def graph-plus-def*
  by *auto*

lemma *finite-acg-subset*:
  $A \leq B \Longrightarrow$ *finite-acg* $B \Longrightarrow$ *finite-acg* A
  unfolding *finite-acg-def finite-graph-def all-finite-def*
  has-edge-def graph-leq-def
  by (*auto elim*:*finite-subset*)

lemma *scg-finite*:
  fixes G :: *'a scg*
  assumes *fin*: *finite* (*nodes* G)
  shows *finite-graph* G
  unfolding *finite-graph-def*
proof (*rule finite-subset*)
  show *dest-graph* $G \subseteq$ *nodes* $G \times$ *UNIV* $\times$ *nodes* G (**is** - $\subseteq$ *?P*)
    unfolding *nodes-def*
    by *force*
  show *finite ?P*
    by (*intro finite-cartesian-product fin finite*)
qed

lemma *smallnodes-sum*[*simp*]:
  smallnodes $(A + B) =$ *smallnodes* $A \cup$ *smallnodes* B
  unfolding *smallnodes-def*
  by *auto*

lemma *in-smallnodes*:
  fixes A :: *'a acg*
  assumes *e*: *has-edge* $A\ x\ G\ y$
  shows *nodes* $G \subseteq$ *smallnodes* A
proof −
  have *fst* (*snd* (x, G, y)) $\in$ *fst* ‘ *snd* ‘ *dest-graph* A
    unfolding *has-edge-def*
    by (*rule imageI*)+ (*rule e*[*unfolded has-edge-def*])
  then have $G \in$ *edges* A
    unfolding *edges-def* **by** *simp*
  thus *?thesis*
    unfolding *smallnodes-def*
    by *blast*

```
qed

lemma finite-smallnodes:
  assumes fA: finite-acg A
  shows finite (smallnodes A)
  unfolding smallnodes-def edges-def
proof
  from fA
  show finite (nodes ` fst ` snd ` dest-graph A)
    unfolding finite-acg-def finite-graph-def
    by simp

  fix M assume M ∈ nodes ` fst ` snd ` dest-graph A
  then obtain n G m
    where M: M = nodes G and nGm: (n,G,m) ∈ dest-graph A
    by auto

  from fA
  have all-finite A unfolding finite-acg-def by simp
  with nGm have finite-graph G
    unfolding all-finite-def has-edge-def by auto
  with finite-nodes
  show finite M
    unfolding finite-graph-def M .
qed

lemma nodes-tcl:
  nodes (tcl A) = nodes A
proof
  show nodes A ⊆ nodes (tcl A)
    apply (rule nodes-subgraph)
    by (subst tcl-unfold-right) simp

  show nodes (tcl A) ⊆ nodes A
  proof
    fix x assume x ∈ nodes (tcl A)
    then obtain z G y
      where z: z ∈ dest-graph (tcl A)
      and dis: z = (x, G, y) ∨ z = (y, G, x)
      unfolding nodes-def
      by auto force+

    from dis
    show x ∈ nodes A
    proof
      assume z = (x, G, y)
      with z have has-edge (tcl A) x G y unfolding has-edge-def by simp
      then obtain n where n > 0  and An: has-edge (A ^ n) x G y
        unfolding in-tcl by auto
```

then obtain n' **where** $n = Suc\ n'$ **by** (*cases n*, *auto*)
hence $A\ \hat{}\ n = A * A\ \hat{}\ n'$ **by** (*simp add*:*power-Suc*)
with *An* **obtain** *e k*
where *has-edge A x e k* **by** (*auto simp*:*in-grcomp*)
thus $x \in$ *nodes A* **unfolding** *has-edge-def nodes-def*
by *force*
next
assume $z = (y, G, x)$
with *z* **have** *has-edge* (*tcl A*) *y G x* **unfolding** *has-edge-def* **by** *simp*
then obtain *n* **where** $n > 0$ **and** *An*: *has-edge* $(A\ \hat{}\ n)$ *y G x*
unfolding *in-tcl* **by** *auto*
then obtain n' **where** $n = Suc\ n'$ **by** (*cases n*, *auto*)
hence $A\ \hat{}\ n = A\ \hat{}\ n' * A$ **by** (*simp add*:*power-Suc power-commutes*)
with *An* **obtain** *e k*
where *has-edge A k e x* **by** (*auto simp*:*in-grcomp*)
thus $x \in$ *nodes A* **unfolding** *has-edge-def nodes-def*
by *force*
qed
qed
qed

lemma *smallnodes-tcl*:
fixes *A* :: *'a acg*
shows *smallnodes* (*tcl A*) = *smallnodes A*
proof (*intro equalityI subsetI*)
fix *n* **assume** $n \in$ *smallnodes* (*tcl A*)
then obtain *x G y* **where** *edge*: *has-edge* (*tcl A*) *x G y*
and $n \in$ *nodes G*
unfolding *smallnodes-def edges-def has-edge-def*
by *auto*

from ‹$n \in$ *nodes G*›
have $n \in$ *fst* ‘ *dest-graph G* $\vee$ $n \in$ *snd* ‘ *snd* ‘ *dest-graph G*
(**is** *?A* $\vee$ *?B*)
unfolding *nodes-def* **by** *blast*
thus $n \in$ *smallnodes A*
proof
assume *?A*
then obtain *m e* **where** *A*: *has-edge G n e m*
unfolding *has-edge-def* **by** *auto*

have *tcl A* = $A *$ *star A*
unfolding *tcl-def*
by (*simp add*: *star-commute*[*of A A A*, *simplified*])

with *edge*
have *has-edge* ($A *$ *star A*) *x G y* **by** *simp*
then obtain *H H' z*
where *AH*: *has-edge A x H z* **and** *G*: $G = H * H'$

```
      by (auto simp:in-grcomp)
    from A
    obtain m′ e′ where has-edge H n e′ m′
      by (auto simp:G in-grcomp)
    hence n ∈ nodes H unfolding nodes-def has-edge-def
      by force
    with in-smallnodes[OF AH] show n ∈ smallnodes A ..
  next
    assume ?B
    then obtain m e where B: has-edge G m e n
      unfolding has-edge-def by auto

    with edge
    have has-edge (star A ∗ A) x G y by (simp add:tcl-def)
    then obtain H H′ z
      where AH′: has-edge A z H′ y and G: G = H ∗ H′
      by (auto simp:in-grcomp)
    from B
    obtain m′ e′ where has-edge H′ m′ e′ n
      by (auto simp:G in-grcomp)
    hence n ∈ nodes H′ unfolding nodes-def has-edge-def
      by force
    with in-smallnodes[OF AH′] show n ∈ smallnodes A ..
  qed
next
  fix x assume x ∈ smallnodes A
  then show x ∈ smallnodes (tcl A)
    by (subst tcl-unfold-right) simp
qed

lemma finite-nodegraphs:
  assumes F: finite F
  shows finite { G::′a scg. nodes G ⊆ F } (is finite ?P)
proof (rule finite-subset)
  show ?P ⊆ Graph ‘ (Pow (F × UNIV × F)) (is ?P ⊆ ?Q)
  proof
    fix x assume xP: x ∈ ?P
    obtain S where x[simp]: x = Graph S
      by (cases x) auto
    from xP
    show x ∈ ?Q
      apply (simp add:nodes-def)
      apply (rule imageI)
      apply (rule PowI)
      apply force
      done
  qed
  show finite ?Q
    by (auto intro:finite-imageI finite-cartesian-product F finite)
```

qed

lemma *finite-graphI*:
  fixes A :: $'a$ *acg*
  assumes *fin*: *finite* (*nodes* A) *finite* (*smallnodes* A)
  shows *finite-graph* A
proof −
  obtain S **where** A[*simp*]: A = *Graph* S
    by (*cases* A) *auto*

  have *finite* S
  proof (*rule finite-subset*)
    show $S \subseteq$ *nodes* $A \times$ { G::$'a$ *scg*. *nodes* $G \subseteq$ *smallnodes* A } $\times$ *nodes* A
      (**is** $S \subseteq$ *?T*)
    proof
      fix x **assume** xS: $x \in S$
      obtain a b c **where** x[*simp*]: $x = (a, b, c)$
        by (*cases* x) *auto*

      then have *edg*: *has-edge* A a b c
        unfolding *has-edge-def* **using** xS
        by *simp*

      hence $a \in$ *nodes* A $c \in$ *nodes* A
        unfolding *nodes-def has-edge-def* **by** *force+*
      moreover
      from *edg* **have** *nodes* $b \subseteq$ *smallnodes* A **by** (*rule in-smallnodes*)
      hence $b \in$ { G :: $'a$ *scg*. *nodes* $G \subseteq$ *smallnodes* A } **by** *simp*
      ultimately show $x \in$ *?T* **by** *simp*
    qed

    show *finite ?T*
      by (*intro finite-cartesian-product fin finite-nodegraphs*)
  qed
  thus *?thesis*
    unfolding *finite-graph-def* **by** *simp*
qed

lemma *smallnodes-allfinite*:
  fixes A :: $'a$ *acg*
  assumes *fin*: *finite* (*smallnodes* A)
  shows *all-finite* A
  unfolding *all-finite-def*
proof (*intro allI impI*)
  fix n H m **assume** *has-edge* A n H m
  then have *nodes* $H \subseteq$ *smallnodes* A
    by (*rule in-smallnodes*)
  then have *finite* (*nodes* H)

```
    by (rule finite-subset) (rule fin)
  thus finite-graph H by (rule scg-finite)
qed

lemma finite-tcl:
  fixes A :: 'a acg
  shows finite-acg (tcl A) ⟷ finite-acg A
proof
  assume f: finite-acg A
  from f have g: finite-graph A and all-finite A
    unfolding finite-acg-def by auto

  from g have finite (nodes A) by (rule finite-nodes)
  then have finite (nodes (tcl A)) unfolding nodes-tcl .
  moreover
  from f have finite (smallnodes A) by (rule finite-smallnodes)
  then have fs: finite (smallnodes (tcl A)) unfolding smallnodes-tcl .
  ultimately
  have finite-graph (tcl A) by (rule finite-graphI)

  moreover from fs have all-finite (tcl A)
    by (rule smallnodes-allfinite)
  ultimately show finite-acg (tcl A) unfolding finite-acg-def ..
next
  assume a: finite-acg (tcl A)
  have A ≤ tcl A by (rule less-tcl)
  thus finite-acg A using a
    by (rule finite-acg-subset)
qed

lemma finite-acg-empty: finite-acg (Graph {})
  unfolding finite-acg-def finite-graph-def all-finite-def
  has-edge-def
  by simp

lemma finite-acg-ins:
  assumes fA: finite-acg (Graph A)
  assumes fG: finite G
  shows finite-acg (Graph (insert (a, Graph G, b) A))
  using fA fG
  unfolding finite-acg-def finite-graph-def all-finite-def
  has-edge-def
  by auto

lemmas finite-acg-simps = finite-acg-empty finite-acg-ins finite-graph-def
```

5.3 Contraction and more

abbreviation

pdesc P == (*fst P*, *prod P*, *end-node P*)

lemma *pdesc-acgplus*:
  assumes *has-ipath* $\mathcal{A}$ *p*
  and $i < j$
  shows *has-edge* (*tcl* $\mathcal{A}$) (*fst* ($p\langle i,j\rangle$)) (*prod* ($p\langle i,j\rangle$)) (*end-node* ($p\langle i,j\rangle$))
  unfolding *plus-paths*
  apply (*rule exI*)
  apply (*insert prems*)
  by (*auto intro*:*sub-path-is-path*[*of* $\mathcal{A}$ *p i j*] *simp*:*sub-path-def*)

lemma *combine-fthreads*:
  assumes *range*: $i < j$ $j \le k$
  shows
  has-fth p i k m r =
  ($\exists$ *n*. *has-fth p i j m n* $\wedge$ *has-fth p j k n r*) (**is** *?L* = *?R*)
proof (*intro iffI*)
  assume *?L*
  then obtain ϑ
    where *is-fthread* ϑ *p i k*
    and [*simp*]: ϑ $i = m$ ϑ $k = r$
    by (*auto simp*:*has-fth-def*)

  with *range*
  have *is-fthread* ϑ *p i j* **and** *is-fthread* ϑ *p j k*
    by (*auto simp*:*is-fthread-def*)
  hence *has-fth p i j m* (ϑ *j*) **and** *has-fth p j k* (ϑ *j*) *r*
    by (*auto simp*:*has-fth-def*)
  thus *?R* **by** *auto*
next
  assume *?R*
  then obtain *n* $\vartheta 1$ $\vartheta 2$
    where *ths*: *is-fthread* $\vartheta 1$ *p i j* *is-fthread* $\vartheta 2$ *p j k*
    and [*simp*]: $\vartheta 1$ $i = m$ $\vartheta 1$ $j = n$ $\vartheta 2$ $j = n$ $\vartheta 2$ $k = r$
    by (*auto simp*:*has-fth-def*)

  let $?\vartheta = (\lambda i.$ *if* $i < j$ *then* $\vartheta 1$ *i else* $\vartheta 2$ *i*$)$
  have *is-fthread* $?\vartheta$ *p i k*
    unfolding *is-fthread-def*
  proof
    fix *l* **assume** *range*: $l \in \{i..<k\}$

    show *eqlat p* $?\vartheta$ *l*
    proof (*cases rule*:*three-cases*)
      assume *Suc* $l < j$
      with *ths range* **show** *?thesis*
        unfolding *is-fthread-def Ball-def*
        by *simp*

```
    next
      assume Suc l = j
      hence l < j ϑ2 (Suc l) = ϑ1 (Suc l) by auto
      with ths range show ?thesis
        unfolding is-fthread-def Ball-def
        by simp
    next
      assume j ≤ l
      with ths range show ?thesis
        unfolding is-fthread-def Ball-def
        by simp
    qed arith
  qed
  moreover
  have ?ϑ i = m ?ϑ k = r using range by auto
  ultimately show has-fth p i k m r
    by (auto simp:has-fth-def)
qed


lemma desc-is-fthread:
  is-desc-fthread ϑ p i k ⟹ is-fthread ϑ p i k
  unfolding is-desc-fthread-def
  by simp


lemma combine-dfthreads:
  assumes range: i < j j ≤ k
  shows
  has-desc-fth p i k m r =
  (∃ n. (has-desc-fth p i j m n ∧ has-fth p j k n r)
  ∨ (has-fth p i j m n ∧ has-desc-fth p j k n r)) (is ?L = ?R)
proof
  assume ?L
  then obtain ϑ
    where desc: is-desc-fthread ϑ p i k
    and [simp]: ϑ i = m ϑ k = r
    by (auto simp:has-desc-fth-def)

  hence is-fthread ϑ p i k
    by (simp add: desc-is-fthread)
  with range have fths: is-fthread ϑ p i j is-fthread ϑ p j k
    unfolding is-fthread-def
    by auto
  hence hfths: has-fth p i j m (ϑ j) has-fth p j k (ϑ j) r
    by (auto simp:has-fth-def)

  from desc obtain l
    where i ≤ l l < k
```

and *descat p* ϑ *l*
by (*auto simp*:*is-desc-fthread-def*)

with *fths*
have *is-desc-fthread* ϑ *p i j* $\vee$ *is-desc-fthread* ϑ *p j k*
unfolding *is-desc-fthread-def*
by (*cases* $l < j$) *auto*
hence *has-desc-fth p i j m* (ϑ *j*) $\vee$ *has-desc-fth p j k* (ϑ *j*) *r*
by (*auto simp*:*has-desc-fth-def*)
with *hfths* **show** *?R*
by *auto*
next
assume *?R*
then obtain *n* $\vartheta 1$ $\vartheta 2$
where (*is-desc-fthread* $\vartheta 1$ *p i j* $\wedge$ *is-fthread* $\vartheta 2$ *p j k*)
$\vee$ (*is-fthread* $\vartheta 1$ *p i j* $\wedge$ *is-desc-fthread* $\vartheta 2$ *p j k*)
and [*simp*]: $\vartheta 1\ i = m$ $\vartheta 1\ j = n$ $\vartheta 2\ j = n$ $\vartheta 2\ k = r$
by (*auto simp*:*has-fth-def has-desc-fth-def*)

hence *ths2*: *is-fthread* $\vartheta 1$ *p i j* *is-fthread* $\vartheta 2$ *p j k*
and *dths*: *is-desc-fthread* $\vartheta 1$ *p i j* $\vee$ *is-desc-fthread* $\vartheta 2$ *p j k*
by (*auto simp*:*desc-is-fthread*)

let $?\vartheta = (\lambda i.\ \mathit{if}\ i < j\ \mathit{then}\ \vartheta 1\ i\ \mathit{else}\ \vartheta 2\ i)$
have *is-fthread* $?\vartheta$ *p i k*
unfolding *is-fthread-def*
proof
fix *l* **assume** *range*: $l \in \{i..<k\}$

show *eqlat p* $?\vartheta$ *l*
proof (*cases rule*:*three-cases*)
assume *Suc* $l < j$
with *ths2 range* **show** *?thesis*
unfolding *is-fthread-def Ball-def*
by *simp*
next
assume *Suc* $l = j$
hence $l < j$ $\vartheta 2$ (*Suc l*) $= \vartheta 1$ (*Suc l*) **by** *auto*
with *ths2 range* **show** *?thesis*
unfolding *is-fthread-def Ball-def*
by *simp*
next
assume $j \le l$
with *ths2 range* **show** *?thesis*
unfolding *is-fthread-def Ball-def*
by *simp*
qed *arith*
qed
moreover

from *dths*
have $\exists l.\ i \leq l \wedge l < k \wedge descat\ p\ ?\vartheta\ l$
proof
assume *is-desc-fthread* $\vartheta 1\ p\ i\ j$

then obtain l **where** *range*: $i \leq l$ $l < j$ **and** *descat* $p\ \vartheta 1\ l$
unfolding *is-desc-fthread-def Bex-def* **by** *auto*
hence *descat* $p\ ?\vartheta\ l$
by (*cases Suc* $l = j$, *auto*)
with ‹$j \leq k$› **and** *range* **show** *?thesis*
by (*rule-tac* $x{=}l$ **in** *exI*, *auto*)
next
assume *is-desc-fthread* $\vartheta 2\ p\ j\ k$
then obtain l **where** *range*: $j \leq l$ $l < k$ **and** *descat* $p\ \vartheta 2\ l$
unfolding *is-desc-fthread-def Bex-def* **by** *auto*
with ‹$i < j$› **have** *descat* $p\ ?\vartheta\ l$ $i \leq l$
by *auto*
with *range* **show** *?thesis*
by (*rule-tac* $x{=}l$ **in** *exI*, *auto*)
qed
ultimately have *is-desc-fthread* $?\vartheta\ p\ i\ k$
by (*simp add*: *is-desc-fthread-def Bex-def*)

moreover
have $?\vartheta\ i = m$ $?\vartheta\ k = r$ **using** *range* **by** *auto*

ultimately show *has-desc-fth* $p\ i\ k\ m\ r$
by (*auto simp*:*has-desc-fth-def*)
qed

lemma *fth-single*:
has-fth $p\ i\ (Suc\ i)\ m\ n = eql\ (snd\ (p\ i))\ m\ n$ (**is** $?L = ?R$)
proof
assume $?L$ **thus** $?R$
unfolding *is-fthread-def Ball-def has-fth-def*
by *auto*
next
let $?\vartheta = \lambda k.\ if\ k = i\ then\ m\ else\ n$
assume *edge*: $?R$
hence *is-fthread* $?\vartheta\ p\ i\ (Suc\ i) \wedge ?\vartheta\ i = m \wedge ?\vartheta\ (Suc\ i) = n$
unfolding *is-fthread-def Ball-def*
by *auto*

thus $?L$
unfolding *has-fth-def*
by *auto*
qed

lemma *desc-fth-single*:
  has-desc-fth p i (*Suc i*) *m n* =
  dsc (*snd* (*p i*)) *m n* (**is** *?L* = *?R*)
proof
  assume *?L* **thus** *?R*
    unfolding *is-desc-fthread-def has-desc-fth-def is-fthread-def*
    Bex-def
    by (*elim exE conjE*) (*case-tac k* = *i*, *auto*)
next
  let $?\vartheta = \lambda k.$ *if* $k = i$ *then* m *else* n
  assume *edge*: *?R*
  hence *is-desc-fthread* $?\vartheta$ *p i* (*Suc i*) $\wedge$ $?\vartheta$ $i = m$ $\wedge$ $?\vartheta$ (*Suc i*) $= n$
    unfolding *is-desc-fthread-def is-fthread-def Ball-def Bex-def*
    by *auto*
  thus *?L*
    unfolding *has-desc-fth-def*
    by *auto*
qed

lemma *mk-eql*: ($G \vdash m \leadsto^{e} n$) $\Longrightarrow$ *eql G m n*
  by (*cases e*, *auto*)

lemma *eql-scgcomp*:
  eql ($G * H$) *m r* =
  ($\exists\, n.$ *eql G m n* $\wedge$ *eql H n r*) (**is** *?L* = *?R*)
proof
  show *?L* $\Longrightarrow$ *?R*
    by (*auto simp*:*in-grcomp intro*!:*mk-eql*)

  assume *?R*
  then obtain *n* **where** *l*: *eql G m n* **and** *r*:*eql H n r* **by** *auto*
  thus *?L*
    by (*cases dsc G m n*) (*auto simp*:*in-grcomp mult-sedge-def*)
qed

lemma *desc-scgcomp*:
  dsc ($G * H$) *m r* =
  ($\exists\, n.$ (*dsc G m n* $\wedge$ *eql H n r*) $\vee$ (*eq G m n* $\wedge$ *dsc H n r*)) (**is** *?L* = *?R*)
proof
  show *?R* $\Longrightarrow$ *?L* **by** (*auto simp*:*in-grcomp mult-sedge-def*)

  assume *?L*
  thus *?R*
    by (*auto simp*:*in-grcomp mult-sedge-def*)
  (*case-tac e*, *auto*, *case-tac e'*, *auto*)
qed

lemma *has-fth-unfold*:
assumes $i < j$
shows *has-fth* $p\ i\ j\ m\ n =$
$(\exists\, r.\ \textit{has-fth}\ p\ i\ (\textit{Suc}\ i)\ m\ r \land \textit{has-fth}\ p\ (\textit{Suc}\ i)\ j\ r\ n)$
by (*rule combine-fthreads*) (*insert* ‹$i < j$›, *auto*)

lemma *has-dfth-unfold*:
assumes *range*: $i < j$
shows
has-desc-fth $p\ i\ j\ m\ r =$
$(\exists\, n.\ (\textit{has-desc-fth}\ p\ i\ (\textit{Suc}\ i)\ m\ n \land \textit{has-fth}\ p\ (\textit{Suc}\ i)\ j\ n\ r)$
$\lor\ (\textit{has-fth}\ p\ i\ (\textit{Suc}\ i)\ m\ n \land \textit{has-desc-fth}\ p\ (\textit{Suc}\ i)\ j\ n\ r))$
by (*rule combine-dfthreads*) (*insert* ‹$i < j$›, *auto*)

lemma *Lemma7a*:
$i \le j \Longrightarrow \textit{has-fth}\ p\ i\ j\ m\ n = \textit{eql}\ (\textit{prod}\ (p\langle i,j\rangle))\ m\ n$
proof (*induct i arbitrary*: *m rule*:*inc-induct*)
case *base* **show** *?case*
unfolding *has-fth-def is-fthread-def sub-path-def*
by (*auto simp*:*in-grunit one-sedge-def*)
next
case (*step i*)
note $IH =$ ‹$\bigwedge m.\ \textit{has-fth}\ p\ (\textit{Suc}\ i)\ j\ m\ n =$
$\textit{eql}\ (\textit{prod}\ (p\langle \textit{Suc}\ i,j\rangle))\ m\ n$›

have *has-fth* $p\ i\ j\ m\ n$
$= (\exists\, r.\ \textit{has-fth}\ p\ i\ (\textit{Suc}\ i)\ m\ r \land \textit{has-fth}\ p\ (\textit{Suc}\ i)\ j\ r\ n)$
by (*rule has-fth-unfold*[*OF* ‹$i < j$›])
also have $\ldots = (\exists\, r.\ \textit{has-fth}\ p\ i\ (\textit{Suc}\ i)\ m\ r$
$\land\ \textit{eql}\ (\textit{prod}\ (p\langle \textit{Suc}\ i,j\rangle))\ r\ n)$
by (*simp only*:*IH*)
also have $\ldots = (\exists\, r.\ \textit{eql}\ (\textit{snd}\ (p\ i))\ m\ r$
$\land\ \textit{eql}\ (\textit{prod}\ (p\langle \textit{Suc}\ i,j\rangle))\ r\ n)$
by (*simp only*:*fth-single*)
also have $\ldots = \textit{eql}\ (\textit{snd}\ (p\ i) * \textit{prod}\ (p\langle \textit{Suc}\ i,j\rangle))\ m\ n$
by (*simp only*:*eql-scgcomp*)
also have $\ldots = \textit{eql}\ (\textit{prod}\ (p\langle i,j\rangle))\ m\ n$
by (*simp only*:*prod-unfold*[*OF* ‹$i < j$›])
finally show *?case* **.**
qed

lemma *Lemma7b*:
assumes $i \le j$
shows
has-desc-fth $p\ i\ j\ m\ n =$
$\textit{dsc}\ (\textit{prod}\ (p\langle i,j\rangle))\ m\ n$
using *prems*

proof (*induct i arbitrary*: *m rule*:*inc-induct*)
  case *base* **show** *?case*
    unfolding *has-desc-fth-def is-desc-fthread-def sub-path-def*
    by (*auto simp*:*in-grunit one-sedge-def*)
next
  case (*step i*)
  thus *?case*
    by (*simp only*:*prod-unfold desc-scgcomp desc-fth-single*
    has-dfth-unfold fth-single Lemma7a) *auto*
qed

lemma *descat-contract*:
  assumes [*simp*]: *increasing s*
  shows
  descat (*contract s p*) ϑ *i* =
  has-desc-fth p (*s i*) (*s* (*Suc i*)) (ϑ *i*) (ϑ (*Suc i*))
  by (*simp add*:*Lemma7b increasing-weak contract-def*)

lemma *eqlat-contract*:
  assumes [*simp*]: *increasing s*
  shows
  eqlat (*contract s p*) ϑ *i* =
  has-fth p (*s i*) (*s* (*Suc i*)) (ϑ *i*) (ϑ (*Suc i*))
  by (*auto simp*:*Lemma7a increasing-weak contract-def*)

5.3.1 Connecting threads

definition
  connect s ϑs = (λk. ϑs (*section-of s k*) *k*)

lemma *next-in-range*:
  assumes [*simp*]: *increasing s*
  assumes *a*: $k \in$ *section s i*
  shows (*Suc k* $\in$ *section s i*) $\vee$ (*Suc k* $\in$ *section s* (*Suc i*))
proof −
  from *a* **have** $k <$ *s* (*Suc i*) **by** *simp*

  hence *Suc k* $<$ *s* (*Suc i*) $\vee$ *Suc k* $=$ *s* (*Suc i*) **by** *arith*
  thus *?thesis*
  proof
    assume *Suc k* $<$ *s* (*Suc i*)
    with *a* **have** *Suc k* $\in$ *section s i* **by** *simp*
    thus *?thesis* **..**
  next
    assume *eq*: *Suc k* $=$ *s* (*Suc i*)
    with *increasing-strict* **have** *Suc k* $<$ *s* (*Suc* (*Suc i*)) **by** *simp*
    with *eq* **have** *Suc k* $\in$ *section s* (*Suc i*) **by** *simp*

```
    thus ?thesis ..
  qed
qed


lemma connect-threads:
  assumes [simp]: increasing s
  assumes connected: ϑs i (s (Suc i)) = ϑs (Suc i) (s (Suc i))
  assumes fth: is-fthread (ϑs i) p (s i) (s (Suc i))

  shows
  is-fthread (connect s ϑs) p (s i) (s (Suc i))
  unfolding is-fthread-def
proof
  fix k assume krng: k ∈ section s i

  with fth have eqlat: eqlat p (ϑs i) k
    unfolding is-fthread-def by simp

  from krng and next-in-range
  have (Suc k ∈ section s i) ∨ (Suc k ∈ section s (Suc i))
    by simp
  thus eqlat p (connect s ϑs) k
  proof
    assume Suc k ∈ section s i
    with krng eqlat show ?thesis
      unfolding connect-def
      by (simp only:section-of-known ‹increasing s›)
  next
    assume skrng: Suc k ∈ section s (Suc i)
    with krng have Suc k = s (Suc i) by auto

    with krng skrng eqlat show ?thesis
      unfolding connect-def
      by (simp only:section-of-known connected[symmetric] ‹increasing s›)
  qed
qed


lemma connect-dthreads:
  assumes inc[simp]: increasing s
  assumes connected: ϑs i (s (Suc i)) = ϑs (Suc i) (s (Suc i))
  assumes fth: is-desc-fthread (ϑs i) p (s i) (s (Suc i))

  shows
  is-desc-fthread (connect s ϑs) p (s i) (s (Suc i))
  unfolding is-desc-fthread-def
proof
  show is-fthread (connect s ϑs) p (s i) (s (Suc i))
```

apply (*rule connect-threads*)
apply (*insert fth*)
by (*auto simp*:*connected is-desc-fthread-def*)

from *fth*
obtain k **where** *dsc*: *descat* p $(\vartheta s\ i)$ k **and** *krng*: $k \in$ *section* $s\ i$
unfolding *is-desc-fthread-def* **by** *blast*

from *krng* **and** *next-in-range*
have $(Suc\ k \in section\ s\ i) \vee (Suc\ k \in section\ s\ (Suc\ i))$
by *simp*
hence *descat* p $(connect\ s\ \vartheta s)$ k
proof
assume $Suc\ k \in section\ s\ i$
with *krng dsc* **show** *?thesis* **unfolding** *connect-def*
by (*simp only*:*section-of-known inc*)
next
assume *skrng*: $Suc\ k \in section\ s\ (Suc\ i)$
with *krng* **have** $Suc\ k = s\ (Suc\ i)$ **by** *auto*

with *krng skrng dsc* **show** *?thesis* **unfolding** *connect-def*
by (*simp only*:*section-of-known connected*[*symmetric*] *inc*)
qed
with *krng* **show** $\exists k \in section\ s\ i.\ descat\ p\ (connect\ s\ \vartheta s)\ k$ **..**
qed

lemma *mk-inf-thread*:
assumes [*simp*]: *increasing* s
assumes *fths*: $\bigwedge i.\ i > n \Longrightarrow$ *is-fthread* $\vartheta\ p\ (s\ i)\ (s\ (Suc\ i))$
shows *is-thread* $(s\ (Suc\ n))\ \vartheta\ p$
unfolding *is-thread-def*
proof (*intro allI impI*)
fix j **assume** *st*: $s\ (Suc\ n) \leq j$

let $?k = section\text{-}of\ s\ j$
from *in-section-of st*
have *rs*: $j \in section\ s\ ?k$ **by** *simp*

with *st* **have** $s\ (Suc\ n) < s\ (Suc\ ?k)$ **by** *simp*
with *increasing-bij* **have** $n < ?k$ **by** *simp*
with *rs* **and** *fths*[*of ?k*]
show *eqlat* $p\ \vartheta\ j$ **by** (*simp add*:*is-fthread-def*)
qed

lemma *mk-inf-desc-thread*:
assumes [*simp*]: *increasing* s
assumes *fths*: $\bigwedge i.\ i > n \Longrightarrow$ *is-fthread* $\vartheta\ p\ (s\ i)\ (s\ (Suc\ i))$
assumes *fdths*: $\exists_\infty i.$ *is-desc-fthread* $\vartheta\ p\ (s\ i)\ (s\ (Suc\ i))$

shows *is-desc-thread* ϑ *p*
unfolding *is-desc-thread-def*
proof (*intro exI conjI*)

from *mk-inf-thread*[*of s n* ϑ *p*] *fths*
show $\forall\, i \geq s\ (Suc\ n).\ eqlat\ p\ \vartheta\ i$
by (*fold is-thread-def*) *simp*

show $\exists_\infty l.\ descat\ p\ \vartheta\ l$
unfolding *INF-nat*
proof
fix *i*

let $?k = section\text{-}of\ s\ i$
from *fdths* **obtain** *j*
where $?k < j$ *is-desc-fthread* ϑ *p* (*s j*) (*s* (*Suc j*))
unfolding *INF-nat* **by** *auto*
then obtain *l* **where** $s\ j \leq l$ **and** *desc*: *descat p* ϑ *l*
unfolding *is-desc-fthread-def*
by *auto*

have $i < s\ (Suc\ ?k)$ **by** (*rule section-of2*) *simp*
also have $\ldots \leq s\ j$
by (*rule increasing-weak* [*OF* ‹*increasing s*›]) (*insert* ‹$?k < j$›, *arith*)
also note ‹$\ldots \leq l$›
finally have $i < l$ **.**
with *desc*
show $\exists\, l.\ i < l \wedge descat\ p\ \vartheta\ l$ **by** *blast*
qed
qed

lemma *desc-ex-choice*:
assumes *A*: $((\exists\, n.\forall\, i \geq n.\ \exists\, x.\ P\ x\ i) \wedge (\exists_\infty i.\ \exists\, x.\ Q\ x\ i))$
and *imp*: $\bigwedge x\ i.\ Q\ x\ i \Longrightarrow P\ x\ i$
shows $\exists\, xs.\ ((\exists\, n.\forall\, i \geq n.\ P\ (xs\ i)\ i) \wedge (\exists_\infty i.\ Q\ (xs\ i)\ i))$
(**is** $\exists\, xs.\ ?Ps\ xs \wedge ?Qs\ xs$)
proof
let $?w = \lambda i.\ (if\ (\exists\, x.\ Q\ x\ i)\ then\ (SOME\ x.\ Q\ x\ i)$
$else\ (SOME\ x.\ P\ x\ i))$

from *A*
obtain *n* **where** *P*: $\bigwedge i.\ n \leq i \Longrightarrow \exists\, x.\ P\ x\ i$
by *auto*
{
fix *i*::*'a* **assume** $n \leq i$

have *P* (*?w i*) *i*
proof (*cases* $\exists\, x.\ Q\ x\ i$)

case *True*
hence Q (*?w i*) *i* **by** (*auto intro*:*someI*)
with *imp* **show** P (*?w i*) *i* **.**
next
case *False*
with $P[OF \langle n \leq i \rangle]$ **show** P (*?w i*) *i*
by (*auto intro*:*someI*)
qed
}

hence *?Ps ?w* **by** (*rule-tac x=n* **in** *exI*) *auto*

moreover
from *A* **have** $\exists_\infty i.\ (\exists x.\ Q\ x\ i)$ **..**
hence *?Qs ?w* **by** (*rule INF-mono*) (*auto intro*:*someI*)
ultimately
show *?Ps ?w* $\wedge$ *?Qs ?w* **..**
qed

lemma *dthreads-join*:
assumes [*simp*]: *increasing s*
assumes *dthread*: *is-desc-thread* ϑ (*contract s p*)
shows $\exists \vartheta s.\ desc\ (\lambda i.\ is\text{-}fthread\ (\vartheta s\ i)\ p\ (s\ i)\ (s\ (Suc\ i))$
$\wedge\ \vartheta s\ i\ (s\ i) = \vartheta\ i$
$\wedge\ \vartheta s\ i\ (s\ (Suc\ i)) = \vartheta\ (Suc\ i))$
$(\lambda i.\ is\text{-}desc\text{-}fthread\ (\vartheta s\ i)\ p\ (s\ i)\ (s\ (Suc\ i))$
$\wedge\ \vartheta s\ i\ (s\ i) = \vartheta\ i$
$\wedge\ \vartheta s\ i\ (s\ (Suc\ i)) = \vartheta\ (Suc\ i))$
apply (*rule desc-ex-choice*)
apply (*insert dthread*)
apply (*simp only*:*is-desc-thread-def*)
apply (*simp add*:*eqlat-contract*)
apply (*simp add*:*descat-contract*)
apply (*simp only*:*has-fth-def has-desc-fth-def*)
by (*auto simp*:*is-desc-fthread-def*)

lemma *INF-drop-prefix*:
$(\exists_\infty i{::}nat.\ i > n \wedge P\ i) = (\exists_\infty i.\ P\ i)$
apply (*auto simp*:*INF-nat*)
apply (*drule-tac x = max m n* **in** *spec*)
apply (*elim exE conjE*)
apply (*rule-tac x = na* **in** *exI*)
by *auto*

lemma *contract-keeps-threads*:
  assumes *inc*[*simp*]: *increasing s*
  shows $(\exists\,\vartheta.\ \textit{is-desc-thread}\ \vartheta\ p)$
  $\longleftrightarrow (\exists\,\vartheta.\ \textit{is-desc-thread}\ \vartheta\ (\textit{contract}\ s\ p))$
  (**is** *?A* $\longleftrightarrow$ *?B*)
proof
  assume *?A*
  then obtain ϑ *n*
    where *fr*: $\forall\, i \geq n.\ \textit{eqlat}\ p\ \vartheta\ i$
      and *ds*: $\exists_\infty i.\ \textit{descat}\ p\ \vartheta\ i$
    unfolding *is-desc-thread-def*
    by *auto*

  let $?c\vartheta = \lambda i.\ \vartheta\ (s\ i)$

  have *is-desc-thread* $?c\vartheta$ (*contract s p*)
    unfolding *is-desc-thread-def*
  proof (*intro exI conjI*)

    show $\forall\, i \geq n.\ \textit{eqlat}\ (\textit{contract}\ s\ p)\ ?c\vartheta\ i$
    proof (*intro allI impI*)
      fix *i* **assume** $n \leq i$
      also have $i \leq s\ i$
        using *increasing-inc* **by** *auto*
      finally have $n \leq s\ i$ **.**

      with *fr* **have** *is-fthread* ϑ *p* (*s i*) (*s* (*Suc i*))
        unfolding *is-fthread-def* **by** *auto*
      hence *has-fth p* (*s i*) (*s* (*Suc i*)) (ϑ (*s i*)) (ϑ (*s* (*Suc i*)))
        unfolding *has-fth-def* **by** *auto*
      with *less-imp-le*[*OF increasing-strict*]
      have *eql* (*prod* ($p\langle s\ i, s\ (Suc\ i)\rangle$)) ($\vartheta$ (*s i*)) (ϑ (*s* (*Suc i*)))
        by (*simp add*:*Lemma7a*)
      thus *eqlat* (*contract s p*) $?c\vartheta$ *i* **unfolding** *contract-def*
        by *auto*
    qed

    show $\exists_\infty i.\ \textit{descat}\ (\textit{contract}\ s\ p)\ ?c\vartheta\ i$
      unfolding *INF-nat*
    proof
      fix *i*

      let *?K* = *section-of s* (*max* (*s* (*Suc i*)) *n*)
      from $\langle\exists_\infty i.\ \textit{descat}\ p\ \vartheta\ i\rangle$ **obtain** *j*
          where *s* (*Suc ?K*) < *j descat p* ϑ *j*
        unfolding *INF-nat* **by** *blast*

      let *?L* = *section-of s j*

```
{
  fix x assume r: x ∈ section s ?L

  have e1: max (s (Suc i)) n < s (Suc ?K) by (rule section-of2) simp
  note ‹s (Suc ?K) < j›
  also have j < s (Suc ?L)
    by (rule section-of2) simp
  finally have Suc ?K ≤ ?L
    by (simp add:increasing-bij)
  with increasing-weak have s (Suc ?K) ≤ s ?L by simp
  with e1 r have max (s (Suc i)) n < x by simp

  hence (s (Suc i)) < x n < x by auto
}
note range-est = this

have is-desc-fthread ϑ p (s ?L) (s (Suc ?L))
  unfolding is-desc-fthread-def is-fthread-def
proof
  show ∀ m∈section s ?L. eqlat p ϑ m
  proof
    fix m assume m∈section s ?L
    with range-est(2) have n < m .
    with fr show eqlat p ϑ m by simp
  qed

  from in-section-of inc less-imp-le[OF ‹s (Suc ?K) < j›]
  have j ∈ section s ?L .

  with ‹descat p ϑ j›
  show ∃ m∈section s ?L. descat p ϑ m ..
qed

with less-imp-le[OF increasing-strict]
have a: descat (contract s p) ?cϑ ?L
  unfolding contract-def Lemma7b[symmetric]
  by (auto simp:Lemma7b[symmetric] has-desc-fth-def)

have i < ?L
proof (rule classical)
  assume ¬ i < ?L
  hence s ?L < s (Suc i)
    by (simp add:increasing-bij)
  also have ... < s ?L
    by (rule range-est(1)) (simp add:increasing-strict)
  finally show ?thesis .
qed
with a show ∃ l. i < l ∧ descat (contract s p) ?cϑ l
  by blast
```

qed
qed
with *exI* **show** *?B* **.**
next
assume *?B*
then obtain ϑ
where *dthread*: *is-desc-thread* ϑ (*contract s p*) **..**

with *dthreads-join inc*
obtain ϑs **where** *ths-spec*:
desc (λi. *is-fthread* (ϑs *i*) *p* (*s i*) (*s* (*Suc i*))
$\wedge$ ϑs *i* (*s i*) = ϑ *i*
$\wedge$ ϑs *i* (*s* (*Suc i*)) = ϑ (*Suc i*))
(λi. *is-desc-fthread* (ϑs *i*) *p* (*s i*) (*s* (*Suc i*))
$\wedge$ ϑs *i* (*s i*) = ϑ *i*
$\wedge$ ϑs *i* (*s* (*Suc i*)) = ϑ (*Suc i*))
(**is** *desc ?alw ?inf*)
by *blast*

then obtain *n* **where** *fr*: $\forall i \geq n$. *?alw i* **by** *blast*
hence *connected*: $\bigwedge i.\ n < i \Longrightarrow \vartheta s\ i\ (s\ (Suc\ i)) = \vartheta s\ (Suc\ i)\ (s\ (Suc\ i))$
by *auto*

let *?j*ϑ = *connect s* ϑs

from *fr ths-spec* **have** *ths-spec2*:
$\bigwedge i.\ i > n \Longrightarrow$ *is-fthread* (ϑs *i*) *p* (*s i*) (*s* (*Suc i*))
$\exists_\infty i$. *is-desc-fthread* (ϑs *i*) *p* (*s i*) (*s* (*Suc i*))
by (*auto intro*:*INF-mono*)

have *p1*: $\bigwedge i.\ i > n \Longrightarrow$ *is-fthread ?j*ϑ *p* (*s i*) (*s* (*Suc i*))
by (*rule connect-threads*) (*auto simp*:*connected ths-spec2*)

from *ths-spec2*(*2*)
have $\exists_\infty i.\ n < i \wedge$ *is-desc-fthread* (ϑs *i*) *p* (*s i*) (*s* (*Suc i*))
unfolding *INF-drop-prefix* **.**

hence *p2*: $\exists_\infty i$. *is-desc-fthread ?j*ϑ *p* (*s i*) (*s* (*Suc i*))
apply (*rule INF-mono*)
apply (*rule connect-dthreads*)
by (*auto simp*:*connected*)

with ‹*increasing s*› *p1*
have *is-desc-thread ?j*ϑ *p*
by (*rule mk-inf-desc-thread*)
with *exI* **show** *?A* **.**
qed

lemma *repeated-edge*:
  assumes $\bigwedge i.\ i > n \Longrightarrow dsc\ (snd\ (p\ i))\ k\ k$
  shows *is-desc-thread* $(\lambda i.\ k)\ p$
proof−
  have *th*: $\forall\ m.\ \exists\, na{>}m.\ n < na$ **by** *arith*
  show *?thesis* **using** *prems*
  unfolding *is-desc-thread-def*
  apply (*auto*)
  apply (*rule-tac* $x{=}Suc\ n$ **in** *exI*, *auto*)
  apply (*rule INF-mono*[**where** $P{=}\lambda i.\ n < i$])
  apply (*simp only*:*INF-nat*)
  by (*auto simp add*: *th*)
qed

lemma *fin-from-inf*:
  assumes *is-thread* $n\ \vartheta\ p$
  assumes $n < i$
  assumes $i < j$
  shows *is-fthread* $\vartheta\ p\ i\ j$
  using *prems*
  unfolding *is-thread-def is-fthread-def*
  by *auto*

5.4 Ramsey's Theorem

definition
  set2pair $S = (THE\ (x,y).\ x < y \wedge S = \{x,y\})$

lemma *set2pair-conv*:
  fixes $x\ y$ :: *nat*
  assumes $x < y$
  shows *set2pair* $\{x,\ y\} = (x,\ y)$
  unfolding *set2pair-def*
proof (*rule the-equality*, *simp-all only*:*split-conv split-paired-all*)
  from ‹$x < y$› **show** $x < y \wedge \{x,y\}{=}\{x,y\}$ **by** *simp*
next
  fix $a\ b$
  assume a: $a < b \wedge \{x,\ y\} = \{a,\ b\}$
  hence $\{a,\ b\} = \{x,\ y\}$ **by** *simp-all*
  hence $(a,\ b) = (x,\ y) \vee (a,\ b) = (y,\ x)$
    by (*cases* $x = y$) *auto*
  thus $(a,\ b) = (x,\ y)$
  proof
    assume $(a,\ b) = (y,\ x)$
    with a **and** ‹$x < y$›
    show *?thesis* **by** *auto*
  qed
qed

definition
  set2list = *inv set*

lemma *finite-set2list*:
  assumes *finite S*
  shows *set* (*set2list S*) = *S*
  unfolding *set2list-def*
proof (*rule f-inv-f*)
  from ‹*finite S*› **have** $\exists\, l.\ set\ l = S$
    by (*rule finite-list*)
  thus $S \in range\ set$
    unfolding *image-def*
    by *auto*
qed

corollary *RamseyNatpairs*:
  fixes $S :: 'a\ set$
    and $f :: nat \times nat \Rightarrow 'a$

  assumes *finite S*
  and *inS*: $\bigwedge x\ y.\ x < y \Longrightarrow f\ (x,\ y) \in S$

  obtains $T :: nat\ set$ **and** $s :: 'a$
  where *infinite T*
    and $s \in S$
    and $\bigwedge x\ y.\ [\![x \in T;\ y \in T;\ x < y]\!] \Longrightarrow f\ (x,\ y) = s$
proof −
  from ‹*finite S*›
  have *set* (*set2list S*) = *S* **by** (*rule finite-set2list*)
  then
  obtain *l* **where** *S*: *S* = *set l* **by** *auto*
  also from *set-conv-nth* **have** $\ldots = \{l\ !\ i\ |i.\ i < length\ l\}$ **.**
  finally have $S = \{l\ !\ i\ |i.\ i < length\ l\}$ **.**

  let *?s* = *length l*

  from *inS*
  have *index-less*: $\bigwedge x\ y.\ x \neq y \Longrightarrow index\text{-}of\ l\ (f\ (set2pair\ \{x,\ y\})) < ?s$
  proof −
    fix $x\ y :: nat$
    assume *neq*: $x \neq y$
    have $f\ (set2pair\ \{x,\ y\}) \in S$
    proof (*cases* $x < y$)
      case *True* **hence** $set2pair\ \{x,\ y\} = (x,\ y)$
        by (*rule set2pair-conv*)
      with *True inS*
      show *?thesis* **by** *simp*
    next

```
      case False
      with neq have y-less: y < x by simp
      have x:{x,y} = {y,x} by auto
      with y-less have set2pair {x, y} = (y, x)
        by (simp add:set2pair-conv)
      with y-less inS
      show ?thesis by simp
    qed

    thus index-of l (f (set2pair {x, y})) < length l
      by (simp add: S index-of-length)
  qed

  have ∃ Y. infinite Y ∧
    (∃ t. t < ?s ∧
         (∀ x∈Y. ∀ y∈Y. x ≠ y ⟶
                       index-of l (f (set2pair {x, y})) = t))
    by (rule Ramsey2[of UNIV::nat set, simplified])
       (auto simp:index-less)
  then obtain T i
    where inf: infinite T
    and i: i < length l
    and d: ⋀x y. ⟦x ∈ T; y∈T; x ≠ y⟧
    ⟹ index-of l (f (set2pair {x, y})) = i
    by auto

  have l ! i ∈ S unfolding S using i
    by (rule nth-mem)
  moreover
  have ⋀x y. x ∈ T ⟹ y∈T ⟹ x < y
    ⟹ f (x, y) = l ! i
  proof −
    fix x y assume x ∈ T y ∈ T x < y
    with d have
      index-of l (f (set2pair {x, y})) = i by auto
    with ‹x < y›
    have i = index-of l (f (x, y))
      by (simp add:set2pair-conv)
    with ‹i < length l›
    show f (x, y) = l ! i
      by (auto intro:index-of-member[symmetric] iff:index-of-length)
  qed
  moreover note inf
  ultimately
  show ?thesis using prems
    by blast
qed
```

5.5 Main Result

theorem *LJA-Theorem4*:
  assumes *finite-acg A*
  shows $SCT\ A \longleftrightarrow SCT'\ A$
proof
  assume *SCT A*

  show $SCT'\ A$
  proof (*rule classical*)
    assume $\neg\ SCT'\ A$

    then obtain *n G*
      where *in-closure*: $(tcl\ A) \vdash n \leadsto^{G} n$
      and *idemp*: $G * G = G$
      and *no-strict-arc*: $\forall p.\ \neg(G \vdash p \leadsto^{\downarrow} p)$
      unfolding *SCT′-def no-bad-graphs-def* **by** *auto*

    from *in-closure* **obtain** *k*
      where *k-pow*: $A\ \hat{}\ k \vdash n \leadsto^{G} n$
      and $0 < k$
      unfolding *in-tcl* **by** *auto*

    from *power-induces-path k-pow*
    obtain *loop* **where** *loop-props*:
      has-fpath A loop
      $n = fst\ loop$ $n = end\text{-}node\ loop$
      $G = prod\ loop$ $k = length\ (snd\ loop)$ **.**

    with ‹$0 < k$› **and** *path-loop-graph*
    have *has-ipath A* (*omega loop*) **by** *blast*
    with ‹*SCT A*›
    have *thread*: $\exists \vartheta.\ is\text{-}desc\text{-}thread\ \vartheta\ (omega\ loop)$ **by** (*auto simp*:*SCT-def*)

    let $?s = \lambda i.\ k * i$
    let $?cp = \lambda i{::}nat.\ (n,\ G)$

    from *loop-props* **have** $fst\ loop = end\text{-}node\ loop$ **by** *auto*
    with ‹$0 < k$› ‹$k = length\ (snd\ loop)$›
    have $\bigwedge i.\ (omega\ loop)\langle ?s\ i, ?s\ (Suc\ i)\rangle = loop$
      by (*rule sub-path-loop*)

    with ‹$n = fst\ loop$› ‹$G = prod\ loop$› ‹$k = length\ (snd\ loop)$›
    have *a*: $contract\ ?s\ (omega\ loop) = ?cp$
      unfolding *contract-def*
      by (*simp add*:*path-loop-def split-def fst-p0*)

    from ‹$0 < k$› **have** *increasing ?s*
      by (*auto simp*:*increasing-def*)
    with *thread* **have** $\exists \vartheta.\ is\text{-}desc\text{-}thread\ \vartheta\ ?cp$

  unfolding *a*[*symmetric*]
  by (*unfold contract-keeps-threads*[*symmetric*])

then obtain ϑ **where** *desc*: *is-desc-thread* ϑ *?cp* **by** *auto*

then obtain n **where** *thr*: *is-thread* n ϑ *?cp*
  unfolding *is-desc-thread-def is-thread-def*
  by *auto*

have *finite* (*range* ϑ)
proof (*rule finite-range-ignore-prefix*)

  from ‹*finite-acg A*›
  have *finite-acg* (*tcl A*) **by** (*simp add*:*finite-tcl*)
  with *in-closure* **have** *finite-graph G*
    unfolding *finite-acg-def all-finite-def* **by** *blast*
  thus *finite* (*nodes G*) **by** (*rule finite-nodes*)

  from *thread-image-nodes*[*OF thr*]
  show $\forall\, i \geq n.\ \vartheta\ i \in$ *nodes G* **by** *simp*
qed
with *finite-range*
obtain p **where** *inf-visit*: $\exists_\infty i.\ \vartheta\ i = p$ **by** *auto*

then obtain i **where** $n < i$ $\vartheta\ i = p$
  by (*auto simp*:*INF-nat*)

from *desc*
have $\exists_\infty i.$ *descat ?cp* $\vartheta\ i$
  unfolding *is-desc-thread-def* **by** *auto*
then obtain j
  where $i < j$ **and** *descat ?cp* $\vartheta\ j$
  unfolding *INF-nat* **by** *auto*
from *inf-visit* **obtain** k **where** $j < k$ $\vartheta\ k = p$
  by (*auto simp*:*INF-nat*)

from ‹$i < j$› ‹$j < k$› ‹$n < i$› *thr*
  fin-from-inf[*of n* ϑ *?cp*]
  ‹*descat ?cp* $\vartheta\ j$›
have *is-desc-fthread* ϑ *?cp i k*
  unfolding *is-desc-fthread-def*
  by *auto*

with ‹$\vartheta\ k = p$› ‹$\vartheta\ i = p$›
have *dfth*: *has-desc-fth ?cp i k p p*
  unfolding *has-desc-fth-def*
  by *auto*

from ‹$i < j$› ‹$j < k$› **have** $i < k$ **by** *auto*

hence *prod* (*?cp*⟨*i*, *k*⟩) = *G*
proof (*induct i rule*:*strict-inc-induct*)
case *base* **thus** *?case* **by** (*simp add*:*sub-path-def*)
next
case (*step i*) **thus** *?case*
by (*simp add*:*sub-path-def upt-rec*[*of i k*] *idemp*)
qed

with ‹*i* < *j*› ‹*j* < *k*› *dfth Lemma7b*[*of i k ?cp p p*]
have *dsc G p p* **by** *auto*
with *no-strict-arc* **have** *False* **by** *auto*
thus *?thesis* **..**
qed
next
assume *SCT′ A*

show *SCT A*
proof (*rule classical*)
assume ¬ *SCT A*

with *SCT-def*
obtain *p*
where *ipath*: *has-ipath A p*
and *no-desc-th*: ¬ (∃ϑ. *is-desc-thread* ϑ *p*)
by *blast*

from ‹*finite-acg A*›
have *finite-acg* (*tcl A*) **by** (*simp add*: *finite-tcl*)
hence *finite* (*dest-graph* (*tcl A*)) (**is** *finite ?AG*)
by (*simp add*: *finite-acg-def finite-graph-def*)

from *pdesc-acgplus*[*OF ipath*]
have *a*: ⋀*x y*. *x*<*y* ⟹ *pdesc p*⟨*x*,*y*⟩ ∈ *dest-graph* (*tcl A*)
unfolding *has-edge-def* **.**

obtain *S G*
where *infinite S G* ∈ *dest-graph* (*tcl A*)
and *all-G*: ⋀*x y*. ⟦ *x* ∈ *S*; *y* ∈ *S*; *x* < *y*⟧ ⟹
pdesc (*p*⟨*x*,*y*⟩) = *G*
apply (*rule RamseyNatpairs*[*of ?AG* λ(*x*,*y*). *pdesc p*⟨*x*, *y*⟩])
apply (*rule* ‹*finite ?AG*›)
by (*simp only*:*split-conv*, *rule a*, *auto*)

obtain *n H m* **where**
G-struct: *G* = (*n*, *H*, *m*) **by** (*cases G*)

let *?s* = *enumerate S*
let *?q* = *contract ?s p*

note *all-in-S*[*simp*] = *enumerate-in-set*[*OF* ‹*infinite S*›]
  from ‹*infinite S*›
have *inc*[*simp*]: *increasing ?s*
 unfolding *increasing-def* **by** (*simp add*:*enumerate-mono*)
note *increasing-bij*[*OF this*, *simp*]

from *ipath-contract inc ipath*
have *has-ipath* (*tcl A*) *?q* **.**

from *all-G G-struct*
have *all-H*: $\bigwedge i$. (*snd* (*?q i*)) = *H*
   unfolding *contract-def*
 by *simp*

have *loop*: (*tcl A*) $\vdash n \leadsto^{H} n$
 and *idemp*: $H * H = H$
proof −
 let *?i* = *?s 0* **and** *?j* = *?s* (*Suc 0*) **and** *?k* = *?s* (*Suc* (*Suc 0*))

 have *pdesc* (*p*⟨*?i*,*?j*⟩) = *G*
   and *pdesc* (*p*⟨*?j*,*?k*⟩) = *G*
   and *pdesc* (*p*⟨*?i*,*?k*⟩) = *G*
   using *all-G*
   by *auto*

 with *G-struct*
 have *m* = *end-node* (*p*⟨*?i*,*?j*⟩)
     n = *fst* (*p*⟨*?j*,*?k*⟩)
     and *Hs*: *prod* (*p*⟨*?i*,*?j*⟩) = *H*
     prod (*p*⟨*?j*,*?k*⟩) = *H*
     prod (*p*⟨*?i*,*?k*⟩) = *H*
   by *auto*

 hence *m* = *n* **by** *simp*
 thus *tcl A* $\vdash n \leadsto^{H} n$
   using *G-struct* ‹$G \in$ *dest-graph* (*tcl A*)›
   by (*simp add*:*has-edge-def*)

 from *sub-path-prod*[*of ?i ?j ?k p*]
 show $H * H = H$
   unfolding *Hs* **by** *simp*
qed
moreover have $\bigwedge k$. ¬*dsc H k k*
proof
 fix *k* :: *'a* **assume** *dsc H k k*

 with *all-H repeated-edge*
 have ∃ϑ. *is-desc-thread* ϑ *?q* **by** *fast*
  with *inc* **have** ∃ϑ. *is-desc-thread* ϑ *p*

by (*subst contract-keeps-threads*)
with *no-desc-th*
show *False* **..**
qed
ultimately
have *False*
using ⟨*SCT′ A*⟩[*unfolded SCT′-def no-bad-graphs-def*]
by *blast*
thus *?thesis* **..**
qed
qed

end

6 Applying SCT to function definitions

theory *Interpretation*
imports *Main Misc-Tools Criterion*
begin

definition
$idseq\ R\ s\ x = (s\ 0 = x \land (\forall i.\ R\ (s\ (Suc\ i))\ (s\ i)))$

lemma *not-acc-smaller*:
assumes *notacc*: $\neg\ accp\ R\ x$
shows $\exists y.\ R\ y\ x \land \neg\ accp\ R\ y$
proof (*rule classical*)
assume $\neg$ *?thesis*
hence $\bigwedge y.\ R\ y\ x \Longrightarrow accp\ R\ y$ **by** *blast*
with *accp.accI* **have** $accp\ R\ x$ **.**
with *notacc* **show** *?thesis* **by** *contradiction*
qed

lemma *non-acc-has-idseq*:
assumes $\neg\ accp\ R\ x$
shows $\exists s.\ idseq\ R\ s\ x$
proof −

have $\exists f.\ \forall x.\ \neg accp\ R\ x \longrightarrow R\ (f\ x)\ x \land \neg accp\ R\ (f\ x)$
by (*rule choice*, *auto simp*:*not-acc-smaller*)

then obtain f **where**
in-R: $\bigwedge x.\ \neg accp\ R\ x \Longrightarrow R\ (f\ x)\ x$
and *nia*: $\bigwedge x.\ \neg accp\ R\ x \Longrightarrow \neg accp\ R\ (f\ x)$
by *blast*

let $?s = \lambda i.\ (f\ \hat{}\ i)\ x$

```
{
    fix i
    have ¬accp R (?s i)
      by (induct i) (auto simp:nia ‹¬accp R x›)
    hence R (f (?s i)) (?s i)
      by (rule in-R)
}

hence idseq R ?s x
    unfolding idseq-def
    by auto

thus ?thesis by auto
qed
```

```
types ('a, 'q) cdesc =
  ('q ⇒ bool) × ('q ⇒ 'a) ×('q ⇒ 'a)
```

```
fun in-cdesc :: ('a, 'q) cdesc ⇒ 'a ⇒ 'a ⇒ bool
where
  in-cdesc (Γ, r, l) x y = (∃ q. x = r q ∧ y = l q ∧ Γ q)

fun mk-rel :: ('a, 'q) cdesc list ⇒ 'a ⇒ 'a ⇒ bool
where
  mk-rel [] x y = False
| mk-rel (c#cs) x y =
  (in-cdesc c x y ∨ mk-rel cs x y)
```

```
lemma some-rd:
  assumes mk-rel rds x y
  shows ∃ rd∈set rds. in-cdesc rd x y
  using assms
  by (induct rds) (auto simp:in-cdesc-def)
```

```
lemma ex-cs:
  assumes idseq: idseq (mk-rel rds) s x
  shows ∃ cs. ∀ i. cs i ∈ set rds ∧ in-cdesc (cs i) (s (Suc i)) (s i)
proof −
  from idseq
  have a: ∀ i. ∃ rd ∈ set rds. in-cdesc rd (s (Suc i)) (s i)
    by (auto simp:idseq-def intro:some-rd)
```

show *?thesis*
    by (*rule choice*) (*insert a*, *blast*)
qed

types $'a$ *measures* = *nat* ⇒ $'a$ ⇒ *nat*

fun *stepP* :: ($'a$, $'q$) *cdesc* ⇒ ($'a$, $'q$) *cdesc* ⇒
($'a$ ⇒ *nat*) ⇒ ($'a$ ⇒ *nat*) ⇒ (*nat* ⇒ *nat* ⇒ *bool*) ⇒ *bool*
where
stepP (Γ*1*,*r1*,*l1*) (Γ*2*,*r2*,*l2*) *m1 m2 R*
$= (\forall q_1\ q_2.\ \Gamma 1\ q_1 \wedge \Gamma 2\ q_2 \wedge r1\ q_1 = l2\ q_2$
$\longrightarrow R\ (m2\ (l2\ q_2))\ ((m1\ (l1\ q_1))))$

definition
decr :: ($'a$, $'q$) *cdesc* ⇒ ($'a$, $'q$) *cdesc* ⇒
($'a$ ⇒ *nat*) ⇒ ($'a$ ⇒ *nat*) ⇒ *bool*
where
decr c1 c2 m1 m2 = *stepP c1 c2 m1 m2* (*op* <)

definition
decreq :: ($'a$, $'q$) *cdesc* ⇒ ($'a$, $'q$) *cdesc* ⇒
($'a$ ⇒ *nat*) ⇒ ($'a$ ⇒ *nat*) ⇒ *bool*
where
decreq c1 c2 m1 m2 = *stepP c1 c2 m1 m2* (*op* ≤)

definition
no-step :: ($'a$, $'q$) *cdesc* ⇒ ($'a$, $'q$) *cdesc* ⇒ *bool*
where
no-step c1 c2 = *stepP c1 c2* (λ*x*. *0*) (λ*x*. *0*) (λ*x y*. *False*)

lemma *decr-in-cdesc*:
assumes *in-cdesc RD1 y x*
assumes *in-cdesc RD2 z y*
assumes *decr RD1 RD2 m1 m2*
shows *m2 y* < *m1 x*
using *assms*
by (*cases RD1*, *cases RD2*, *auto simp*:*decr-def*)

lemma *decreq-in-cdesc*:
assumes *in-cdesc RD1 y x*
assumes *in-cdesc RD2 z y*
assumes *decreq RD1 RD2 m1 m2*
shows *m2 y* ≤ *m1 x*
using *assms*

by (*cases RD1*, *cases RD2*, *auto simp*:*decreq-def*)

lemma *no-inf-desc-nat-sequence*:
fixes s :: *nat* $\Rightarrow$ *nat*
assumes *leq*: $\bigwedge i.\ n \leq i \Longrightarrow s\ (Suc\ i) \leq s\ i$
assumes *less*: $\exists_{\infty} i.\ s\ (Suc\ i) < s\ i$
shows *False*
proof −
{
fix $i\ j$:: *nat*
assume $n \leq i$
assume $i \leq j$
{
fix k
have $s\ (i + k) \leq s\ i$
proof (*induct* k)
case *0* **thus** *?case* **by** *simp*
next
case (*Suc* k)
with *leq*[*of* $i + k$] ‹$n \leq i$›
show *?case* **by** *simp*
qed
}
from *this*[*of* $j - i$] ‹$n \leq i$› ‹$i \leq j$›
have $s\ j \leq s\ i$ **by** *auto*
}
note *decr* = *this*

let *?min* = *LEAST* $x.\ x \in$ *range* $(\lambda i.\ s\ (n + i))$
have *?min* $\in$ *range* $(\lambda i.\ s\ (n + i))$
by (*rule LeastI*) *auto*
then obtain k **where** *min*: *?min* $= s\ (n + k)$ **by** *auto*

from *less*
obtain k' **where** $n + k < k'$
and $s\ (Suc\ k') < s\ k'$
unfolding *INF-nat* **by** *auto*

with *decr*[*of* $n + k\ k'$] *min*
have $s\ (Suc\ k') <$ *?min* **by** *auto*
moreover from ‹$n + k < k'$›
have $s\ (Suc\ k') = s\ (n + (Suc\ k' - n))$ **by** *simp*
ultimately
show *False* **using** *not-less-Least* **by** *blast*
qed

definition
approx :: *nat scg* $\Rightarrow$ $('a,\ 'q)$ *cdesc* $\Rightarrow$ $('a,\ 'q)$ *cdesc*
$\Rightarrow$ *'a measures* $\Rightarrow$ *'a measures* $\Rightarrow$ *bool*
where
approx G C C' M M'
$= (\forall\, i\ j.\ (dsc\ G\ i\ j \longrightarrow decr\ C\ C'\ (M\ i)\ (M'\ j))$
$\wedge(eq\ G\ i\ j \longrightarrow decreq\ C\ C'\ (M\ i)\ (M'\ j)))$

lemma *approx-empty*:
approx (*Graph* {}) *c1 c2 ms1 ms2*
unfolding *approx-def has-edge-def dest-graph.simps* **by** *simp*

lemma *approx-less*:
assumes *stepP c1 c2* (*ms1 i*) (*ms2 j*) (*op* <)
assumes *approx* (*Graph Es*) *c1 c2 ms1 ms2*
shows *approx* (*Graph* (*insert* (*i*, ↓, *j*) *Es*)) *c1 c2 ms1 ms2*
using *assms*
unfolding *approx-def has-edge-def dest-graph.simps decr-def*
by *auto*

lemma *approx-leq*:
assumes *stepP c1 c2* (*ms1 i*) (*ms2 j*) (*op* ≤)
assumes *approx* (*Graph Es*) *c1 c2 ms1 ms2*
shows *approx* (*Graph* (*insert* (*i*, ⇓, *j*) *Es*)) *c1 c2 ms1 ms2*
using *assms*
unfolding *approx-def has-edge-def dest-graph.simps decreq-def*
by *auto*

lemma *approx* (*Graph* {(*1*, ↓, *2*),(*2*, ⇓, *3*)}) *c1 c2 ms1 ms2*
apply (*intro approx-less approx-leq approx-empty*)
oops

lemma *no-stepI*:
stepP c1 c2 m1 m2 ($\lambda x\ y.$ *False*)
$\Longrightarrow$ *no-step c1 c2*
by (*cases c1*, *cases c2*) (*auto simp*: *no-step-def*)

definition
sound-int :: *nat acg* $\Rightarrow$ $('a,\ 'q)$ *cdesc list*
$\Rightarrow$ *'a measures list* $\Rightarrow$ *bool*

```
where
  sound-int 𝒜 RDs M =
  (∀ n<length RDs. ∀ m<length RDs.
  no-step (RDs ! n) (RDs ! m) ∨
  (∃ G. (𝒜 ⊢ n ⇝^G m) ∧ approx G (RDs ! n) (RDs ! m) (M ! n) (M ! m)))


lemma length-simps: length [] = 0 length (x#xs) = Suc (length xs)
  by auto

lemma all-less-zero: ∀ n<(0::nat). P n
  by simp

lemma all-less-Suc:
  assumes Pk: P k
  assumes Pn: ∀ n<k. P n
  shows ∀ n<Suc k. P n
proof (intro allI impI)
  fix n assume n < Suc k
  show P n
  proof (cases n < k)
    case True with Pn show ?thesis by simp
  next
    case False with ‹n < Suc k› have n = k by simp
    with Pk show ?thesis by simp
  qed
qed


lemma step-witness:
  assumes in-cdesc RD1 y x
  assumes in-cdesc RD2 z y
  shows ¬ no-step RD1 RD2
  using assms
  by (cases RD1, cases RD2) (auto simp:no-step-def)


theorem SCT-on-relations:
  assumes R: R = mk-rel RDs
  assumes sound: sound-int 𝒜 RDs M
  assumes SCT 𝒜
  shows ∀ x. accp R x
proof (rule, rule classical)
  fix x
  assume ¬ accp R x
  with non-acc-has-idseq
  have ∃ s. idseq R s x .
  then obtain s where idseq R s x ..
```

```
hence ∃cs. ∀i. cs i ∈ set RDs ∧
      in-cdesc (cs i) (s (Suc i)) (s i)
      unfolding R by (rule ex-cs)
then obtain cs where
      [simp]: ⋀i. cs i ∈ set RDs
        and ird[simp]: ⋀i. in-cdesc (cs i) (s (Suc i)) (s i)
      by blast

let ?cis = λi. index-of RDs (cs i)
have ∀i. ∃G. (𝒜 ⊢ ?cis i ↝G (?cis (Suc i)))
      ∧ approx G (RDs ! ?cis i) (RDs ! ?cis (Suc i))
      (M ! ?cis i) (M ! ?cis (Suc i)) (is ∀i. ∃G. ?P i G)
proof
      fix i
      let ?n = ?cis i and ?n' = ?cis (Suc i)

      have in-cdesc (RDs ! ?n) (s (Suc i)) (s i)
        in-cdesc (RDs ! ?n') (s (Suc (Suc i))) (s (Suc i))
        by (simp-all add:index-of-member)
      with step-witness
      have ¬ no-step (RDs ! ?n) (RDs ! ?n') .
      moreover have
        ?n < length RDs
        ?n' < length RDs
        by (simp-all add:index-of-length[symmetric])
      ultimately
      obtain G
        where 𝒜 ⊢ ?n ↝G ?n'
        and approx G (RDs ! ?n) (RDs ! ?n') (M ! ?n) (M ! ?n')
        using sound
        unfolding sound-int-def by auto

      thus ∃G. ?P i G by blast
qed
with choice
have ∃Gs. ∀i. ?P i (Gs i) .
then obtain Gs where
      A: ⋀i. 𝒜 ⊢ ?cis i ↝(Gs i) (?cis (Suc i))
      and B: ⋀i. approx (Gs i) (RDs ! ?cis i) (RDs ! ?cis (Suc i))
      (M ! ?cis i) (M ! ?cis (Suc i))
      by blast

let ?p = λi. (?cis i, Gs i)

from A have has-ipath 𝒜 ?p
      unfolding has-ipath-def
      by auto

with ‹SCT 𝒜› SCT-def
```

obtain *th* **where** *is-desc-thread th ?p*
  by *auto*

then obtain *n*
  where *fr*: $\forall\, i{\geq}n.$ *eqlat ?p th i*
  and *inf*: $\exists_{\infty} i.$ *descat ?p th i*
  unfolding *is-desc-thread-def* **by** *auto*

from *B*
have *approx*:
  $\bigwedge i.$ *approx* (*Gs i*) (*cs i*) (*cs* (*Suc i*))
  (*M* ! *?cis i*) (*M* ! *?cis* (*Suc i*))
  by (*simp add*:*index-of-member*)

let *?seq* = $\lambda i.$ (*M* ! *?cis i*) (*th i*) (*s i*)

have $\bigwedge i.$ $n < i \Longrightarrow$ *?seq* (*Suc i*) $\leq$ *?seq i*
proof −
  fix *i*
  let *?q1* = *th i* **and** *?q2* = *th* (*Suc i*)
  assume $n < i$

  with *fr* **have** *eqlat ?p th i* **by** *simp*
  hence *dsc* (*Gs i*) *?q1 ?q2* $\vee$ *eq* (*Gs i*) *?q1 ?q2*
 by *simp*
  thus *?seq* (*Suc i*) $\leq$ *?seq i*
  proof
   assume *dsc* (*Gs i*) *?q1 ?q2*

   with *approx*
   have *a*:*decr* (*cs i*) (*cs* (*Suc i*))
    ((*M* ! *?cis i*) *?q1*) ((*M* ! *?cis* (*Suc i*)) *?q2*)
    unfolding *approx-def* **by** *auto*

   show *?thesis*
    apply (*rule less-imp-le*)
    apply (*rule decr-in-cdesc*[*of* - *s* (*Suc i*) *s i*])
    by (*rule ird a*)+
  next
   assume *eq* (*Gs i*) *?q1 ?q2*

   with *approx*
   have *a*:*decreq* (*cs i*) (*cs* (*Suc i*))
    ((*M* ! *?cis i*) *?q1*) ((*M* ! *?cis* (*Suc i*)) *?q2*)
    unfolding *approx-def* **by** *auto*

   show *?thesis*
    apply (*rule decreq-in-cdesc*[*of* - *s* (*Suc i*) *s i*])
    by (*rule ird a*)+

```
      qed
  qed
  moreover have ∃∞i. ?seq (Suc i) < ?seq i unfolding INF-nat
  proof
      fix i
      from inf obtain j where i < j and d: descat ?p th j
        unfolding INF-nat by auto
      let ?q1 = th j and ?q2 = th (Suc j)
      from d have dsc (Gs j) ?q1 ?q2 by auto

      with approx
      have a:decr (cs j) (cs (Suc j))
        ((M ! ?cis j) ?q1) ((M ! ?cis (Suc j)) ?q2)
        unfolding approx-def by auto

      have ?seq (Suc j) < ?seq j
        apply (rule decr-in-cdesc[of - s (Suc j) s j])
        by (rule ird a)+
      with ‹i < j›
      show ∃j. i < j ∧ ?seq (Suc j) < ?seq j by auto
  qed
  ultimately have False
    by (rule no-inf-desc-nat-sequence[of Suc n]) simp
  thus accp R x ..
qed

end
```

7 Implemtation of the SCT criterion

```
theory Implementation
imports Correctness
begin

fun edges-match :: ('n × 'e × 'n) × ('n × 'e × 'n) ⇒ bool
where
  edges-match ((n, e, m), (n',e',m')) = (m = n')

fun connect-edges ::
  ('n × ('e::times) × 'n) × ('n × 'e × 'n)
  ⇒ ('n × 'e × 'n)
where
  connect-edges ((n,e,m), (n', e', m')) = (n, e ∗ e', m')

lemma grcomp-code [code]:
 grcomp (Graph G) (Graph H) = Graph (connect-edges ‘ { x ∈ G×H. edges-match
x })
  by (rule graph-ext) (auto simp:graph-mult-def has-edge-def image-def)
```

```
lemma mk-tcl-finite-terminates:
  fixes A :: 'a acg
  assumes fA: finite-acg A
  shows mk-tcl-dom (A, A)
proof −
  from fA have fin-tcl: finite-acg (tcl A)
    by (simp add:finite-tcl)

  hence finite (dest-graph (tcl A))
    unfolding finite-acg-def finite-graph-def ..

  let ?count = λG. card (dest-graph G)
  let ?N = ?count (tcl A)
  let ?m = λX. ?N − (?count X)

  let ?P = λX. mk-tcl-dom (A, X)

  {
    fix X
    assume X ≤ tcl A
    then
    have mk-tcl-dom (A, X)
    proof (induct X rule:measure-induct-rule[of ?m])
      case (less X)
      show ?case
      proof (cases X ∗ A ≤ X)
        case True
        with mk-tcl.domintros show ?thesis by auto
      next
        case False
        then have l: X < X + X ∗ A
          unfolding graph-less-def graph-leq-def graph-plus-def
          by auto

        from ‹X ≤ tcl A›
        have X ∗ A ≤ tcl A ∗ A by (simp add:mult-mono)
        also have . . . ≤ A + tcl A ∗ A by simp
        also have . . . = tcl A by (simp add:tcl-unfold-right[symmetric])
        finally have X ∗ A ≤ tcl A .
        with ‹X ≤ tcl A›
        have X + X ∗ A ≤ tcl A + tcl A
          by (rule add-mono)
        hence less-tcl: X + X ∗ A ≤ tcl A by simp
        hence X < tcl A
          using l ‹X ≤ tcl A› by auto

        from less-tcl fin-tcl
```

```
      have finite-acg (X + X * A) by (rule finite-acg-subset)
      hence finite (dest-graph (X + X * A))
        unfolding finite-acg-def finite-graph-def ..

      hence X: ?count X < ?count (X + X * A)
        using l[simplified graph-less-def graph-leq-def]
        by (rule psubset-card-mono)

      have ?count X < ?N
        apply (rule psubset-card-mono)
        by fact (rule ‹X < tcl A›[simplified graph-less-def])

      with X have ?m (X + X * A) < ?m X by arith

      from  less.hyps this less-tcl
      have mk-tcl-dom (A, X + X * A) .
      with mk-tcl.domintros show ?thesis .
    qed
  qed
 }
 from this less-tcl show ?thesis .
qed


lemma mk-tcl-finite-tcl:
  fixes A :: 'a acg
  assumes fA: finite-acg A
  shows mk-tcl A A = tcl A
  using mk-tcl-finite-terminates[OF fA]
  by (simp only: tcl-def mk-tcl-correctness star-commute)

definition test-SCT :: nat acg ⇒ bool
where
  test-SCT 𝒜 =
  (let 𝒯 = mk-tcl 𝒜 𝒜
    in (∀ (n,G,m)∈dest-graph 𝒯.
        n ≠ m ∨ G * G ≠ G ∨
       (∃ (p::nat,e,q)∈dest-graph G. p = q ∧ e = LESS)))


lemma SCT'-exec:
  assumes fin: finite-acg A
  shows SCT' A = test-SCT A
  using mk-tcl-finite-tcl[OF fin]
  unfolding test-SCT-def Let-def
  unfolding SCT'-def no-bad-graphs-def has-edge-def
  by force

code-modulename SML
```

Implementation Graphs

lemma [*code func*]:
$(G::('a::eq,\ 'b::eq)\ graph) \leq H \longleftrightarrow dest\text{-}graph\ G \subseteq dest\text{-}graph\ H$
$(G::('a::eq,\ 'b::eq)\ graph) < H \longleftrightarrow dest\text{-}graph\ G \subset dest\text{-}graph\ H$
unfolding *graph-leq-def graph-less-def* **by** *rule+*

lemma [*code func*]:
$(G::('a::eq,\ 'b::eq)\ graph) + H = Graph\ (dest\text{-}graph\ G \cup dest\text{-}graph\ H)$
unfolding *graph-plus-def* **..**

lemma [*code func*]:
$(G::('a::eq,\ 'b::\{eq,\ times\})\ graph) * H = grcomp\ G\ H$
unfolding *graph-mult-def* **..**

lemma *SCT′-empty*: $SCT'\ (Graph\ \{\})$
unfolding *SCT′-def no-bad-graphs-def graph-zero-def*[*symmetric*]
tcl-zero
by (*simp add*:*in-grzero*)

7.1 Witness checking

definition $test\text{-}SCT\text{-}witness :: nat\ acg \Rightarrow nat\ acg \Rightarrow bool$
where
$test\text{-}SCT\text{-}witness\ A\ T =$
$(A \leq T \wedge A * T \leq T \wedge$
$(\forall (n,G,m)\in dest\text{-}graph\ T.$
$n \neq m \vee G * G \neq G \vee$
$(\exists (p::nat,e,q)\in dest\text{-}graph\ G.\ p = q \wedge e = LESS)))$

lemma *no-bad-graphs-ucl*:
assumes $A \leq B$
assumes *no-bad-graphs B*
shows *no-bad-graphs A*
using *assms*
unfolding *no-bad-graphs-def has-edge-def graph-leq-def*
by *blast*

lemma *SCT′-witness*:
assumes *a*: *test-SCT-witness A T*
shows $SCT'\ A$
proof −
from *a* **have** $A \leq T\ A * T \leq T$ **by** (*auto simp*:*test-SCT-witness-def*)
hence $A + A * T \leq T$

    by (*subst add-idem*[*of* T, *symmetric*], *rule add-mono*)
  with *star3'* **have** *tcl* $A \leq T$ **unfolding** *tcl-def* **.**
  moreover
  from *a* **have** *no-bad-graphs* T
    unfolding *no-bad-graphs-def test-SCT-witness-def has-edge-def*
    by *auto*
  ultimately
  show *?thesis*
    unfolding *SCT'-def*
    by (*rule no-bad-graphs-ucl*)
qed

code-modulename *SML*
  Graphs SCT
  Kleene-Algebras SCT
  Implementation SCT

export-code *test-SCT* **in** *SML*

end

8 Size-Change Termination

theory *Size-Change-Termination*
imports *Correctness Interpretation Implementation*
uses *sct.ML*
begin

8.1 Simplifier setup

This is needed to run the SCT algorithm in the simplifier:

lemma *setbcomp-simps*:
  $\{x \in \{\}.\ P\ x\} = \{\}$
  $\{x \in \mathit{insert}\ y\ ys.\ P\ x\} = (\mathit{if}\ P\ y\ \mathit{then}\ \mathit{insert}\ y\ \{x \in ys.\ P\ x\}\ \mathit{else}\ \{x \in ys.\ P\ x\})$
  by *auto*

lemma *setbcomp-cong*:
  $A = B \Longrightarrow (\bigwedge x.\ P\ x = Q\ x) \Longrightarrow \{x \in A.\ P\ x\} = \{x \in B.\ Q\ x\}$
  by *auto*

lemma *cartprod-simps*:
  $\{\} \times A = \{\}$
  $\mathit{insert}\ a\ A \times B = \mathit{Pair}\ a\ \text{`}\ B \cup (A \times B)$
  by (*auto simp*:*image-def*)

lemma *image-simps*:

fu ‘ $\{\} = \{\}$
fu ‘ $insert\ a\ A = insert\ (fu\ a)\ (fu$ ‘ $A)$
by (*auto simp*:*image-def*)

lemmas *union-simps* =
Un-empty-left Un-empty-right Un-insert-left

lemma *subset-simps*:
$\{\} \subseteq B$
$insert\ a\ A \subseteq B \equiv a \in B \wedge A \subseteq B$
by *auto*

lemma *element-simps*:
$x \in \{\} \equiv False$
$x \in insert\ a\ A \equiv x = a \vee x \in A$
by *auto*

lemma *set-eq-simp*:
$A = B \longleftrightarrow A \subseteq B \wedge B \subseteq A$ **by** *auto*

lemma *ball-simps*:
$\forall x\in\{\}.\ P\ x \equiv True$
$(\forall x\in insert\ a\ A.\ P\ x) \equiv P\ a \wedge (\forall x\in A.\ P\ x)$
by *auto*

lemma *bex-simps*:
$\exists x\in\{\}.\ P\ x \equiv False$
$(\exists x\in insert\ a\ A.\ P\ x) \equiv P\ a \vee (\exists x\in A.\ P\ x)$
by *auto*

lemmas *set-simps* =
setbcomp-simps
cartprod-simps image-simps union-simps subset-simps
element-simps set-eq-simp
ball-simps bex-simps

lemma *sedge-simps*:
$\downarrow * x = \downarrow$
$\Downarrow * x = x$
by (*auto simp*:*mult-sedge-def*)

lemmas *sctTest-simps* =
simp-thms
if-True
if-False
nat.inject
nat.distinct
Pair-eq

grcomp-code
edges-match.simps
connect-edges.simps

sedge-simps
sedge.distinct
set-simps

graph-mult-def
graph-leq-def
dest-graph.simps
graph-plus-def
graph.inject
graph-zero-def

test-SCT-def
mk-tcl-code

Let-def
split-conv

lemmas *sctTest-congs* =
if-weak-cong let-weak-cong setbcomp-cong

lemma *SCT-Main*:
finite-acg A $\Longrightarrow$ *test-SCT A* $\Longrightarrow$ *SCT A*
using *LJA-Theorem4 SCT′-exec*
by *auto*

end

9 Examples for Size-Change Termination

theory *Examples*
imports *Size-Change-Termination*
begin

function f :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *nat*
where
$f\ n\ 0 = n$
$|\ f\ 0\ (Suc\ m) = f\ (Suc\ m)\ m$
$|\ f\ (Suc\ n)\ (Suc\ m) = f\ m\ n$
by *pat-completeness auto*

termination
unfolding *f-rel-def lfp-const*

apply (*rule SCT-on-relations*)
apply (*tactic Sct.abs-rel-tac*)
apply (*rule ext*, *rule ext*, *simp*)
apply (*tactic Sct.mk-call-graph*)
apply (*rule SCT-Main*)
apply (*simp add:finite-acg-simps*)
by *eval*

function *p* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat*
where
p m n r = (*if r>0 then p m* (*r* − *1*) *n else*
if n>0 then p r (*n* − *1*) *m*
else m)
by *pat-completeness auto*

termination
unfolding *p-rel-def lfp-const*
apply (*rule SCT-on-relations*)
apply (*tactic Sct.abs-rel-tac*)
apply (*rule ext*, *rule ext*, *simp*)
apply (*tactic Sct.mk-call-graph*)
apply (*rule SCT-Main*)
apply (*simp add:finite-acg-ins finite-acg-empty finite-graph-def*)
by *eval*

function *foo* :: *bool* ⇒ *nat* ⇒ *nat* ⇒ *nat*
where
foo True (*Suc n*) *m* = *foo True n* (*Suc m*)
| *foo True 0 m* = *foo False 0 m*
| *foo False n* (*Suc m*) = *foo False* (*Suc n*) *m*
| *foo False n 0* = *n*
by *pat-completeness auto*

termination
unfolding *foo-rel-def lfp-const*
apply (*rule SCT-on-relations*)
apply (*tactic Sct.abs-rel-tac*)
apply (*rule ext*, *rule ext*, *simp*)
apply (*tactic Sct.mk-call-graph*)
apply (*rule SCT-Main*)
apply (*simp add:finite-acg-ins finite-acg-empty finite-graph-def*)
by *eval*

function (*sequential*)
bar :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat*
where
bar 0 (*Suc n*) *m* = *bar m m m*
| *bar k n m* = *0*

by *pat-completeness auto*

termination
unfolding *bar-rel-def lfp-const*
apply (*rule SCT-on-relations*)
apply (*tactic Sct.abs-rel-tac*)
apply (*rule ext*, *rule ext*, *simp*)
apply (*tactic Sct.mk-call-graph*)
apply (*rule SCT-Main*)
apply (*simp add*:*finite-acg-ins finite-acg-empty finite-graph-def*)
by (*simp only*:*sctTest-simps cong*: *sctTest-congs*)

end