Size-Change Termination

Alexander Krauss

June 8, 2008

1 Miscellaneous Tools for Size-Change Termination

theory *Misc-Tools*
imports *Main*
begin

1.1 Searching in lists

fun *index-of* :: *'a list* ⇒ *'a* ⇒ *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* ∈ *set l*) ⟹ (*l* ! *index-of l x* = *x*)
  ⟨*proof*⟩

lemma *index-of-length*:
  (*x* ∈ *set l*) = (*index-of l x* < *length l*)
  ⟨*proof*⟩

1.2 Some reasoning tools

lemma *three-cases*:
  assumes *a1* ⟹ *thesis*
  assumes *a2* ⟹ *thesis*
  assumes *a3* ⟹ *thesis*
  assumes ⋀*R*. ⟦*a1* ⟹ *R*; *a2* ⟹ *R*; *a3* ⟹ *R*⟧ ⟹ *R*
  shows *thesis*
  ⟨*proof*⟩

1.3 Sequences

types
  'a sequence = *nat* ⇒ *'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 $s\ i < s\ j$
⟨*proof*⟩

lemma *increasing-weak*:
assumes *increasing* s
assumes $i \leq j$
shows $s\ i \leq s\ j$
⟨*proof*⟩

lemma *increasing-inc*:
assumes *increasing* s
shows $n \leq s\ n$
⟨*proof*⟩

lemma *increasing-bij*:
assumes [*simp*]: *increasing* s
shows $(s\ i < s\ j) = (i < j)$
⟨*proof*⟩

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 $\exists\, i.\ n < s\ (Suc\ i)$
⟨*proof*⟩

lemma *section-of2*:
assumes *increasing* s
shows $n < s\ (Suc\ (section\text{-}of\ s\ n))$
⟨*proof*⟩

lemma *section-of1*:
assumes [*simp*, *intro*]: *increasing* s
assumes $s\ i \leq n$
shows $s\ (section\text{-}of\ s\ n) \leq n$

⟨*proof*⟩

lemma *section-of-known*:
  assumes [*simp*]: *increasing s*
  assumes *in-sect*: $k \in$ *section s i*
  shows *section-of s k* = *i* (**is** *?s* = *i*)
⟨*proof*⟩

lemma *in-section-of*:
  assumes *increasing s*
  assumes *s i* $\leq k$
  shows $k \in$ *section s* (*section-of s k*)
  ⟨*proof*⟩

end

2 Kleene Algebras

theory *Kleene-Algebras*
imports *Main*
begin

A type class of kleene algebras

class *star* = *type* +
  fixes *star* :: $'a \Rightarrow 'a$

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

lemma *add-idem2*[*simp*]: (x::$'a$::*idem-add*) + ($x + y$) = $x + y$
  ⟨*proof*⟩

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::$'a$::*order-by-add*) $\leq y \Longrightarrow x + y = y$
  ⟨*proof*⟩
lemma *ord-simp2*[*simp*]: (x::$'a$::*order-by-add*) $\leq y \Longrightarrow y + x = y$
  ⟨*proof*⟩
lemma *ord-intro*: (x::$'a$::*order-by-add*) + $y = y \Longrightarrow x \leq y$
  ⟨*proof*⟩

instance *order-by-add* $\subseteq$ *order*
⟨*proof*⟩

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

instance *pre-kleene* $\subseteq$ *pordered-semiring*
⟨*proof*⟩

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 $x :: 'a :: order\text{-}by\text{-}add$
  shows $x \leq z \Longrightarrow y \leq z \Longrightarrow x + y \leq z$
  ⟨*proof*⟩

lemma *le-SUPI'*:
  fixes $l :: 'a :: complete\text{-}lattice$
  assumes $l \leq M\ i$
  shows $l \leq (SUP\ i.\ M\ i)$
  ⟨*proof*⟩

lemma *zero-minimum*[*simp*]: $(0::'a::pre\text{-}kleene) \leq x$
  ⟨*proof*⟩

instance *kleene-by-complete-lattice* $\subseteq$ *kleene*
⟨*proof*⟩

lemma *less-add*[*simp*]:
  fixes $a\ b :: 'a :: order\text{-}by\text{-}add$
  shows $a \leq a + b$
  and $b \leq a + b$
  ⟨*proof*⟩

lemma *add-est1*:
  fixes $a\ b\ c :: 'a :: order\text{-}by\text{-}add$
  assumes *a*: $a + b \leq c$
  shows $a \leq c$
  ⟨*proof*⟩

lemma *add-est2*:
  fixes $a\ b\ c :: 'a :: order\text{-}by\text{-}add$
  assumes *a*: $a + b \leq c$
  shows $b \leq c$
  ⟨*proof*⟩

lemma *star3'*:
fixes $a\ b\ x :: 'a :: kleene$
assumes *a*: $b + a * x \leq x$
shows $star\ a * b \leq x$
⟨*proof*⟩

lemma *star4'*:
fixes $a\ b\ x :: 'a :: kleene$
assumes *a*: $b + x * a \leq x$
shows $b * star\ a \leq x$
⟨*proof*⟩

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

lemma *star-unfold-left*:
fixes $a :: 'a :: kleene$
shows $1 + a * star\ a = star\ a$
⟨*proof*⟩

lemma *star-unfold-right*:
fixes $a :: 'a :: kleene$
shows $1 + star\ a * a = star\ a$
⟨*proof*⟩

lemma *star-zero*[*simp*]:
shows $star\ (0::'a::kleene) = 1$
⟨*proof*⟩

lemma *star-commute*:
fixes $a\ b\ x :: 'a :: kleene$
assumes *a*: $a * x = x * b$
shows $star\ a * x = x * star\ b$
⟨*proof*⟩

lemma *star-assoc*:
fixes $c\ d :: 'a :: kleene$
shows $star\ (c * d) * c = c * star\ (d * c)$
⟨*proof*⟩

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

⟨*proof*⟩

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*⟩

lemma *star-mono*:
fixes *x y* :: *'a* :: *kleene*
assumes $x \leq y$
shows $star\ x \leq star\ y$
⟨*proof*⟩

lemma *x-less-star*[*simp*]:
fixes *x* :: *'a* :: *kleene*
shows $x \leq x * star\ a$
⟨*proof*⟩

2.1 Transitive Closure

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

lemma *tcl-zero*:
$tcl\ (0::'a::kleene) = 0$
⟨*proof*⟩

lemma *tcl-unfold-right*: $tcl\ a = a + tcl\ a * a$
⟨*proof*⟩

lemma *less-tcl*: $a \leq tcl\ a$
⟨*proof*⟩

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
$mk\text{-}tcl\ A\ X = (if\ X * A \leq X\ then\ X\ else\ mk\text{-}tcl\ A\ (X + X * A))$
⟨*proof*⟩

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

lemma *mk-tcl-code*[*code*]:

mk-tcl A X =
(let XA = X ∗ A
in if XA ≤ X then X else mk-tcl A (X + XA))
⟨*proof*⟩

lemma *mk-tcl-lemma1*:
fixes *X* :: *'a* :: *kleene*
shows *(X + X ∗ A) ∗ star A = X ∗ star A*
⟨*proof*⟩

lemma *mk-tcl-lemma2*:
fixes *X* :: *'a* :: *kleene*
shows *X ∗ A ≤ X ⟹ X ∗ star A = X*
⟨*proof*⟩

lemma *mk-tcl-correctness*:
fixes *A X* :: *'a* :: {*kleene*}
assumes *mk-tcl-dom (A, X)*
shows *mk-tcl A X = X ∗ star A*
⟨*proof*⟩

lemma *graph-implies-dom*: *mk-tcl-graph x y ⟹ mk-tcl-dom x*
⟨*proof*⟩

lemma *mk-tcl-default*: ¬ *mk-tcl-dom (a,x) ⟹ mk-tcl a x = 0*
⟨*proof*⟩

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*
⟨*proof*⟩

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*

primrec *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
  ⟨*proof*⟩

lemma *split-graph-all*:
  $(\bigwedge gr.$ *PROP P* $gr) \equiv (\bigwedge set.$ *PROP P* (*Graph set*))
⟨*proof*⟩

definition
  has-edge :: $('n,'e)$ *graph* $\Rightarrow$ $'n \Rightarrow 'e \Rightarrow 'n \Rightarrow$ *bool*
(- ⊢ - ⇝ - -)
where
  has-edge G n e n' = $((n, e, n') \in$ *dest-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 $\bigwedge n\ e\ n'.$ *has-edge* G n e n' = *has-edge* H n e n'
  shows $G = H$
  ⟨*proof*⟩

instantiation *graph* :: (*type*, *type*) *comm-monoid-add*
begin

definition
  graph-zero-def: 0 = *Graph* $\{\}$

definition
  graph-plus-def [*code func del*]: $G + H$ = *Graph* (*dest-graph* $G \cup$ *dest-graph* H)

instance ⟨*proof*⟩

end

instantiation *graph* :: (*type*, *type*) {*distrib-lattice*, *complete-lattice*}
begin

definition
graph-leq-def [*code func del*]: $G \leq H \longleftrightarrow$ *dest-graph* $G \subseteq$ *dest-graph* H

definition
graph-less-def [*code func del*]: $G < H \longleftrightarrow$ *dest-graph* $G \subset$ *dest-graph* H

definition
[*code func del*]: *inf* G H = *Graph* (*dest-graph* $G \cap$ *dest-graph* H)

definition
[*code func del*]: *sup* (G :: ('a, 'b) *graph*) $H = G + H$

definition
Inf-graph-def [*code func del*]: *Inf* = (λ*Gs*. *Graph* ($\bigcap$(*dest-graph* ‘ *Gs*)))

definition
Sup-graph-def [*code func del*]: *Sup* = (λ*Gs*. *Graph* ($\bigcup$(*dest-graph* ‘ *Gs*)))

instance ⟨*proof*⟩

end

lemma *in-grplus*:
has-edge (G + H) p b q = (*has-edge* G p b q ∨ *has-edge* H p b q)
⟨*proof*⟩

lemma *in-grzero*:
has-edge 0 p b q = *False*
⟨*proof*⟩

3.2.1 Multiplicative Structure

instantiation *graph* :: (*type*, *times*) *mult-zero*
begin

definition
graph-mult-def [*code func del*]: $G * H$ = *grcomp* G H

instance ⟨*proof*⟩

end

instantiation *graph* :: (*type*, *one*) *one*

begin

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

instance ⟨*proof*⟩

end

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'*)
⟨*proof*⟩

lemma *in-grunit*:
has-edge 1 p b q = (*p* = *q* ∧ *b* = *1*)
⟨*proof*⟩

instance *graph* :: (*type*, *semigroup-mult*) *semigroup-mult*
⟨*proof*⟩

instantiation *graph* :: (*type*, *monoid-mult*) {*semiring-1*, *idem-add*, *recpower*, *star*}
begin

primrec *power-graph* :: ('*a*::*type*, '*b*::*monoid-mult*) *graph* ⇒ *nat* => ('*a*, '*b*) *graph*
where
(*A* :: ('*a*, '*b*) *graph*) ^ *0* = *1*
| (*A* :: ('*a*, '*b*) *graph*) ^ *Suc n* = *A* ∗ (*A* ^ *n*)

definition
graph-star-def: *star* (*G* :: ('*a*, '*b*) *graph*) = (*SUP n*. *G* ^ *n*)

instance ⟨*proof*⟩

end

lemma *graph-leqI*:
assumes ⋀*n e n'*. *has-edge G n e n'* ⟹ *has-edge H n e n'*
shows *G* ≤ *H*
⟨*proof*⟩

lemma *in-graph-plusE*:
assumes *has-edge* (*G* + *H*) *n e n'*
assumes *has-edge G n e n'* ⟹ *P*
assumes *has-edge H n e n'* ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *in-graph-compE*:

assumes *GH*: *has-edge* (*G* ∗ *H*) *n e n′*
obtains *e1 k e2*
where *has-edge G n e1 k has-edge H k e2 n′ e* = *e1* ∗ *e2*
⟨*proof*⟩

lemma
assumes *x* ∈ *S k*
shows *x* ∈ (⋃ *k*. *S k*)
⟨*proof*⟩

lemma *graph-union-least*:
assumes ⋀*n*. *Graph* (*G n*) ≤ *C*
shows *Graph* (⋃ *n*. *G n*) ≤ *C*
⟨*proof*⟩

lemma *Sup-graph-eq*:
(*SUP n*. *Graph* (*G n*)) = *Graph* (⋃ *n*. *G n*)
⟨*proof*⟩

lemma *has-edge-leq*: *has-edge G p b q* = (*Graph* {(*p*,*b*,*q*)} ≤ *G*)
⟨*proof*⟩

lemma *Sup-graph-eq2*:
(*SUP n*. *G n*) = *Graph* (⋃ *n*. *dest-graph* (*G n*))
⟨*proof*⟩

lemma *in-SUP*:
has-edge (*SUP x*. *Gs x*) *p b q* = (∃ *x*. *has-edge* (*Gs x*) *p b q*)
⟨*proof*⟩

instance *graph* :: (*type*, *monoid-mult*) *kleene-by-complete-lattice*
⟨*proof*⟩

lemma *in-star*:
has-edge (*star G*) *a x b* = (∃ *n*. *has-edge* (*G* ^ *n*) *a x b*)
⟨*proof*⟩

lemma *tcl-is-SUP*:
tcl (*G*::('*a*::*type*, '*b*::*monoid-mult*) *graph*) =
(*SUP n*. *G* ^ (*Suc n*))
⟨*proof*⟩

lemma *in-tcl*:
has-edge (*tcl G*) *a x b* = (∃ *n*>*0*. *has-edge* (*G* ^ *n*) *a x b*)
⟨*proof*⟩

3.3 Infinite Paths

types $('n, 'e)$ *ipath* $= ('n \times 'e)$ *sequence*

definition *has-ipath* :: $('n, 'e)$ *graph* $\Rightarrow$ $('n, 'e)$ *ipath* $\Rightarrow$ *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* $= ('n \times ('e \times 'n)$ *list*$)$

inductive *has-fpath* :: $('n, 'e)$ *graph* $\Rightarrow$ $('n, 'e)$ *fpath* $\Rightarrow$ *bool*
for *G* :: $('n, 'e)$ *graph*
where
has-fpath-empty: *has-fpath G* $(n, [])$
| *has-fpath-join*: $[\![G \vdash n \rightsquigarrow^e n'$; *has-fpath G* $(n', es)]\!] \Longrightarrow$ *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* $\Rightarrow$ *nat* $\Rightarrow$ $('n \times 'e \times 'n)$
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*))
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *path-nth-connected*:
assumes *Suc k* < *length* (*snd p*)
shows *fst* (*path-nth p* (*Suc k*)) = *snd* (*snd* (*path-nth p k*))
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*⟩

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))
⟨*proof*⟩

lemma *power-induces-path*:
assumes *a*: *has-edge (A ^ k) n G m*
obtains *p*
where *has-fpath A p*
and *n = fst p m = end-node p*
and *G = prod p*
and *k = length (snd p)*
⟨*proof*⟩

3.5 Sub-Paths

definition *sub-path* :: (*'n*, *'e*) *ipath* ⇒ *nat* ⇒ *nat* ⇒ (*'n*, *'e*) *fpath*
((-⟨-,-⟩))
where
p⟨i,j⟩ =
(fst (p i), map (λk. (snd (p k), fst (p (Suc k)))) [i ..< j])

lemma *sub-path-is-path*:
assumes *ipath*: *has-ipath G p*
assumes *l*: *i ≤ j*
shows *has-fpath G (p⟨i,j⟩)*
⟨*proof*⟩

lemma *sub-path-start*[*simp*]:
fst (*p*⟨*i*,*j*⟩) = *fst* (*p i*)
⟨*proof*⟩

lemma *nth-upto*[*simp*]: *k* < *j* − *i* ⟹ [*i* ..< *j*] ! *k* = *i* + *k*
⟨*proof*⟩

lemma *sub-path-end*[*simp*]:
i < *j* ⟹ *end-node* (*p*⟨*i*,*j*⟩) = *fst* (*p j*)
⟨*proof*⟩

lemma *foldr-map*: *foldr f* (*map g xs*) = *foldr* (*f o g*) *xs*
⟨*proof*⟩

lemma *upto-append*[*simp*]:
assumes *i* ≤ *j j* ≤ *k*
shows [*i* ..< *j*] @ [*j* ..< *k*] = [*i* ..< *k*]
⟨*proof*⟩

lemma *foldr-monoid*: *foldr* (*op* ∗) *xs 1* ∗ *foldr* (*op* ∗) *ys 1*
= *foldr* (*op* ∗) (*xs* @ *ys*) (*1*::*'a*::*monoid-mult*)
⟨*proof*⟩

lemma *sub-path-prod*:
assumes *i* < *j*
assumes *j* < *k*
shows *prod* (*p*⟨*i*,*k*⟩) = *prod* (*p*⟨*i*,*j*⟩) ∗ *prod* (*p*⟨*j*,*k*⟩)
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *path-acgpow*:
has-fpath G p
⟹ *has-edge* (*G* ^ *length* (*snd p*)) (*fst p*) (*prod p*) (*end-node p*)
⟨*proof*⟩

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*⟩

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*⟩

definition
contract s p =
(λ*i*. (*fst* (*p* (*s i*)), *prod* (*p*⟨*s i*,*s* (*Suc i*)⟩)))

lemma *ipath-contract*:
assumes [*simp*]: *increasing s*
assumes *ipath*: *has-ipath G p*
shows *has-ipath* (*tcl G*) (*contract s p*)
⟨*proof*⟩

lemma *prod-unfold*:
i < *j* ⟹ *prod* (*p*⟨*i*,*j*⟩)
= *snd* (*p i*) ∗ *prod* (*p*⟨*Suc i*, *j*⟩)
⟨*proof*⟩

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*⟩

end

4 The Size-Change Principle (Definition)

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

4.1 Size-Change Graphs

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

instantiation *sedge* :: *comm-monoid-mult*
begin

definition
one-sedge-def: $1 = \Downarrow$

definition
mult-sedge-def: $a * b = (\mathit{if}\ a = \downarrow\ \mathit{then}\ \downarrow\ \mathit{else}\ b)$

instance ⟨*proof*⟩

end

lemma *sedge-UNIV*:
$\mathit{UNIV} = \{\ \mathit{LESS},\ \mathit{LEQ}\ \}$
⟨*proof*⟩

instance *sedge* :: *finite*
⟨*proof*⟩

types $'a\ \mathit{scg} = ('a,\ \mathit{sedge})\ \mathit{graph}$
types $'a\ \mathit{acg} = ('a,\ 'a\ \mathit{scg})\ \mathit{graph}$

4.2 Size-Change Termination

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

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

abbreviation (*input*)
$\mathit{eqp}\ G\ i\ j \equiv \mathit{has\text{-}edge}\ G\ i\ \mathit{LEQ}\ j$

abbreviation
$\mathit{eql} :: 'a\ \mathit{scg} \Rightarrow 'a \Rightarrow 'a \Rightarrow \mathit{bool}$
(- ⊢ - ⇝ -)
where
$\mathit{eql}\ G\ i\ j \equiv \mathit{has\text{-}edge}\ G\ i\ \mathit{LESS}\ j \vee \mathit{has\text{-}edge}\ G\ i\ \mathit{LEQ}\ j$

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

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

abbreviation (*input*) *eqlat* :: ('*a*, '*a scg*) *ipath* ⇒ '*a sequence* ⇒ *nat* ⇒ *bool*
where
eqlat p ϑ i ≡ (*has-edge* (*snd* (*p i*)) (*ϑ i*) *LESS* (*ϑ* (*Suc i*))
∨ *has-edge* (*snd* (*p i*)) (*ϑ i*) *LEQ* (*ϑ* (*Suc i*)))

definition *is-desc-thread* :: '*a sequence* ⇒ ('*a*, '*a scg*) *ipath* ⇒ *bool*
where
is-desc-thread ϑ p = ((∃ *n*.∀ *i*≥*n*. *eqlat p ϑ i*) ∧ (∃ $_\infty$*i*. *descat p ϑ i*))

definition *SCT* :: '*a acg* ⇒ *bool*
where
SCT $\mathcal{A}$ =
(∀ *p*. *has-ipath* $\mathcal{A}$ *p* ⟶ (∃ *ϑ*. *is-desc-thread ϑ p*))

definition *no-bad-graphs* :: '*a acg* ⇒ *bool*
where
no-bad-graphs A =
(∀ *n G*. *has-edge A n G n* ∧ *G* ∗ *G* = *G*
⟶ (∃ *p*. *has-edge G p LESS p*))

definition *SCT'* :: '*a acg* ⇒ *bool*
where
SCT' A = *no-bad-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 ∃ *x*. ∃$_\infty$ *i*. *f i* = *x*
⟨*proof*⟩

lemma *finite-range-ignore-prefix*:
fixes *f* :: *nat* ⇒ *'a*
assumes *fA*: *finite A*
assumes *inA*: ∀ *x*≥*n*. *f x* ∈ *A*
shows *finite* (*range f*)
⟨*proof*⟩

definition
finite-graph G = *finite* (*dest-graph G*)
definition
all-finite G = (∀ *n H m*. *has-edge G n H m* ⟶ *finite-graph H*)
definition
finite-acg A = (*finite-graph A* ∧ *all-finite A*)
definition
nodes G = *fst* ‘ *dest-graph G* ∪ *snd* ‘ *snd* ‘ *dest-graph G*
definition
edges G = *fst* ‘ *snd* ‘ *dest-graph G*
definition
smallnodes G = ⋃(*nodes* ‘ *edges G*)

lemma *thread-image-nodes*:
assumes *th*: *is-thread n* ϑ *p*
shows ∀ *i*≥*n*. ϑ *i* ∈ *nodes* (*snd* (*p i*))
⟨*proof*⟩

lemma *finite-nodes*: *finite-graph* $G \Longrightarrow$ *finite* (*nodes* G)
⟨*proof*⟩

lemma *nodes-subgraph*: $A \leq B \Longrightarrow$ *nodes* $A \subseteq$ *nodes* B
⟨*proof*⟩

lemma *finite-edges*: *finite-graph* $G \Longrightarrow$ *finite* (*edges* G)
⟨*proof*⟩

lemma *edges-sum*[*simp*]: *edges* $(A + B) =$ *edges* $A \cup$ *edges* B
⟨*proof*⟩

lemma *nodes-sum*[*simp*]: *nodes* $(A + B) =$ *nodes* $A \cup$ *nodes* B
⟨*proof*⟩

lemma *finite-acg-subset*:
$A \leq B \Longrightarrow$ *finite-acg* $B \Longrightarrow$ *finite-acg* A
⟨*proof*⟩

lemma *scg-finite*:
fixes $G :: 'a$ *scg*
assumes *fin*: *finite* (*nodes* G)
shows *finite-graph* G
⟨*proof*⟩

lemma *smallnodes-sum*[*simp*]:
smallnodes $(A + B) =$ *smallnodes* $A \cup$ *smallnodes* B
⟨*proof*⟩

lemma *in-smallnodes*:
fixes $A :: 'a$ *acg*
assumes *e*: *has-edge* A x G y
shows *nodes* $G \subseteq$ *smallnodes* A
⟨*proof*⟩

lemma *finite-smallnodes*:
assumes *fA*: *finite-acg* A
shows *finite* (*smallnodes* A)
⟨*proof*⟩

lemma *nodes-tcl*:
nodes (*tcl* A) = *nodes* A
⟨*proof*⟩

lemma *smallnodes-tcl*:
fixes $A :: 'a$ *acg*
shows *smallnodes* (*tcl* A) = *smallnodes* A
⟨*proof*⟩

lemma *finite-nodegraphs*:
assumes *F*: *finite F*
shows *finite* { *G*::'*a scg*. *nodes G* ⊆ *F* } (**is** *finite ?P*)
⟨*proof*⟩

lemma *finite-graphI*:
fixes *A* :: '*a acg*
assumes *fin*: *finite* (*nodes A*) *finite* (*smallnodes A*)
shows *finite-graph A*
⟨*proof*⟩

lemma *smallnodes-allfinite*:
fixes *A* :: '*a acg*
assumes *fin*: *finite* (*smallnodes A*)
shows *all-finite A*
⟨*proof*⟩

lemma *finite-tcl*:
fixes *A* :: '*a acg*
shows *finite-acg* (*tcl A*) ⟷ *finite-acg A*
⟨*proof*⟩

lemma *finite-acg-empty*: *finite-acg* (*Graph* {})
⟨*proof*⟩

lemma *finite-acg-ins*:
assumes *fA*: *finite-acg* (*Graph A*)
assumes *fG*: *finite G*
shows *finite-acg* (*Graph* (*insert* (*a*, *Graph G*, *b*) *A*))
⟨*proof*⟩

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* 𝒜 *p*
and *i* < *j*
shows *has-edge* (*tcl* 𝒜) (*fst* (*p*⟨*i*,*j*⟩)) (*prod* (*p*⟨*i*,*j*⟩)) (*end-node* (*p*⟨*i*,*j*⟩))
⟨*proof*⟩

lemma *combine-fthreads*:
assumes *range*: *i* < *j j* ≤ *k*

shows
has-fth p i k m r =
(∃ *n*. *has-fth p i j m n* ∧ *has-fth p j k n r*) (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *desc-is-fthread*:
is-desc-fthread ϑ p i k ⟹ *is-fthread ϑ p i k*
⟨*proof*⟩

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*⟩

lemma *fth-single*:
has-fth p i (*Suc i*) *m n* = *eql* (*snd* (*p i*)) *m n* (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *desc-fth-single*:
has-desc-fth p i (*Suc i*) *m n* =
dsc (*snd* (*p i*)) *m n* (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *mk-eql*: (*G* ⊢ *m* ⇝^{e} *n*) ⟹ *eql G m n*
⟨*proof*⟩

lemma *eql-scgcomp*:
eql (*G* ∗ *H*) *m r* =
(∃ *n*. *eql G m n* ∧ *eql H n r*) (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *desc-scgcomp*:
dsc (*G* ∗ *H*) *m r* =
(∃ *n*. (*dsc G m n* ∧ *eql H n r*) ∨ (*eqp G m n* ∧ *dsc H n r*)) (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *has-fth-unfold*:
assumes *i* < *j*
shows *has-fth p i j m n* =
(∃ *r*. *has-fth p i* (*Suc i*) *m r* ∧ *has-fth p* (*Suc i*) *j r n*)
⟨*proof*⟩

lemma *has-dfth-unfold*:
assumes *range*: $i < j$
shows
$has\text{-}desc\text{-}fth\ p\ i\ j\ m\ r =$
$(\exists\, n.\ (has\text{-}desc\text{-}fth\ p\ i\ (Suc\ i)\ m\ n \wedge has\text{-}fth\ p\ (Suc\ i)\ j\ n\ r)$
$\vee\ (has\text{-}fth\ p\ i\ (Suc\ i)\ m\ n \wedge has\text{-}desc\text{-}fth\ p\ (Suc\ i)\ j\ n\ r))$
⟨*proof*⟩

lemma *Lemma7a*:
$i \leq j \Longrightarrow has\text{-}fth\ p\ i\ j\ m\ n = eql\ (prod\ (p\langle i,j\rangle))\ m\ n$
⟨*proof*⟩

lemma *Lemma7b*:
assumes $i \leq j$
shows
$has\text{-}desc\text{-}fth\ p\ i\ j\ m\ n =$
$dsc\ (prod\ (p\langle i,j\rangle))\ m\ n$
⟨*proof*⟩

lemma *descat-contract*:
assumes [*simp*]: *increasing s*
shows
$descat\ (contract\ s\ p)\ \vartheta\ i =$
$has\text{-}desc\text{-}fth\ p\ (s\ i)\ (s\ (Suc\ i))\ (\vartheta\ i)\ (\vartheta\ (Suc\ i))$
⟨*proof*⟩

lemma *eqlat-contract*:
assumes [*simp*]: *increasing s*
shows
$eqlat\ (contract\ s\ p)\ \vartheta\ i =$
$has\text{-}fth\ p\ (s\ i)\ (s\ (Suc\ i))\ (\vartheta\ i)\ (\vartheta\ (Suc\ i))$
⟨*proof*⟩

5.3.1 Connecting threads

definition
$connect\ s\ \vartheta s = (\lambda k.\ \vartheta s\ (section\text{-}of\ s\ k)\ k)$

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*⟩

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

shows
is-fthread $(connect\ s\ \vartheta s)\ p\ (s\ i)\ (s\ (Suc\ i))$
⟨*proof*⟩

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

shows
is-desc-fthread $(connect\ s\ \vartheta s)\ p\ (s\ i)\ (s\ (Suc\ i))$
⟨*proof*⟩

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* $(s\ (Suc\ n))\ \vartheta\ p$
⟨*proof*⟩

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* $\vartheta\ p$
⟨*proof*⟩

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*⟩

lemma *dthreads-join*:
assumes [*simp*]: *increasing s*
assumes *dthread*: *is-desc-thread* $\vartheta\ (contract\ s\ p)$
shows $\exists \vartheta s.\ desc\ (\lambda i.\ is\text{-}fthread\ (\vartheta s\ i)\ p\ (s\ i)\ (s\ (Suc\ i))$
$\wedge\ \vartheta s\ i\ (s\ i) = \vartheta\ i$

$\wedge\ \vartheta s\ i\ (s\ (Suc\ i)) = \vartheta\ (Suc\ i))$
$(\lambda i.\ is\text{-}desc\text{-}fthread\ (\vartheta s\ i)\ p\ (s\ i)\ (s\ (Suc\ i))$
$\wedge\ \vartheta s\ i\ (s\ i) = \vartheta\ i$
$\wedge\ \vartheta s\ i\ (s\ (Suc\ i)) = \vartheta\ (Suc\ i))$
⟨*proof*⟩

lemma *INF-drop-prefix*:
$(\exists_\infty i{::}nat.\ i > n \wedge P\ i) = (\exists_\infty i.\ P\ i)$
⟨*proof*⟩

lemma *contract-keeps-threads*:
assumes *inc*[*simp*]: *increasing s*
shows $(\exists\,\vartheta.\ is\text{-}desc\text{-}thread\ \vartheta\ p)$
$\longleftrightarrow (\exists\,\vartheta.\ is\text{-}desc\text{-}thread\ \vartheta\ (contract\ s\ p))$
(**is** *?A* $\longleftrightarrow$ *?B*)
⟨*proof*⟩

lemma *repeated-edge*:
assumes $\bigwedge i.\ i > n \Longrightarrow dsc\ (snd\ (p\ i))\ k\ k$
shows *is-desc-thread* $(\lambda i.\ k)\ p$
⟨*proof*⟩

lemma *fin-from-inf*:
assumes *is-thread n* ϑ *p*
assumes $n < i$
assumes $i < j$
shows *is-fthread* ϑ *p i j*
⟨*proof*⟩

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* $\{x, y\} = (x, y)$
⟨*proof*⟩

definition
set2list = *inv set*

lemma *finite-set2list*:

assumes *finite S*
shows *set* (*set2list S*) = *S*
⟨*proof*⟩

corollary *RamseyNatpairs*:
fixes *S* :: *'a set*
and *f* :: *nat* × *nat* ⇒ *'a*

assumes *finite S*
and *inS*: ⋀*x y*. *x* < *y* ⟹ *f* (*x*, *y*) ∈ *S*

obtains *T* :: *nat set* **and** *s* :: *'a*
where *infinite T*
and *s* ∈ *S*
and ⋀*x y*. ⟦*x* ∈ *T*; *y* ∈ *T*; *x* < *y*⟧ ⟹ *f* (*x*, *y*) = *s*
⟨*proof*⟩

5.5 Main Result

theorem *LJA-Theorem4*:
assumes *finite-acg A*
shows *SCT A* ⟷ *SCT' A*
⟨*proof*⟩

end

6 Applying SCT to function definitions

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

definition
idseq R s x = (*s 0* = *x* ∧ (∀ *i*. *R* (*s* (*Suc i*)) (*s i*)))

lemma *not-acc-smaller*:
assumes *notacc*: ¬ *accp R x*
shows ∃ *y*. *R y x* ∧ ¬ *accp R y*
⟨*proof*⟩

lemma *non-acc-has-idseq*:
assumes ¬ *accp R x*
shows ∃ *s*. *idseq R s x*
⟨*proof*⟩

types $('a, 'q)\ cdesc =$
$('q \Rightarrow bool) \times ('q \Rightarrow 'a) \times ('q \Rightarrow 'a)$

fun *in-cdesc* $:: ('a, 'q)\ cdesc \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$
where
$in\text{-}cdesc\ (\Gamma, r, l)\ x\ y = (\exists\, q.\ x = r\ q \wedge y = l\ q \wedge \Gamma\ q)$

primrec *mk-rel* $:: ('a, 'q)\ cdesc\ list \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$
where
$mk\text{-}rel\ []\ x\ y = False$
$|\ mk\text{-}rel\ (c\#cs)\ x\ y =$
$(in\text{-}cdesc\ c\ x\ y \vee mk\text{-}rel\ cs\ x\ y)$

lemma *some-rd*:
assumes *mk-rel rds x y*
shows $\exists\, rd{\in}set\ rds.\ in\text{-}cdesc\ rd\ x\ y$
⟨*proof*⟩

lemma *ex-cs*:
assumes *idseq*: *idseq* (*mk-rel rds*) *s x*
shows $\exists\, cs.\ \forall\, i.\ cs\ i \in set\ rds \wedge in\text{-}cdesc\ (cs\ i)\ (s\ (Suc\ i))\ (s\ i)$
⟨*proof*⟩

types $'a\ measures = nat \Rightarrow 'a \Rightarrow nat$

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

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

definition
$decreq :: ('a, 'q)\ cdesc \Rightarrow ('a, 'q)\ cdesc \Rightarrow$
$('a \Rightarrow nat) \Rightarrow ('a \Rightarrow nat) \Rightarrow bool$

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 (λx. 0) (λx. 0) (λx 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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *no-inf-desc-nat-sequence*:
fixes *s* :: *nat ⇒ 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*⟩

definition
approx :: *nat scg ⇒ ('a, 'q) cdesc ⇒ ('a, 'q) cdesc*
⇒ 'a measures ⇒ 'a measures ⇒ bool
where
approx G C C' M M'
= (∀ i j. (dsc G i j ⟶ decr C C' (M i) (M' j))
∧(eqp G i j ⟶ decreq C C' (M i) (M' j)))

lemma *approx-empty*:
approx (Graph {}) c1 c2 ms1 ms2

⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *approx* (*Graph* {(*1*, ↓, *2*),(*2*, ⇓, *3*)}) *c1 c2 ms1 ms2*
⟨*proof*⟩

lemma *no-stepI*:
stepP c1 c2 m1 m2 (λ*x y*. *False*)
⟹ *no-step c1 c2*
⟨*proof*⟩

definition
sound-int :: *nat acg* ⇒ (′*a*, ′*q*) *cdesc list*
⇒ ′*a measures list* ⇒ *bool*
where
sound-int $\mathcal{A}$ *RDs M* =
(∀ *n*<*length RDs*. ∀ *m*<*length RDs*.
no-step (*RDs* ! *n*) (*RDs* ! *m*) ∨
(∃ *G*. ($\mathcal{A}$ ⊢ *n* ⇝^{G} *m*) ∧ *approx G* (*RDs* ! *n*) (*RDs* ! *m*) (*M* ! *n*) (*M* ! *m*)))

lemma *length-simps*: *length* [] = *0 length* (*x*#*xs*) = *Suc* (*length xs*)
⟨*proof*⟩

lemma *all-less-zero*: ∀ *n*<(*0*::*nat*). *P n*
⟨*proof*⟩

lemma *all-less-Suc*:
assumes *Pk*: *P k*
assumes *Pn*: ∀ *n*<*k*. *P n*
shows ∀ *n*<*Suc k*. *P n*
⟨*proof*⟩

lemma *step-witness*:
assumes *in-cdesc RD1 y x*
assumes *in-cdesc RD2 z y*
shows ¬ *no-step RD1 RD2*
⟨*proof*⟩

theorem *SCT-on-relations*:
assumes *R*: *R* = *mk-rel RDs*
assumes *sound*: *sound-int* $\mathcal{A}$ *RDs M*
assumes *SCT* $\mathcal{A}$
shows ∀ *x*. *accp R x*
⟨*proof*⟩

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* })
⟨*proof*⟩

lemma *mk-tcl-finite-terminates*:
fixes *A* :: '*a acg*
assumes *fA*: *finite-acg A*
shows *mk-tcl-dom* (*A*, *A*)
⟨*proof*⟩

lemma *mk-tcl-finite-tcl*:
fixes *A* :: '*a acg*

assumes *fA*: *finite-acg A*
shows *mk-tcl A A* = *tcl A*
⟨*proof*⟩

definition *test-SCT* :: *nat acg* ⇒ *bool*
where
test-SCT $\mathcal{A}$ =
(*let* $\mathcal{T}$ = *mk-tcl* $\mathcal{A}$ $\mathcal{A}$
in (∀(*n*,*G*,*m*)∈*dest-graph* $\mathcal{T}$.
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*
⟨*proof*⟩

code-modulename *SML*
Implementation Graphs

lemma [*code func*]:
(*G*::('*a*::*eq*, '*b*::*eq*) *graph*) ≤ *H* ⟷ *dest-graph G* ⊆ *dest-graph H*
(*G*::('*a*::*eq*, '*b*::*eq*) *graph*) < *H* ⟷ *dest-graph G* ⊂ *dest-graph H*
⟨*proof*⟩

lemma [*code func*]:
(*G*::('*a*::*eq*, '*b*::*eq*) *graph*) + *H* = *Graph* (*dest-graph G* ∪ *dest-graph H*)
⟨*proof*⟩

lemma [*code func*]:
(*G*::('*a*::*eq*, '*b*::{*eq*, *times*}) *graph*) ∗ *H* = *grcomp G H*
⟨*proof*⟩

lemma *SCT′-empty*: *SCT′* (*Graph* {})
⟨*proof*⟩

7.1 Witness checking

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

lemma *no-bad-graphs-ucl*:
assumes $A \leq B$
assumes *no-bad-graphs B*
shows *no-bad-graphs A*
⟨*proof*⟩

lemma *SCT'-witness*:
assumes *a*: *test-SCT-witness A T*
shows *SCT' A*
⟨*proof*⟩

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

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*:
$\{x \in \{\}.\ P\ x\} = \{\}$
$\{x \in insert\ y\ ys.\ P\ x\} = (if\ P\ y\ then\ insert\ y\ \{x \in ys.\ P\ x\}\ else\ \{x \in ys.\ P\ x\})$
⟨*proof*⟩

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

lemma *cartprod-simps*:
$\{\} \times A = \{\}$
$insert\ a\ A \times B = Pair\ a\ `\ B \cup (A \times B)$
⟨*proof*⟩

lemma *image-simps*:
fu ‘ {} = {}
fu ‘ *insert a A* = *insert* (*fu a*) (*fu* ‘ *A*)
⟨*proof*⟩

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

lemma *subset-simps*:
{} ⊆ *B*
insert a A ⊆ *B* ≡ *a* ∈ *B* ∧ *A* ⊆ *B*
⟨*proof*⟩

lemma *element-simps*:
x ∈ {} ≡ *False*
x ∈ *insert a A* ≡ *x* = *a* ∨ *x* ∈ *A*
⟨*proof*⟩

lemma *set-eq-simp*:
A = *B* ⟷ *A* ⊆ *B* ∧ *B* ⊆ *A* ⟨*proof*⟩

lemma *ball-simps*:
∀ *x*∈{}. *P x* ≡ *True*
(∀ *x*∈*insert a A*. *P x*) ≡ *P a* ∧ (∀ *x*∈*A*. *P x*)
⟨*proof*⟩

lemma *bex-simps*:
∃ *x*∈{}. *P x* ≡ *False*
(∃ *x*∈*insert a A*. *P x*) ≡ *P a* ∨ (∃ *x*∈*A*. *P x*)
⟨*proof*⟩

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*:
↓ ∗ *x* = ↓
⇓ ∗ *x* = *x*
⟨*proof*⟩

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*
⟨*proof*⟩

end

9 Examples for Size-Change Termination

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

function $f :: nat \Rightarrow nat \Rightarrow nat$
where
$f\ n\ 0 = n$
$|\ f\ 0\ (Suc\ m) = f\ (Suc\ m)\ m$
$|\ f\ (Suc\ n)\ (Suc\ m) = f\ m\ n$
⟨*proof*⟩

termination

⟨*proof*⟩

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)
⟨*proof*⟩

termination
⟨*proof*⟩

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*
⟨*proof*⟩

termination
⟨*proof*⟩

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

termination
⟨*proof*⟩

end