Finfuns

Andreas Lochbihler

March 17, 2025

Contents

1 Almost everywhere constant functions

theory *FinFun*
imports *HOL−Library.Cardinality*
begin

This theory defines functions which are constant except for finitely many points (FinFun) and introduces a type finfin along with a number of operators for them. The code generator is set up such that such functions can be represented as data in the generated code and all operators are executable.

For details, see Formalising FinFuns - Generating Code for Functions as Data by A. Lochbihler in TPHOLs 2009.

1.1 The *map-default* operation

definition *map-default* :: $'b \Rightarrow ('a \rightharpoonup 'b) \Rightarrow 'a \Rightarrow 'b$
where *map-default b f a* $\equiv$ *case f a of None* $\Rightarrow$ *b* | *Some b'* $\Rightarrow$ *b'*

lemma *map-default-delete* [*simp*]:
  map-default b (*f*(*a* := *None*)) = (*map-default b f*)(*a* := *b*)
by(*simp add*: *map-default-def fun-eq-iff*)

lemma *map-default-insert*:
  map-default b (*f*(*a* $\mapsto$ *b'*)) = (*map-default b f*)(*a* := *b'*)
by(*simp add*: *map-default-def fun-eq-iff*)

lemma *map-default-empty* [*simp*]: *map-default b Map.empty* = (λa. *b*)
by(*simp add*: *fun-eq-iff map-default-def*)

lemma *map-default-inject*:
  fixes *g g'* :: $'a \rightharpoonup 'b$
  assumes *infin-eq*: $\neg$ *finite* (*UNIV* :: *'a set*) $\vee$ *b* = *b'*
  and *fin*: *finite* (*dom g*) **and** *b*: *b* $\notin$ *ran g*
  and *fin'*: *finite* (*dom g'*) **and** *b'*: *b'* $\notin$ *ran g'*
  and *eq'*: *map-default b g* = *map-default b' g'*
  shows *b* = *b'* *g* = *g'*
proof –
  from *infin-eq* **show** *bb'*: *b* = *b'*
  proof
    assume *infin*: $\neg$ *finite* (*UNIV* :: *'a set*)
    from *fin fin'* **have** *finite* (*dom g* $\cup$ *dom g'*) **by** *auto*
    with *infin* **have** *UNIV* – (*dom g* $\cup$ *dom g'*) $\neq$ {} **by**(*auto dest*: *finite-subset*)
    then obtain *a* **where** *a*: *a* $\notin$ *dom g* $\cup$ *dom g'* **by** *auto*
     hence *map-default b g a* = *b* *map-default b' g' a* = *b'* **by**(*auto simp add*: *map-default-def*)
    with *eq'* **show** *b* = *b'* **by** *simp*
  qed

  show *g* = *g'*
  proof
    fix *x*
    show *g x* = *g' x*
    proof(*cases g x*)
      case *None*

hence *map-default b g x = b* **by**(*simp add*: *map-default-def*)
with *bb′ eq′* **have** *map-default b′ g′ x = b′* **by** *simp*
with *b′* **have** *g′ x = None* **by**(*simp add*: *map-default-def ran-def split*: *option.split-asm*)
with *None* **show** *?thesis* **by** *simp*
next
case (*Some c*)
with *b* **have** *cb*: $c \neq b$ **by**(*auto simp add*: *ran-def*)
moreover from *Some* **have** *map-default b g x = c* **by**(*simp add*: *map-default-def*)
with *eq′* **have** *map-default b′ g′ x = c* **by** *simp*
ultimately have *g′ x = Some c* **using** *b′ bb′* **by**(*auto simp add*: *map-default-def split*: *option.splits*)
with *Some* **show** *?thesis* **by** *simp*
qed
qed
qed

1.2 The finfun type

definition *finfun* = $\{f::'a \Rightarrow 'b.\ \exists b.\ finite\ \{a.\ f\ a \neq b\}\}$

typedef (′*a*,′*b*) *finfun* (‹(- ⇒f /-)› [*22*, *21*] *21*) = *finfun* :: (′*a* => ′*b*) *set*
morphisms *finfun-apply Abs-finfun*
proof −
have $\exists f.\ finite\ \{x.\ f\ x \neq undefined\}$
proof
show $finite\ \{x.\ (\lambda y.\ undefined)\ x \neq undefined\}$ **by** *auto*
qed
then show *?thesis* **unfolding** *finfun-def* **by** *auto*
qed

type-notation *finfun* (‹(- ⇒f /-)› [*22*, *21*] *21*)

setup-lifting *type-definition-finfun*

lemma *fun-upd-finfun*: $y(a := b) \in finfun \longleftrightarrow y \in finfun$
proof −
{ fix *b′*
have $finite\ \{a'.\ (y(a := b))\ a' \neq b'\} = finite\ \{a'.\ y\ a' \neq b'\}$
proof(*cases* $b = b'$)
case *True*
hence $\{a'.\ (y(a := b))\ a' \neq b'\} = \{a'.\ y\ a' \neq b'\} - \{a\}$ **by** *auto*
thus *?thesis* **by** *simp*
next
case *False*
hence $\{a'.\ (y(a := b))\ a' \neq b'\} = insert\ a\ \{a'.\ y\ a' \neq b'\}$ **by** *auto*
thus *?thesis* **by** *simp*
qed }
thus *?thesis* **unfolding** *finfun-def* **by** *blast*

```
qed

lemma const-finfun: (λx. a) ∈ finfun
by(auto simp add: finfun-def)

lemma finfun-left-compose:
  assumes y ∈ finfun
  shows g ∘ y ∈ finfun
proof −
  from assms obtain b where finite {a. y a ≠ b}
    unfolding finfun-def by blast
  hence finite {c. g (y c) ≠ g b}
  proof(induct {a. y a ≠ b} arbitrary: y)
    case empty
    hence y = (λa. b) by(auto)
    thus ?case by(simp)
  next
    case (insert x F)
    note IH = ‹⋀y. F = {a. y a ≠ b} ⟹ finite {c. g (y c) ≠ g b}›
    from ‹insert x F = {a. y a ≠ b}› ‹x ∉ F›
    have F: F = {a. (y(x := b)) a ≠ b} by(auto)
    show ?case
    proof(cases g (y x) = g b)
      case True
      hence {c. g ((y(x := b)) c) ≠ g b} = {c. g (y c) ≠ g b} by auto
      with IH[OF F] show ?thesis by simp
    next
      case False
      hence {c. g (y c) ≠ g b} = insert x {c. g ((y(x := b)) c) ≠ g b} by auto
      with IH[OF F] show ?thesis by(simp)
    qed
  qed
  thus ?thesis unfolding finfun-def by auto
qed

lemma assumes y ∈ finfun
  shows fst-finfun: fst ∘ y ∈ finfun
  and snd-finfun: snd ∘ y ∈ finfun
proof −
  from assms obtain b c where bc: finite {a. y a ≠ (b, c)}
    unfolding finfun-def by auto
  have {a. fst (y a) ≠ b} ⊆ {a. y a ≠ (b, c)}
    and {a. snd (y a) ≠ c} ⊆ {a. y a ≠ (b, c)} by auto
  hence finite {a. fst (y a) ≠ b}
    and finite {a. snd (y a) ≠ c} using bc by(auto intro: finite-subset)
  thus fst ∘ y ∈ finfun snd ∘ y ∈ finfun
    unfolding finfun-def by auto
qed
```

lemma *map-of-finfun*: *map-of xs* ∈ *finfun*
unfolding *finfun-def*
by(*induct xs*)(*auto simp add*: *Collect-neg-eq Collect-conj-eq Collect-imp-eq intro*: *finite-subset*)

lemma *Diag-finfun*: (λ*x*. (*f x*, *g x*)) ∈ *finfun* ⟷ *f* ∈ *finfun* ∧ *g* ∈ *finfun*
by(*auto intro*: *finite-subset simp add*: *Collect-neg-eq Collect-imp-eq Collect-conj-eq finfun-def*)

lemma *finfun-right-compose*:
  assumes *g*: *g* ∈ *finfun* **and** *inj*: *inj f*
  shows *g o f* ∈ *finfun*
proof −
  from *g* **obtain** *b* **where** *b*: *finite* {*a*. *g a* ≠ *b*} **unfolding** *finfun-def* **by** *blast*
  moreover have *f* ‘ {*a*. *g* (*f a*) ≠ *b*} ⊆ {*a*. *g a* ≠ *b*} **by** *auto*
  moreover from *inj* **have** *inj-on f* {*a*. *g* (*f a*) ≠ *b*} **by**(*rule subset-inj-on*) *blast*
  ultimately have *finite* {*a*. *g* (*f a*) ≠ *b*}
    by(*blast intro*: *finite-imageD*[**where** *f*=*f*] *finite-subset*)
  thus *?thesis* **unfolding** *finfun-def* **by** *auto*
qed

lemma *finfun-curry*:
  assumes *fin*: *f* ∈ *finfun*
  shows *curry f* ∈ *finfun curry f a* ∈ *finfun*
proof −
  from *fin* **obtain** *c* **where** *c*: *finite* {*ab*. *f ab* ≠ *c*} **unfolding** *finfun-def* **by** *blast*
  moreover have {*a*. ∃ *b*. *f* (*a*, *b*) ≠ *c*} = *fst* ‘ {*ab*. *f ab* ≠ *c*} **by**(*force*)
  hence {*a*. *curry f a* ≠ (λ*b*. *c*)} = *fst* ‘ {*ab*. *f ab* ≠ *c*}
    by(*auto simp add*: *curry-def fun-eq-iff*)
  ultimately have *finite* {*a*. *curry f a* ≠ (λ*b*. *c*)} **by** *simp*
  thus *curry f* ∈ *finfun* **unfolding** *finfun-def* **by** *blast*

  have *snd* ‘ {*ab*. *f ab* ≠ *c*} = {*b*. ∃ *a*. *f* (*a*, *b*) ≠ *c*} **by**(*force*)
  hence {*b*. *f* (*a*, *b*) ≠ *c*} ⊆ *snd* ‘ {*ab*. *f ab* ≠ *c*} **by** *auto*
  hence *finite* {*b*. *f* (*a*, *b*) ≠ *c*} **by**(*rule finite-subset*)(*rule finite-imageI*[*OF c*])
  thus *curry f a* ∈ *finfun* **unfolding** *finfun-def* **by** *auto*
qed

bundle *finfun*
begin

lemmas [*simp*] =
  fst-finfun snd-finfun Abs-finfun-inverse
  finfun-apply-inverse Abs-finfun-inject finfun-apply-inject
  Diag-finfun finfun-curry
lemmas [*iff*] =
  const-finfun fun-upd-finfun finfun-apply map-of-finfun
lemmas [*intro*] =
  finfun-left-compose fst-finfun snd-finfun

```
end

lemma Abs-finfun-inject-finite:
  fixes x y :: 'a ⇒ 'b
  assumes fin: finite (UNIV :: 'a set)
  shows Abs-finfun x = Abs-finfun y ⟷ x = y
proof
  assume Abs-finfun x = Abs-finfun y
  moreover have x ∈ finfun y ∈ finfun unfolding finfun-def
    by(auto intro: finite-subset[OF - fin])
  ultimately show x = y by(simp add: Abs-finfun-inject)
qed simp

lemma Abs-finfun-inject-finite-class:
  fixes x y :: ('a :: finite) ⇒ 'b
  shows Abs-finfun x = Abs-finfun y ⟷ x = y
using finite-UNIV
by(simp add: Abs-finfun-inject-finite)

lemma Abs-finfun-inj-finite:
  assumes fin: finite (UNIV :: 'a set)
  shows inj (Abs-finfun :: ('a ⇒ 'b) ⇒ 'a ⇒f 'b)
proof(rule inj-onI)
  fix x y :: 'a ⇒ 'b
  assume Abs-finfun x = Abs-finfun y
  moreover have x ∈ finfun y ∈ finfun unfolding finfun-def
    by(auto intro: finite-subset[OF - fin])
  ultimately show x = y by(simp add: Abs-finfun-inject)
qed

lemma Abs-finfun-inverse-finite:
  fixes x :: 'a ⇒ 'b
  assumes fin: finite (UNIV :: 'a set)
  shows finfun-apply (Abs-finfun x) = x
  including finfun
proof −
  from fin have x ∈ finfun
    by(auto simp add: finfun-def intro: finite-subset)
  thus ?thesis by simp
qed

lemma Abs-finfun-inverse-finite-class:
  fixes x :: ('a :: finite) ⇒ 'b
  shows finfun-apply (Abs-finfun x) = x
using finite-UNIV by(simp add: Abs-finfun-inverse-finite)

lemma finfun-eq-finite-UNIV: finite (UNIV :: 'a set) ⟹ (finfun :: ('a ⇒ 'b) set)
= UNIV
```

unfolding *finfun-def* **by**(*auto intro*: *finite-subset*)

lemma *finfun-finite-UNIV-class*: *finfun* = (*UNIV* :: ('*a* :: *finite* ⇒ '*b*) *set*)
by(*simp add*: *finfun-eq-finite-UNIV*)

lemma *map-default-in-finfun*:
  assumes *fin*: *finite* (*dom f*)
  shows *map-default b f* ∈ *finfun*
unfolding *finfun-def*
proof(*intro CollectI exI*)
  from *fin* **show** *finite* {*a*. *map-default b f a* ≠ *b*}
    by(*auto simp add*: *map-default-def dom-def Collect-conj-eq split*: *option.splits*)
qed

lemma *finfun-cases-map-default*:
  obtains *b g* **where** *f* = *Abs-finfun* (*map-default b g*) *finite* (*dom g*) *b* ∉ *ran g*
proof −
  obtain *y* **where** *f*: *f* = *Abs-finfun y* **and** *y*: *y* ∈ *finfun* **by**(*cases f*)
  from *y* **obtain** *b* **where** *b*: *finite* {*a*. *y a* ≠ *b*} **unfolding** *finfun-def* **by** *auto*
  let *?g* = (λ*a*. *if y a* = *b then None else Some* (*y a*))
  have *map-default b ?g* = *y* **by**(*simp add*: *fun-eq-iff map-default-def*)
  with *f* **have** *f* = *Abs-finfun* (*map-default b ?g*) **by** *simp*
  moreover from *b* **have** *finite* (*dom ?g*) **by**(*auto simp add*: *dom-def*)
  moreover have *b* ∉ *ran ?g* **by**(*auto simp add*: *ran-def*)
  ultimately show *?thesis* **by**(*rule that*)
qed

1.3 Kernel functions for type '*a* ⇒*f* '*b*

lift-definition *finfun-const* :: '*b* ⇒ '*a* ⇒*f* '*b* (‹*K*$/ -› [*0*] *1*)
is λ *b x*. *b* **by** (*rule const-finfun*)

lift-definition *finfun-update* :: '*a* ⇒*f* '*b* ⇒ '*a* ⇒ '*b* ⇒ '*a* ⇒*f* '*b* (‹-'(- $:= -')›
[*1000*,*0*,*0*] *1000*) **is** *fun-upd*
by (*simp add*: *fun-upd-finfun*)

lemma *finfun-update-twist*: *a* ≠ *a*' ⟹ *f*(*a* $:= *b*)(*a*' $:= *b*') = *f*(*a*' $:= *b*')(*a* $:=
b)
by *transfer* (*simp add*: *fun-upd-twist*)

lemma *finfun-update-twice* [*simp*]:
  f(*a* $:= *b*)(*a* $:= *b*') = *f*(*a* $:= *b*')
by *transfer simp*

lemma *finfun-update-const-same*: (*K*$ *b*)(*a* $:= *b*) = (*K*$ *b*)
by *transfer* (*simp add*: *fun-eq-iff*)

1.4 Code generator setup

definition *finfun-update-code* :: '*a* ⇒*f* '*b* ⇒ '*a* ⇒ '*b* ⇒ '*a* ⇒*f* '*b*

where [*simp*, *code del*]: *finfun-update-code* = *finfun-update*

code-datatype *finfun-const finfun-update-code*

lemma *finfun-update-const-code* [*code*]:
$(K\$\ b)(a\ \${:=}\ b') = (\textit{if}\ b = b'\ \textit{then}\ (K\$\ b)\ \textit{else finfun-update-code}\ (K\$\ b)\ a\ b')$
by(*simp add*: *finfun-update-const-same*)

lemma *finfun-update-update-code* [*code*]:
$(\textit{finfun-update-code}\ f\ a\ b)(a'\ \${:=}\ b') = (\textit{if}\ a = a'\ \textit{then}\ f(a\ \${:=}\ b')\ \textit{else finfun-update-code}\ (f(a'\ \${:=}\ b'))\ a\ b)$
by(*simp add*: *finfun-update-twist*)

1.5 Setup for quickcheck

quickcheck-generator *finfun constructors*: *finfun-update-code*, *finfun-const* :: $'b \Rightarrow 'a \Rightarrow_f 'b$

1.6 *finfun-update* as instance of *comp-fun-commute*

interpretation *finfun-update*: *comp-fun-commute* $\lambda a\ f.\ f(a :: 'a\ \${:=}\ b')$
including *finfun*
proof
fix $a\ a' :: 'a$
show $(\lambda f.\ f(a\ \${:=}\ b')) \circ (\lambda f.\ f(a'\ \${:=}\ b')) = (\lambda f.\ f(a'\ \${:=}\ b')) \circ (\lambda f.\ f(a\ \${:=}\ b'))$
proof
fix b
have $(\textit{finfun-apply}\ b)(a := b',\ a' := b') = (\textit{finfun-apply}\ b)(a' := b',\ a := b')$
by(*cases* $a = a'$)(*auto simp add*: *fun-upd-twist*)
then have $b(a\ \${:=}\ b')(a'\ \${:=}\ b') = b(a'\ \${:=}\ b')(a\ \${:=}\ b')$
by(*auto simp add*: *finfun-update-def fun-upd-twist*)
then show $((\lambda f.\ f(a\ \${:=}\ b')) \circ (\lambda f.\ f(a'\ \${:=}\ b')))\ b = ((\lambda f.\ f(a'\ \${:=}\ b')) \circ (\lambda f.\ f(a\ \${:=}\ b')))\ b$
by (*simp add*: *fun-eq-iff*)
qed
qed

lemma *fold-finfun-update-finite-univ*:
assumes *fin*: *finite* (*UNIV* :: *'a set*)
shows *Finite-Set.fold* $(\lambda a\ f.\ f(a\ \${:=}\ b'))\ (K\$\ b)\ (\textit{UNIV} :: 'a\ \textit{set}) = (K\$\ b')$
proof −
{ fix $A :: 'a\ \textit{set}$
from *fin* **have** *finite A* **by**(*auto intro*: *finite-subset*)
hence *Finite-Set.fold* $(\lambda a\ f.\ f(a\ \${:=}\ b'))\ (K\$\ b)\ A = \textit{Abs-finfun}\ (\lambda a.\ \textit{if}\ a \in A\ \textit{then}\ b'\ \textit{else}\ b)$
proof(*induct*)
case (*insert x F*)
have $(\lambda a.\ \textit{if}\ a = x\ \textit{then}\ b'\ \textit{else}\ (\textit{if}\ a \in F\ \textit{then}\ b'\ \textit{else}\ b)) = (\lambda a.\ \textit{if}\ a = x \vee a \in F\ \textit{then}\ b'\ \textit{else}\ b)$

by(*auto*)
with *insert* **show** *?case*
by(*simp add*: *finfun-const-def fun-upd-def*)(*simp add*: *finfun-update-def Abs-finfun-inverse-finite*[*OF fin*] *fun-upd-def*)
qed(*simp add*: *finfun-const-def*) **}**
thus *?thesis* **by**(*simp add*: *finfun-const-def*)
qed

1.7 Default value for FinFuns

definition *finfun-default-aux* :: $('a \Rightarrow 'b) \Rightarrow 'b$
where [*code del*]: *finfun-default-aux f* = (*if finite* (*UNIV* :: *'a set*) *then undefined else THE b. finite* {*a. f a* ≠ *b*})

lemma *finfun-default-aux-infinite*:
fixes *f* :: $'a \Rightarrow 'b$
assumes *infin*: ¬ *finite* (*UNIV* :: *'a set*)
and *fin*: *finite* {*a. f a* ≠ *b*}
shows *finfun-default-aux f* = *b*
proof −
let *?B* = {*a. f a* ≠ *b*}
from *fin* **have** (*THE b. finite* {*a. f a* ≠ *b*}) = *b*
proof(*rule the-equality*)
fix *b'*
assume *finite* {*a. f a* ≠ *b'*} (**is** *finite ?B'*)
with *infin fin* **have** *UNIV* − (*?B'* ∪ *?B*) ≠ {} **by**(*auto dest*: *finite-subset*)
then obtain *a* **where** *a*: *a* ∉ *?B'* ∪ *?B* **by** *auto*
thus *b'* = *b* **by** *auto*
qed
thus *?thesis* **using** *infin* **by**(*simp add*: *finfun-default-aux-def*)
qed

lemma *finite-finfun-default-aux*:
fixes *f* :: $'a \Rightarrow 'b$
assumes *fin*: *f* ∈ *finfun*
shows *finite* {*a. f a* ≠ *finfun-default-aux f*}
proof(*cases finite* (*UNIV* :: *'a set*))
case *True* **thus** *?thesis* **using** *fin*
by(*auto simp add*: *finfun-def finfun-default-aux-def intro*: *finite-subset*)
next
case *False*
from *fin* **obtain** *b* **where** *b*: *finite* {*a. f a* ≠ *b*} (**is** *finite ?B*)
unfolding *finfun-def* **by** *blast*
with *False* **show** *?thesis* **by**(*simp add*: *finfun-default-aux-infinite*)
qed

lemma *finfun-default-aux-update-const*:
fixes *f* :: $'a \Rightarrow 'b$

assumes *fin*: $f \in$ *finfun*
shows *finfun-default-aux* $(f(a := b))$ = *finfun-default-aux f*
proof(*cases finite* (*UNIV* :: *'a set*))
case *False*
from *fin* **obtain** b' **where** b': *finite* $\{a.\ f\ a \neq b'\}$ **unfolding** *finfun-def* **by** *blast*
hence *finite* $\{a'.\ (f(a := b))\ a' \neq b'\}$
proof(*cases* $b = b' \wedge f\ a \neq b'$)
case *True*
hence $\{a.\ f\ a \neq b'\}$ = *insert a* $\{a'.\ (f(a := b))\ a' \neq b'\}$ **by** *auto*
thus *?thesis* **using** b' **by** *simp*
next
case *False*
moreover
{ assume $b \neq b'$
hence $\{a'.\ (f(a := b))\ a' \neq b'\}$ = *insert a* $\{a.\ f\ a \neq b'\}$ **by** *auto*
hence *?thesis* **using** b' **by** *simp* **}**
moreover
{ assume $b = b'\ f\ a = b'$
hence $\{a'.\ (f(a := b))\ a' \neq b'\} = \{a.\ f\ a \neq b'\}$ **by** *auto*
hence *?thesis* **using** b' **by** *simp* **}**
ultimately show *?thesis* **by** *blast*
qed
with *False* b' **show** *?thesis* **by**(*auto simp del*: *fun-upd-apply simp add*: *finfun-default-aux-infinite*)
next
case *True* **thus** *?thesis* **by**(*simp add*: *finfun-default-aux-def*)
qed

lift-definition *finfun-default* :: *'a* ⇒*f* *'b* ⇒ *'b*
is *finfun-default-aux* **.**

lemma *finite-finfun-default*: *finite* $\{a.$ *finfun-apply f a* $\neq$ *finfun-default f*$\}$
by *transfer* (*erule finite-finfun-default-aux*)

lemma *finfun-default-const*: *finfun-default* ((*K$ b*) :: *'a* ⇒*f* *'b*) = (*if finite* (*UNIV* :: *'a set*) *then undefined else b*)
by(*transfer*)(*auto simp add*: *finfun-default-aux-infinite finfun-default-aux-def*)

lemma *finfun-default-update-const*:
finfun-default (*f*(*a $:= b*)) = *finfun-default f*
by *transfer* (*simp add*: *finfun-default-aux-update-const*)

lemma *finfun-default-const-code* [*code*]:
finfun-default ((*K$ c*) :: *'a* :: *card-UNIV* ⇒*f* *'b*) = (*if CARD*(*'a*) = *0 then c else undefined*)
by(*simp add*: *finfun-default-const*)

lemma *finfun-default-update-code* [*code*]:
finfun-default (*finfun-update-code f a b*) = *finfun-default f*

by(*simp add*: *finfun-default-update-const*)

1.8 Recursion combinator and well-formedness conditions

definition *finfun-rec* :: $('b \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'c) \Rightarrow ('a \Rightarrow f\ 'b) \Rightarrow 'c$
where [*code del*]:
  finfun-rec cnst upd f ≡
   let b = finfun-default f;
     g = THE g. f = Abs-finfun (map-default b g) ∧ *finite (dom g)* ∧ *b* ∉ *ran g*
   in Finite-Set.fold (λ*a. upd a (map-default b g a)*) (*cnst b*) (*dom g*)

locale *finfun-rec-wf-aux* =
  fixes *cnst* :: $'b \Rightarrow 'c$
  and *upd* :: $'a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'c$
  assumes *upd-const-same*: *upd a b (cnst b) = cnst b*
  and *upd-commute*: $a \neq a' \Longrightarrow$ *upd a b (upd a' b' c) = upd a' b' (upd a b c)*
  and *upd-idemp*: $b \neq b' \Longrightarrow$ *upd a b'' (upd a b' (cnst b)) = upd a b'' (cnst b)*
begin

lemma *upd-left-comm*: *comp-fun-commute* (λ*a. upd a (f a)*)
by(*unfold-locales*)(*auto intro*: *upd-commute simp add*: *fun-eq-iff*)

lemma *upd-upd-twice*: *upd a b'' (upd a b' (cnst b)) = upd a b'' (cnst b)*
by(*cases b* ≠ *b'*)(*auto simp add*: *fun-upd-def upd-const-same upd-idemp*)

lemma *map-default-update-const*:
  assumes *fin*: *finite (dom f)*
  and *anf*: *a* ∉ *dom f*
  and *fg*: $f \subseteq_m g$
  shows *upd a d* (*Finite-Set.fold* (λ*a. upd a (map-default d g a)*) (*cnst d*) (*dom f*)) =
      Finite-Set.fold (λ*a. upd a (map-default d g a)*) (*cnst d*) (*dom f*)
proof −
  let *?upd* = λ*a. upd a (map-default d g a)*
  let *?fr* = λ*A. Finite-Set.fold ?upd (cnst d) A*
  interpret *gwf*: *comp-fun-commute ?upd* **by**(*rule upd-left-comm*)

  from *fin anf fg* **show** *?thesis*
  proof(*induct dom f arbitrary*: *f*)
    case *empty*
    from ‹{} = *dom f*› **have** *f = Map.empty* **by**(*auto simp add*: *dom-def*)
    thus *?case* **by**(*simp add*: *finfun-const-def upd-const-same*)
  next
    case (*insert a' A*)
    note *IH* = ‹⋀*f*. ⟦ *A = dom f*; *a* ∉ *dom f*; $f \subseteq_m g$ ⟧ ⟹ *upd a d (?fr (dom f)) = ?fr (dom f)*›
    note *fin* = ‹*finite A*› **note** *anf* = ‹*a* ∉ *dom f*› **note** *a'nA* = ‹*a'* ∉ *A*›
    note *domf* = ‹*insert a' A = dom f*› **note** *fg* = ‹$f \subseteq_m g$›

from *domf* **obtain** *b* **where** *b*: *f a'* = *Some b* **by** *auto*
let *?f'* = *f*(*a'* := *None*)
have *upd a d* (*?fr* (*insert a' A*)) = *upd a d* (*upd a'* (*map-default d g a'*) (*?fr A*))
by(*subst gwf.fold-insert*[*OF fin a'nA*]) *rule*
also from *b fg* **have** *g a'* = *f a'* **by**(*auto simp add*: *map-le-def intro*: *domI dest*: *bspec*)
hence *ga'*: *map-default d g a'* = *map-default d f a'* **by**(*simp add*: *map-default-def*)
also from *anf domf* **have** $a \neq a'$ **by** *auto* **note** *upd-commute*[*OF this*]
also from *domf a'nA anf fg* **have** $a \notin$ *dom ?f'* *?f'* $\subseteq_m$ *g* **and** *A*: *A* = *dom ?f'* **by**(*auto simp add*: *ran-def map-le-def*)
note *A* **also note** *IH*[*OF A* ‹$a \notin$ *dom ?f'*› ‹*?f'* $\subseteq_m$ *g*›]
also have *upd a'* (*map-default d f a'*) (*?fr* (*dom* (*f*(*a'* := *None*)))) = *?fr* (*dom f*)
unfolding *domf*[*symmetric*] *gwf.fold-insert*[*OF fin a'nA*] *ga'* **unfolding** *A* **..**
also have *insert a'* (*dom ?f'*) = *dom f* **using** *domf* **by** *auto*
finally show *?case* **.**
qed
qed

lemma *map-default-update-twice*:
assumes *fin*: *finite* (*dom f*)
and *anf*: $a \notin$ *dom f*
and *fg*: *f* $\subseteq_m$ *g*
shows *upd a d''* (*upd a d'* (*Finite-Set.fold* (λ*a. upd a* (*map-default d g a*)) (*cnst d*) (*dom f*))) =
upd a d'' (*Finite-Set.fold* (λ*a. upd a* (*map-default d g a*)) (*cnst d*) (*dom f*))
proof −
let *?upd* = λ*a. upd a* (*map-default d g a*)
let *?fr* = λ*A. Finite-Set.fold ?upd* (*cnst d*) *A*
interpret *gwf*: *comp-fun-commute ?upd* **by**(*rule upd-left-comm*)

from *fin anf fg* **show** *?thesis*
proof(*induct dom f arbitrary*: *f*)
case *empty*
from ‹{} = *dom f*› **have** *f* = *Map.empty* **by**(*auto simp add*: *dom-def*)
thus *?case* **by**(*auto simp add*: *finfun-const-def finfun-update-def upd-upd-twice*)
next
case (*insert a' A*)
note *IH* = ‹$\bigwedge$*f*. ⟦*A* = *dom f*; $a \notin$ *dom f*; *f* $\subseteq_m$ *g*⟧ $\Longrightarrow$ *upd a d''* (*upd a d'* (*?fr* (*dom f*))) = *upd a d''* (*?fr* (*dom f*))›
note *fin* = ‹*finite A*› **note** *anf* = ‹$a \notin$ *dom f*› **note** *a'nA* = ‹$a' \notin A$›
note *domf* = ‹*insert a' A* = *dom f*› **note** *fg* = ‹*f* $\subseteq_m$ *g*›

from *domf* **obtain** *b* **where** *b*: *f a'* = *Some b* **by** *auto*
let *?f'* = *f*(*a'* := *None*)
let *?b'* = *case f a' of None* $\Rightarrow$ *d* | *Some b* $\Rightarrow$ *b*
from *domf* **have** *upd a d''* (*upd a d'* (*?fr* (*dom f*))) = *upd a d''* (*upd a d'* (*?fr*

(*insert* a' *A*))) **by** *simp*
  also note *gwf.fold-insert*[*OF fin* $a'nA$]
  also from *b fg* **have** $g\ a' = f\ a'$ **by**(*auto simp add*: *map-le-def intro*: *domI dest*: *bspec*)
  hence ga': *map-default d g* a' = *map-default d f* a' **by**(*simp add*: *map-default-def*)
  also from *anf domf* **have** ana': $a \neq a'$ **by** *auto* **note** *upd-commute*[*OF this*]
  also note *upd-commute*[*OF* ana']
  also from *domf* $a'nA$ *anf fg* **have** $a \notin dom\ ?f'\ ?f' \subseteq_m g$ **and** *A*: $A = dom\ ?f'$ **by**(*auto simp add*: *ran-def map-le-def*)
  note *A* **also note** *IH*[*OF A* ‹$a \notin dom\ ?f'$› ‹$?f' \subseteq_m g$›]
  also note *upd-commute*[*OF* ana'[*symmetric*]] **also note** ga'[*symmetric*] **also note** *A*[*symmetric*]
  also note *gwf.fold-insert*[*symmetric*, *OF fin* $a'nA$] **also note** *domf*
  finally show *?case* **.**
 qed
qed

lemma *map-default-eq-id* [*simp*]: *map-default d* ((λa. *Some* (*f a*)) |‘ $\{a.\ f\ a \neq d\}$) = *f*
by(*auto simp add*: *map-default-def restrict-map-def*)

lemma *finite-rec-cong1*:
 assumes *f*: *comp-fun-commute f* **and** *g*: *comp-fun-commute g*
 and *fin*: *finite A*
 and *eq*: $\bigwedge a.\ a \in A \Longrightarrow f\ a = g\ a$
 shows *Finite-Set.fold f z A* = *Finite-Set.fold g z A*
proof −
 interpret *f*: *comp-fun-commute f* **by**(*rule f*)
 interpret *g*: *comp-fun-commute g* **by**(*rule g*)
 { fix *B*
  assume *BsubA*: $B \subseteq A$
  with *fin* **have** *finite B* **by**(*blast intro*: *finite-subset*)
  hence $B \subseteq A \Longrightarrow$ *Finite-Set.fold f z B* = *Finite-Set.fold g z B*
  proof(*induct*)
   case *empty* **thus** *?case* **by** *simp*
  next
   case (*insert a B*)
   note *finB* = ‹*finite B*› **note** *anB* = ‹$a \notin B$› **note** *sub* = ‹*insert a B* $\subseteq A$›
   note *IH* = ‹$B \subseteq A \Longrightarrow$ *Finite-Set.fold f z B* = *Finite-Set.fold g z B*›
   from *sub anB* **have** *BpsubA*: $B \subset A$ **and** *BsubA*: $B \subseteq A$ **and** *aA*: $a \in A$ **by** *auto*
   from *IH*[*OF BsubA*] *eq*[*OF aA*] *finB anB*
   show *?case* **by**(*auto*)
  qed
  with *BsubA* **have** *Finite-Set.fold f z B* = *Finite-Set.fold g z B* **by** *blast* **}**
 thus *?thesis* **by** *blast*
qed

lemma *finfun-rec-upd* [*simp*]:

```
  finfun-rec cnst upd (f(a' $:= b')) = upd a' b' (finfun-rec cnst upd f)
  including finfun
proof -
  obtain b where b: b = finfun-default f by auto
  let ?the = λf g. f = Abs-finfun (map-default b g) ∧ finite (dom g) ∧ b ∉ ran g
  obtain g where g: g = The (?the f) by blast
  obtain y where f: f = Abs-finfun y and y: y ∈ finfun by (cases f)
  from f y b have bfin: finite {a. y a ≠ b} by(simp add: finfun-default-def finite-finfun-default-aux)

  let ?g = (λa. Some (y a)) |` {a. y a ≠ b}
  from bfin have fing: finite (dom ?g) by auto
  have bran: b ∉ ran ?g by(auto simp add: ran-def restrict-map-def)
  have yg: y = map-default b ?g by simp
  have gg: g = ?g unfolding g
  proof(rule the-equality)
    from f y bfin show ?the f ?g
      by(auto)(simp add: restrict-map-def ran-def split: if-split-asm)
  next
    fix g'
    assume ?the f g'
    hence fin': finite (dom g') and ran': b ∉ ran g'
      and eq: Abs-finfun (map-default b ?g) = Abs-finfun (map-default b g') using f yg by auto
    from fin' fing have map-default b ?g ∈ finfun map-default b g' ∈ finfun by(blast intro: map-default-in-finfun)+
    with eq have map-default b ?g = map-default b g' by simp
    with fing bran fin' ran' show g' = ?g by(rule map-default-inject[OF disjI2[OF refl], THEN sym])
  qed

  show ?thesis
  proof(cases b' = b)
    case True
    note b'b = True

    let ?g' = (λa. Some ((y(a' := b)) a)) |` {a. (y(a' := b)) a ≠ b}
    from bfin b'b have fing': finite (dom ?g')
      by(auto simp add: Collect-conj-eq Collect-imp-eq intro: finite-subset)
    have brang': b ∉ ran ?g' by(auto simp add: ran-def restrict-map-def)

    let ?b' = λa. case ?g' a of None ⇒ b | Some b ⇒ b
    let ?b = map-default b ?g
    from upd-left-comm upd-left-comm fing'
    have Finite-Set.fold (λa. upd a (?b' a)) (cnst b) (dom ?g') = Finite-Set.fold (λa. upd a (?b a)) (cnst b) (dom ?g')
      by(rule finite-rec-cong1)(auto simp add: restrict-map-def b'b b map-default-def)
    also interpret gwf: comp-fun-commute λa. upd a (?b a) by(rule upd-left-comm)
    have Finite-Set.fold (λa. upd a (?b a)) (cnst b) (dom ?g') = upd a' b' (Finite-Set.fold
```

(λa. upd a (?b a)) (cnst b) (dom ?g))
proof(cases y a′ = b)
case True
with b′b **have** g′: ?g′ = ?g **by**(auto simp add: restrict-map-def)
from True **have** a′ndomg: a′ ∉ dom ?g **by** auto
from f b′b b **show** ?thesis **unfolding** g′
by(subst map-default-update-const[OF fing a′ndomg map-le-refl, symmetric]) simp
next
case False
hence domg: dom ?g = insert a′ (dom ?g′) **by** auto
from False b′b **have** a′ndomg′: a′ ∉ dom ?g′ **by** auto
have Finite-Set.fold (λa. upd a (?b a)) (cnst b) (insert a′ (dom ?g′)) =
upd a′ (?b a′) (Finite-Set.fold (λa. upd a (?b a)) (cnst b) (dom ?g′))
using fing′ a′ndomg′ **unfolding** b′b **by**(rule gwf.fold-insert)
hence upd a′ b (Finite-Set.fold (λa. upd a (?b a)) (cnst b) (insert a′ (dom ?g′))) =
upd a′ b (upd a′ (?b a′) (Finite-Set.fold (λa. upd a (?b a)) (cnst b) (dom ?g′))) **by** simp
also from b′b **have** g′leg: ?g′ $\subseteq_m$?g **by**(auto simp add: restrict-map-def map-le-def)
note map-default-update-twice[OF fing′ a′ndomg′ this, of b ?b a′ b]
also note map-default-update-const[OF fing′ a′ndomg′ g′leg, of b]
finally show ?thesis **unfolding** b′b domg[unfolded b′b] **by**(rule sym)
qed
also have The (?the (f(a′ $:= b′))) = ?g′
proof(rule the-equality)
from f y b b′b brang′ fing′ **show** ?the (f(a′ $:= b′)) ?g′
by(auto simp del: fun-upd-apply simp add: finfun-update-def)
next
fix g′
assume ?the (f(a′ $:= b′)) g′
hence fin′: finite (dom g′) **and** ran′: b ∉ ran g′
and eq: f(a′ $:= b′) = Abs-finfun (map-default b g′)
by(auto simp del: fun-upd-apply)
from fin′ fing′ **have** map-default b g′ ∈ finfun map-default b ?g′ ∈ finfun
by(blast intro: map-default-in-finfun)+
with eq f b′b b **have** map-default b ?g′ = map-default b g′
by(simp del: fun-upd-apply add: finfun-update-def)
with fing′ brang′ fin′ ran′ **show** g′ = ?g′
by(rule map-default-inject[OF disjI2[OF refl], THEN sym])
qed
ultimately show ?thesis **unfolding** finfun-rec-def Let-def b gg[unfolded g b] **using** bfin b′b b
by(simp only: finfun-default-update-const map-default-def)
next
case False
note b′b = this
let ?g′ = ?g(a′ ↦ b′)

let *?b′* = *map-default b ?g′*
let *?b* = *map-default b ?g*
from *fing* **have** *fing′*: *finite* (*dom ?g′*) **by** *auto*
from *bran b′b* **have** *bnrang′*: *b* ∉ *ran ?g′* **by**(*auto simp add*: *ran-def*)
have *ffmg′*: *map-default b ?g′* = *y*(*a′* := *b′*) **by**(*auto simp add*: *map-default-def restrict-map-def*)
with *f y* **have** *f-Abs*: *f*(*a′* $:= *b′*) = *Abs-finfun* (*map-default b ?g′*) **by**(*auto simp add*: *finfun-update-def*)
have *g′*: *The* (*?the* (*f*(*a′* $:= *b′*))) = *?g′*
proof (*rule the-equality*)
from *fing′ bnrang′ f-Abs* **show** *?the* (*f*(*a′* $:= *b′*)) *?g′*
by(*auto simp add*: *finfun-update-def restrict-map-def*)
next
fix *g′* **assume** *?the* (*f*(*a′* $:= *b′*)) *g′*
hence *f′*: *f*(*a′* $:= *b′*) = *Abs-finfun* (*map-default b g′*)
and *fin′*: *finite* (*dom g′*) **and** *brang′*: *b* ∉ *ran g′* **by** *auto*
from *fing′ fin′* **have** *map-default b ?g′* ∈ *finfun map-default b g′* ∈ *finfun*
by(*auto intro*: *map-default-in-finfun*)
with *f′ f-Abs* **have** *map-default b g′* = *map-default b ?g′* **by** *simp*
with *fin′ brang′ fing′ bnrang′* **show** *g′* = *?g′*
by(*rule map-default-inject*[*OF disjI2*[*OF refl*]])
qed
have *dom*: *dom* (((λ*a*. *Some* (*y a*)) |‘ {*a*. *y a* ≠ *b*})(*a′* ↦ *b′*)) = *insert a′* (*dom* ((λ*a*. *Some* (*y a*)) |‘ {*a*. *y a* ≠ *b*}))
by *auto*
show *?thesis*
proof(*cases y a′* = *b*)
case *True*
hence *a′ndomg*: *a′* ∉ *dom ?g* **by** *auto*
from *f y b′b True* **have** *yff*: *y* = *map-default b* (*?g′* |‘ *dom ?g*)
by(*auto simp add*: *restrict-map-def map-default-def intro*!: *ext*)
hence *f′*: *f* = *Abs-finfun* (*map-default b* (*?g′* |‘ *dom ?g*)) **using** *f* **by** *simp*
interpret *g′wf*: *comp-fun-commute* λ*a*. *upd a* (*?b′ a*) **by**(*rule upd-left-comm*)
from *upd-left-comm upd-left-comm fing*
have *Finite-Set.fold* (λ*a*. *upd a* (*?b a*)) (*cnst b*) (*dom ?g*) = *Finite-Set.fold* (λ*a*. *upd a* (*?b′ a*)) (*cnst b*) (*dom ?g*)
by(*rule finite-rec-cong1*)(*auto simp add*: *restrict-map-def b′b True map-default-def*)
thus *?thesis* **unfolding** *finfun-rec-def Let-def finfun-default-update-const b*[*symmetric*]
unfolding *g′ g*[*symmetric*] *gg g′wf.fold-insert*[*OF fing a′ndomg*, *of cnst b*, *folded dom*]
by −(*rule arg-cong2*[**where** *f*=*upd a′*], *simp-all add*: *map-default-def*)
next
case *False*
hence *insert a′* (*dom ?g*) = *dom ?g* **by** *auto*
moreover {
let *?g′′* = *?g*(*a′* := *None*)
let *?b′′* = *map-default b ?g′′*
from *False* **have** *domg*: *dom ?g* = *insert a′* (*dom ?g′′*) **by** *auto*

from *False* **have** *a′ndomg″*: $a' \notin$ *dom ?g″* **by** *auto*
have *fing″*: *finite* (*dom ?g″*) **by**(*rule finite-subset*[*OF - fing*]) *auto*
have *bnrang″*: $b \notin$ *ran ?g″* **by**(*auto simp add*: *ran-def restrict-map-def*)
interpret *gwf*: *comp-fun-commute* λ*a*. *upd a* (*?b a*) **by**(*rule upd-left-comm*)
interpret *g′wf*: *comp-fun-commute* λ*a*. *upd a* (*?b′ a*) **by**(*rule upd-left-comm*)
have *upd a′ b′* (*Finite-Set.fold* (λ*a*. *upd a* (*?b a*)) (*cnst b*) (*insert a′* (*dom ?g″*))) =
upd a′ b′ (*upd a′* (*?b a′*) (*Finite-Set.fold* (λ*a*. *upd a* (*?b a*)) (*cnst b*) (*dom ?g″*)))
unfolding *gwf.fold-insert*[*OF fing″ a′ndomg″*] *f* **..**
also have *g″leg*: *?g* |‘ *dom ?g″* $\subseteq_m$ *?g* **by**(*auto simp add*: *map-le-def*)
have *dom* (*?g* |‘ *dom ?g″*) = *dom ?g″* **by** *auto*
note *map-default-update-twice*[**where** *d*=*b* **and** *f* = *?g* |‘ *dom ?g″* **and** *a*=*a′* **and** *d′*=*?b a′* **and** *d″*=*b′* **and** *g*=*?g*,
unfolded this, *OF fing″ a′ndomg″ g″leg*]
also have *b′*: *b′* = *?b′ a′* **by**(*auto simp add*: *map-default-def*)
from *upd-left-comm upd-left-comm fing″*
have *Finite-Set.fold* (λ*a*. *upd a* (*?b a*)) (*cnst b*) (*dom ?g″*) =
Finite-Set.fold (λ*a*. *upd a* (*?b′ a*)) (*cnst b*) (*dom ?g″*)
by(*rule finite-rec-cong1*)(*auto simp add*: *restrict-map-def b′b map-default-def*)
with *b′* **have** *upd a′ b′* (*Finite-Set.fold* (λ*a*. *upd a* (*?b a*)) (*cnst b*) (*dom ?g″*)) =
upd a′ (*?b′ a′*) (*Finite-Set.fold* (λ*a*. *upd a* (*?b′ a*)) (*cnst b*) (*dom ?g″*)) **by** *simp*
also note *g′wf.fold-insert*[*OF fing″ a′ndomg″*, *symmetric*]
finally have *upd a′ b′* (*Finite-Set.fold* (λ*a*. *upd a* (*?b a*)) (*cnst b*) (*dom ?g*)) =
Finite-Set.fold (λ*a*. *upd a* (*?b′ a*)) (*cnst b*) (*dom ?g*)
unfolding *domg* **. }**
ultimately have *Finite-Set.fold* (λ*a*. *upd a* (*?b′ a*)) (*cnst b*) (*insert a′* (*dom ?g*)) =
upd a′ b′ (*Finite-Set.fold* (λ*a*. *upd a* (*?b a*)) (*cnst b*) (*dom ?g*)) **by** *simp*
thus *?thesis* **unfolding** *finfun-rec-def Let-def finfun-default-update-const b*[*symmetric*] *g*[*symmetric*] *g′ dom*[*symmetric*]
using *b′b gg* **by**(*simp add*: *map-default-insert*)
qed
qed
qed

end

locale *finfun-rec-wf* = *finfun-rec-wf-aux* +
assumes *const-update-all*:
finite (*UNIV* :: *′a set*) ⟹ *Finite-Set.fold* (λ*a*. *upd a b′*) (*cnst b*) (*UNIV* :: *′a set*) = *cnst b′*
begin

lemma *finfun-rec-const* [*simp*]: *finfun-rec cnst upd* (*K$ c*) = *cnst c*

including *finfun*
proof(*cases finite* (*UNIV* :: *'a set*))
case *False*
hence *finfun-default* ((*K$ c*) :: *'a* ⇒*f* *'b*) = *c* **by**(*simp add*: *finfun-default-const*)
moreover have (*THE g* :: *'a* ⇀ *'b*. (*K$ c*) = *Abs-finfun* (*map-default c g*) ∧ *finite* (*dom g*) ∧ *c* ∉ *ran g*) = *Map.empty*
proof (*rule the-equality*)
show (*K$ c*) = *Abs-finfun* (*map-default c Map.empty*) ∧ *finite* (*dom Map.empty*) ∧ *c* ∉ *ran Map.empty*
by(*auto simp add*: *finfun-const-def*)
next
fix *g* :: *'a* ⇀ *'b*
assume (*K$ c*) = *Abs-finfun* (*map-default c g*) ∧ *finite* (*dom g*) ∧ *c* ∉ *ran g*
hence *g*: (*K$ c*) = *Abs-finfun* (*map-default c g*) **and** *fin*: *finite* (*dom g*) **and** *ran*: *c* ∉ *ran g* **by** *blast*+
from *g map-default-in-finfun*[*OF fin*, *of c*] **have** *map-default c g* = (λ*a*. *c*)
by(*simp add*: *finfun-const-def*)
moreover have *map-default c Map.empty* = (λ*a*. *c*) **by** *simp*
ultimately show *g* = *Map.empty* **by**−(*rule map-default-inject*[*OF disjI2*[*OF refl*] *fin ran*], *auto*)
qed
ultimately show *?thesis* **by**(*simp add*: *finfun-rec-def*)
next
case *True*
hence *default*: *finfun-default* ((*K$ c*) :: *'a* ⇒*f* *'b*) = *undefined* **by**(*simp add*: *finfun-default-const*)
let *?the* = λ*g* :: *'a* ⇀ *'b*. (*K$ c*) = *Abs-finfun* (*map-default undefined g*) ∧ *finite* (*dom g*) ∧ *undefined* ∉ *ran g*
show *?thesis*
proof(*cases c* = *undefined*)
case *True*
have *the*: *The ?the* = *Map.empty*
proof (*rule the-equality*)
from *True* **show** *?the Map.empty* **by**(*auto simp add*: *finfun-const-def*)
next
fix *g'*
assume *?the g'*
hence *fg*: (*K$ c*) = *Abs-finfun* (*map-default undefined g'*)
and *fin*: *finite* (*dom g'*) **and** *g*: *undefined* ∉ *ran g'* **by** *simp-all*
from *fin* **have** *map-default undefined g'* ∈ *finfun* **by**(*rule map-default-in-finfun*)
with *fg* **have** *map-default undefined g'* = (λ*a*. *c*)
by(*auto simp add*: *finfun-const-def intro*: *Abs-finfun-inject*[*THEN iffD1*, *symmetric*])
with *True* **show** *g'* = *Map.empty*
by −(*rule map-default-inject*(*2*)[*OF - fin g*], *auto*)
qed
show *?thesis* **unfolding** *finfun-rec-def* **using** ‹*finite UNIV*› *True*
unfolding *Let-def the default* **by**(*simp*)
next

case *False*
have *the*: *The ?the* = (λa :: *'a. Some c*)
proof (*rule the-equality*)
from *False True* **show** *?the* (λa :: *'a. Some c*)
by(*auto simp add*: *map-default-def* [*abs-def*] *finfun-const-def dom-def ran-def*)
next
fix *g'* :: *'a* ⇀ *'b*
assume *?the g'*
hence *fg*: (*K$ c*) = *Abs-finfun* (*map-default undefined g'*)
and *fin*: *finite* (*dom g'*) **and** *g*: *undefined* ∉ *ran g'* **by** *simp-all*
from *fin* **have** *map-default undefined g'* ∈ *finfun* **by**(*rule map-default-in-finfun*)
with *fg* **have** *map-default undefined g'* = (λa. *c*)
by(*auto simp add*: *finfun-const-def intro*: *Abs-finfun-inject*[*THEN iffD1*])
with *True False* **show** *g'* = (λa::*'a. Some c*)
by − (*rule map-default-inject*(*2*)[*OF - fin g*],
auto simp add: *dom-def ran-def map-default-def* [*abs-def*])
qed
show *?thesis* **unfolding** *finfun-rec-def* **using** *True False*
unfolding *Let-def the default* **by**(*simp add*: *dom-def map-default-def const-update-all*)
qed
qed

end

1.9 Weak induction rule and case analysis for FinFuns

lemma *finfun-weak-induct* [*consumes 0*, *case-names const update*]:
assumes *const*: ⋀*b*. *P* (*K$ b*)
and *update*: ⋀*f a b*. *P f* ⟹ *P* (*f*(*a* $:= *b*))
shows *P x*
including *finfun*
proof(*induct x rule*: *Abs-finfun-induct*)
case (*Abs-finfun y*)
then obtain *b* **where** *finite* {*a*. *y a* ≠ *b*} **unfolding** *finfun-def* **by** *blast*
thus *?case* **using** ‹*y* ∈ *finfun*›
proof(*induct* {*a*. *y a* ≠ *b*} *arbitrary*: *y rule*: *finite-induct*)
case *empty*
hence ⋀*a*. *y a* = *b* **by** *blast*
hence *y* = (λa. *b*) **by**(*auto*)
hence *Abs-finfun y* = *finfun-const b* **unfolding** *finfun-const-def* **by** *simp*
thus *?case* **by**(*simp add*: *const*)
next
case (*insert a A*)
note *IH* = ‹⋀*y*. ⟦ *A* = {*a*. *y a* ≠ *b*}; *y* ∈ *finfun* ⟧ ⟹ *P* (*Abs-finfun y*)›
note *y* = ‹*y* ∈ *finfun*›
with ‹*insert a A* = {*a*. *y a* ≠ *b*}› ‹*a* ∉ *A*›
have *A* = {*a'*. (*y*(*a* := *b*)) *a'* ≠ *b*} *y*(*a* := *b*) ∈ *finfun* **by** *auto*
from *IH*[*OF this*] **have** *P* (*finfun-update* (*Abs-finfun* (*y*(*a* := *b*))) *a* (*y a*))
by(*rule update*)

thus *?case* **using** *y* **unfolding** *finfun-update-def* **by** *simp*
qed
qed

lemma *finfun-exhaust-disj*: (∃ *b*. *x* = *finfun-const b*) ∨ (∃ *f a b*. *x* = *finfun-update f a b*)
by(*induct x rule*: *finfun-weak-induct*) *blast+*

lemma *finfun-exhaust*:
obtains *b* **where** *x* = (*K$ b*)
| *f a b* **where** *x* = *f*(*a* $:= *b*)
by(*atomize-elim*)(*rule finfun-exhaust-disj*)

lemma *finfun-rec-unique*:
fixes *f* :: *'a* ⇒*f* *'b* ⇒ *'c*
assumes *c*: ⋀*c*. *f* (*K$ c*) = *cnst c*
and *u*: ⋀*g a b*. *f* (*g*(*a* $:= *b*)) = *upd g a b* (*f g*)
and *c'*: ⋀*c*. *f'* (*K$ c*) = *cnst c*
and *u'*: ⋀*g a b*. *f'* (*g*(*a* $:= *b*)) = *upd g a b* (*f' g*)
shows *f* = *f'*
proof
fix *g* :: *'a* ⇒*f* *'b*
show *f g* = *f' g*
by(*induct g rule*: *finfun-weak-induct*)(*auto simp add*: *c u c' u'*)
qed

1.10 Function application

notation *finfun-apply* (**infixl** ‹$› *999*)

interpretation *finfun-apply-aux*: *finfun-rec-wf-aux* λ*b*. *b* λ*a' b c*. *if* (*a* = *a'*) *then b else c*
by(*unfold-locales*) *auto*

interpretation *finfun-apply*: *finfun-rec-wf* λ*b*. *b* λ*a' b c*. *if* (*a* = *a'*) *then b else c*
proof(*unfold-locales*)
fix *b' b* :: *'a*
assume *fin*: *finite* (*UNIV* :: *'b set*)
{ fix *A* :: *'b set*
interpret *comp-fun-commute* λ*a'*. *If* (*a* = *a'*) *b'* **by**(*rule finfun-apply-aux.upd-left-comm*)
from *fin* **have** *finite A* **by**(*auto intro*: *finite-subset*)
hence *Finite-Set.fold* (λ*a'*. *If* (*a* = *a'*) *b'*) *b A* = (*if a* ∈ *A then b' else b*)
by *induct auto* **}**
from *this*[*of UNIV*] **show** *Finite-Set.fold* (λ*a'*. *If* (*a* = *a'*) *b'*) *b UNIV* = *b'* **by** *simp*
qed

lemma *finfun-apply-def*: ($) = (λ*f a*. *finfun-rec* (λ*b*. *b*) (λ*a' b c*. *if* (*a* = *a'*) *then b else c*) *f*)

proof(*rule finfun-rec-unique*)
 fix *c* **show** ($) (*K*$ *c*) = (λ*a*. *c*) **by**(*simp add*: *finfun-const.rep-eq*)
next
 fix *g a b* **show** ($) *g*(*a* $:= *b*) = (λ*c*. *if c* = *a then b else g* $ *c*)
  by(*auto simp add*: *finfun-update-def fun-upd-finfun Abs-finfun-inverse finfun-apply*)
qed *auto*

lemma *finfun-upd-apply*: $f(a$ $:= $b)$ $ $a' = ($*if* $a = a'$ *then* b *else* f $ $a')$
 and *finfun-upd-apply-code* [*code*]: (*finfun-update-code f a b*) $ a' = (*if* $a = a'$ *then* b *else* f $ a')
by(*simp-all add*: *finfun-apply-def*)

lemma *finfun-const-apply* [*simp*, *code*]: (*K*$ *b*) $ *a* = *b*
by(*simp add*: *finfun-apply-def*)

lemma *finfun-upd-apply-same* [*simp*]:
 f(*a* $:= *b*) $ *a* = *b*
by(*simp add*: *finfun-upd-apply*)

lemma *finfun-upd-apply-other* [*simp*]:
 $a \neq a' \Longrightarrow f(a$ $:= $b)$ $ $a' = f$ $ a'
by(*simp add*: *finfun-upd-apply*)

lemma *finfun-ext*: ($\bigwedge a$. *f* $ *a* = *g* $ *a*) $\Longrightarrow$ *f* = *g*
by(*auto simp add*: *finfun-apply-inject*[*symmetric*])

lemma *expand-finfun-eq*: (*f* = *g*) = (($) *f* = ($) *g*)
by(*auto intro*: *finfun-ext*)

lemma *finfun-upd-triv* [*simp*]: *f*(*x* $:= *f* $ *x*) = *f*
by(*simp add*: *expand-finfun-eq fun-eq-iff finfun-upd-apply*)

lemma *finfun-const-inject* [*simp*]: (*K*$ *b*) = (*K*$ b') ≡ $b = b'$
by(*simp add*: *expand-finfun-eq fun-eq-iff*)

lemma *finfun-const-eq-update*:
 ((*K*$ *b*) = *f*(*a* $:= b')) = ($b = b' \wedge (\forall a'. a \neq a' \longrightarrow f$ $ $a' = b)$)
by(*auto simp add*: *expand-finfun-eq fun-eq-iff finfun-upd-apply*)

1.11 Function composition

definition *finfun-comp* :: ($'a \Rightarrow 'b) \Rightarrow 'c \Rightarrow_f 'a \Rightarrow 'c \Rightarrow_f 'b$ (**infixr** ‹∘$› *55*)
where [*code del*]: *g* ∘$ *f* = *finfun-rec* (λ*b*. (*K*$ *g b*)) (λ*a b c*. *c*(*a* $:= *g b*)) *f*

notation (*ASCII*)
 finfun-comp (**infixr** ‹*o*$› *55*)

interpretation *finfun-comp-aux*: *finfun-rec-wf-aux* (λ*b*. (*K*$ *g b*)) (λ*a b c*. *c*(*a* $:= *g b*))

by(*unfold-locales*)(*auto simp add*: *finfun-upd-apply intro*: *finfun-ext*)

interpretation *finfun-comp*: *finfun-rec-wf* (λ*b*. (*K*\$ *g b*)) (λ*a b c*. *c*(*a* \$:= *g b*))
proof
  fix *b′ b* :: ′*a*
  assume *fin*: *finite* (*UNIV* :: ′*c set*)
  { fix *A* :: ′*c set*
    from *fin* **have** *finite A* **by**(*auto intro*: *finite-subset*)
    hence *Finite-Set.fold* (λ(*a* :: ′*c*) *c*. *c*(*a* \$:= *g b′*)) (*K*\$ *g b*) *A* =
      Abs-finfun (λ*a*. *if a* ∈ *A then g b′ else g b*)
       by *induct* (*simp-all add*: *finfun-const-def*, *auto simp add*: *finfun-update-def*
Abs-finfun-inverse-finite fun-upd-def Abs-finfun-inject-finite fun-eq-iff fin) **}**
  from *this*[*of UNIV*] **show** *Finite-Set.fold* (λ(*a* :: ′*c*) *c*. *c*(*a* \$:= *g b′*)) (*K*\$ *g b*)
UNIV = (*K*\$ *g b′*)
    by(*simp add*: *finfun-const-def*)
qed

lemma *finfun-comp-const* [*simp*, *code*]:
  g ∘\$ (*K*\$ *c*) = (*K*\$ *g c*)
by(*simp add*: *finfun-comp-def*)

lemma *finfun-comp-update* [*simp*]: *g* ∘\$ (*f*(*a* \$:= *b*)) = (*g* ∘\$ *f*)(*a* \$:= *g b*)
  and *finfun-comp-update-code* [*code*]:
  g ∘\$ (*finfun-update-code f a b*) = *finfun-update-code* (*g* ∘\$ *f*) *a* (*g b*)
by(*simp-all add*: *finfun-comp-def*)

lemma *finfun-comp-apply* [*simp*]:
  (\$) (*g* ∘\$ *f*) = *g* ∘ (\$) *f*
by(*induct f rule*: *finfun-weak-induct*)(*auto simp add*: *finfun-upd-apply*)

lemma *finfun-comp-comp-collapse* [*simp*]: *f* ∘\$ *g* ∘\$ *h* = (*f* ∘ *g*) ∘\$ *h*
by(*induct h rule*: *finfun-weak-induct*) *simp-all*

lemma *finfun-comp-const1* [*simp*]: (λ*x*. *c*) ∘\$ *f* = (*K*\$ *c*)
by(*induct f rule*: *finfun-weak-induct*)(*auto intro*: *finfun-ext simp add*: *finfun-upd-apply*)

lemma *finfun-comp-id1* [*simp*]: (λ*x*. *x*) ∘\$ *f* = *f id* ∘\$ *f* = *f*
by(*induct f rule*: *finfun-weak-induct*) *auto*

lemma *finfun-comp-conv-comp*: *g* ∘\$ *f* = *Abs-finfun* (*g* ∘ (\$) *f*)
  including *finfun*
proof −
  have (λ*f*. *g* ∘\$ *f*) = (λ*f*. *Abs-finfun* (*g* ∘ (\$) *f*))
  proof(*rule finfun-rec-unique*)
    { fix *c* **show** *Abs-finfun* (*g* ∘ (\$) (*K*\$ *c*)) = (*K*\$ *g c*)
        by(*simp add*: *finfun-comp-def o-def*)(*simp add*: *finfun-const-def*) **}**
    { fix *g′ a b* **show** *Abs-finfun* (*g* ∘ (\$) *g′*(*a* \$:= *b*)) = (*Abs-finfun* (*g* ∘ (\$) *g′*))(*a*
\$:= *g b*)
      proof −

```
    obtain y where y: y ∈ finfun and g': g' = Abs-finfun y by(cases g')
   moreover from g' have (g ∘ ($) g') ∈ finfun by(simp add: finfun-left-compose)
    moreover have g ∘ y(a := b) = (g ∘ y)(a := g b) by(auto)
    ultimately show ?thesis by(simp add: finfun-comp-def finfun-update-def)
   qed }
 qed auto
 thus ?thesis by(auto simp add: fun-eq-iff)
qed

definition finfun-comp2 :: 'b ⇒f 'c ⇒ ('a ⇒ 'b) ⇒ 'a ⇒f 'c  (infixr ‹$∘› 55)
where [code del]: g $∘ f = Abs-finfun (($) g ∘ f)

notation (ASCII)
 finfun-comp2 (infixr ‹$o› 55)

lemma finfun-comp2-const [code, simp]: finfun-comp2 (K$ c) f = (K$ c)
 including finfun
by(simp add: finfun-comp2-def finfun-const-def comp-def)

lemma finfun-comp2-update:
 assumes inj: inj f
 shows finfun-comp2 (g(b $:= c)) f = (if b ∈ range f then (finfun-comp2 g f)(inv
f b $:= c) else finfun-comp2 g f)
 including finfun
proof(cases b ∈ range f)
 case True
 from inj have ⋀x. (($) g)(f x := c) ∘ f = (($) g ∘ f)(x := c) by(auto intro!:
ext dest: injD)
 with inj True show ?thesis by(auto simp add: finfun-comp2-def finfun-update-def
finfun-right-compose)
next
 case False
 hence (($) g)(b := c) ∘ f = ($) g ∘ f by(auto simp add: fun-eq-iff)
 with False show ?thesis by(auto simp add: finfun-comp2-def finfun-update-def)
qed
```

1.12 Universal quantification

```
definition finfun-All-except :: 'a list ⇒ 'a ⇒f bool ⇒ bool
where [code del]: finfun-All-except A P ≡ ∀ a. a ∈ set A ∨ P $ a

lemma finfun-All-except-const: finfun-All-except A (K$ b) ⟷ b ∨ set A = UNIV
by(auto simp add: finfun-All-except-def)

lemma finfun-All-except-const-finfun-UNIV-code [code]:
 finfun-All-except A (K$ b) = (b ∨ is-list-UNIV A)
by(simp add: finfun-All-except-const is-list-UNIV-iff)

lemma finfun-All-except-update:
```

finfun-All-except A f(*a* $:= *b*) = ((*a* ∈ *set A* ∨ *b*) ∧ *finfun-All-except* (*a* # *A*) *f*)
by(*fastforce simp add*: *finfun-All-except-def finfun-upd-apply*)

lemma *finfun-All-except-update-code* [*code*]:
fixes *a* :: *'a* :: *card-UNIV*
shows *finfun-All-except A* (*finfun-update-code f a b*) = ((*a* ∈ *set A* ∨ *b*) ∧ *finfun-All-except* (*a* # *A*) *f*)
by(*simp add*: *finfun-All-except-update*)

definition *finfun-All* :: *'a* ⇒*f bool* ⇒ *bool*
where *finfun-All* = *finfun-All-except* []

lemma *finfun-All-const* [*simp*]: *finfun-All* (*K*$ *b*) = *b*
by(*simp add*: *finfun-All-def finfun-All-except-def*)

lemma *finfun-All-update*: *finfun-All f*(*a* $:= *b*) = (*b* ∧ *finfun-All-except* [*a*] *f*)
by(*simp add*: *finfun-All-def finfun-All-except-update*)

lemma *finfun-All-All*: *finfun-All P* = *All* (($) *P*)
by(*simp add*: *finfun-All-def finfun-All-except-def*)

definition *finfun-Ex* :: *'a* ⇒*f bool* ⇒ *bool*
where *finfun-Ex P* = *Not* (*finfun-All* (*Not* ∘$ *P*))

lemma *finfun-Ex-Ex*: *finfun-Ex P* = *Ex* (($) *P*)
unfolding *finfun-Ex-def finfun-All-All* **by** *simp*

lemma *finfun-Ex-const* [*simp*]: *finfun-Ex* (*K*$ *b*) = *b*
by(*simp add*: *finfun-Ex-def*)

1.13 A diagonal operator for FinFuns

definition *finfun-Diag* :: *'a* ⇒*f 'b* ⇒ *'a* ⇒*f 'c* ⇒ *'a* ⇒*f* (*'b* × *'c*) (‹(*1*'($-,/ -$'))› [*0*, *0*] *1000*)
where [*code del*]: ($*f*, *g*$) = *finfun-rec* (λ*b*. *Pair b* ∘$ *g*) (λ*a b c*. *c*(*a* $:= (*b*, *g* $ *a*))) *f*

interpretation *finfun-Diag-aux*: *finfun-rec-wf-aux* λ*b*. *Pair b* ∘$ *g* λ*a b c*. *c*(*a* $:= (*b*, *g* $ *a*))
by(*unfold-locales*)(*simp-all add*: *expand-finfun-eq fun-eq-iff finfun-upd-apply*)

interpretation *finfun-Diag*: *finfun-rec-wf* λ*b*. *Pair b* ∘$ *g* λ*a b c*. *c*(*a* $:= (*b*, *g* $ *a*))
proof
fix *b' b* :: *'a*
assume *fin*: *finite* (*UNIV* :: *'c set*)
{ fix *A* :: *'c set*
interpret *comp-fun-commute* λ*a c*. *c*(*a* $:= (*b'*, *g* $ *a*)) **by**(*rule finfun-Diag-aux.upd-left-comm*)

from *fin* **have** *finite A* **by**(*auto intro*: *finite-subset*)
hence *Finite-Set.fold* (λa *c*. *c*(*a* \$:= (b', *g* \$ *a*))) (*Pair b* ∘\$ *g*) *A* =
Abs-finfun (λa. (*if a* ∈ *A then* b' *else b*, *g* \$ *a*))
by(*induct*)(*simp-all add*: *finfun-const-def finfun-comp-conv-comp o-def*,
auto simp add: *finfun-update-def Abs-finfun-inverse-finite fun-upd-def Abs-finfun-inject-finite fun-eq-iff fin*) **}**
from *this*[*of UNIV*] **show** *Finite-Set.fold* (λa *c*. *c*(*a* \$:= (b', *g* \$ *a*))) (*Pair b* ∘\$ *g*) *UNIV* = *Pair* b' ∘\$ *g*
by(*simp add*: *finfun-const-def finfun-comp-conv-comp o-def*)
qed

lemma *finfun-Diag-const1*: (\$*K*\$ *b*, *g*\$) = *Pair b* ∘\$ *g*
by(*simp add*: *finfun-Diag-def*)

Do not use (\$*K*\$ *?b*, *?g*\$) = *Pair ?b* ∘\$ *?g* for the code generator because *Pair b* is injective, i.e. if *g* is free of redundant updates, there is no need to check for redundant updates as is done for (∘\$).

lemma *finfun-Diag-const-code* [*code*]:
(\$*K*\$ *b*, *K*\$ *c*\$) = (*K*\$ (*b*, *c*))
(\$*K*\$ *b*, *finfun-update-code g a c*\$) = *finfun-update-code* (\$*K*\$ *b*, *g*\$) *a* (*b*, *c*)
by(*simp-all add*: *finfun-Diag-const1*)

lemma *finfun-Diag-update1*: (\$*f*(*a* \$:= *b*), *g*\$) = (\$*f*, *g*\$)(*a* \$:= (*b*, *g* \$ *a*))
and *finfun-Diag-update1-code* [*code*]: (\$*finfun-update-code f a b*, *g*\$) = (\$*f*, *g*\$)(*a* \$:= (*b*, *g* \$ *a*))
by(*simp-all add*: *finfun-Diag-def*)

lemma *finfun-Diag-const2*: (\$*f*, *K*\$ *c*\$) = (λb. (*b*, *c*)) ∘\$ *f*
by(*induct f rule*: *finfun-weak-induct*)(*auto intro*!: *finfun-ext simp add*: *finfun-upd-apply finfun-Diag-const1 finfun-Diag-update1*)

lemma *finfun-Diag-update2*: (\$*f*, *g*(*a* \$:= *c*)\$) = (\$*f*, *g*\$)(*a* \$:= (*f* \$ *a*, *c*))
by(*induct f rule*: *finfun-weak-induct*)(*auto intro*!: *finfun-ext simp add*: *finfun-upd-apply finfun-Diag-const1 finfun-Diag-update1*)

lemma *finfun-Diag-const-const* [*simp*]: (\$*K*\$ *b*, *K*\$ *c*\$) = (*K*\$ (*b*, *c*))
by(*simp add*: *finfun-Diag-const1*)

lemma *finfun-Diag-const-update*:
(\$*K*\$ *b*, *g*(*a* \$:= *c*)\$) = (\$*K*\$ *b*, *g*\$)(*a* \$:= (*b*, *c*))
by(*simp add*: *finfun-Diag-const1*)

lemma *finfun-Diag-update-const*:
(\$*f*(*a* \$:= *b*), *K*\$ *c*\$) = (\$*f*, *K*\$ *c*\$)(*a* \$:= (*b*, *c*))
by(*simp add*: *finfun-Diag-def*)

lemma *finfun-Diag-update-update*:
(\$*f*(*a* \$:= *b*), *g*(a' \$:= *c*)\$) = (*if a* = a' *then* (\$*f*, *g*\$)(*a* \$:= (*b*, *c*)) *else* (\$*f*, *g*\$)(*a* \$:= (*b*, *g* \$ *a*))(a' \$:= (*f* \$ a', *c*)))

by(*auto simp add*: *finfun-Diag-update1 finfun-Diag-update2*)

lemma *finfun-Diag-apply* [*simp*]: ($) ($*f*, *g*$) = (λ*x*. (*f* $ *x*, *g* $ *x*))
by(*induct f rule*: *finfun-weak-induct*)(*auto simp add*: *finfun-Diag-const1 finfun-Diag-update1 finfun-upd-apply*)

lemma *finfun-Diag-conv-Abs-finfun*:
  ($*f*, *g*$) = *Abs-finfun* ((λ*x*. (*f* $ *x*, *g* $ *x*)))
  including *finfun*
proof −
  have (λ*f* :: ′*a* ⇒*f* ′*b*. ($*f*, *g*$)) = (λ*f*. *Abs-finfun* ((λ*x*. (*f* $ *x*, *g* $ *x*))))
  proof(*rule finfun-rec-unique*)
    { fix *c* **show** *Abs-finfun* (λ*x*. ((*K*$ *c*) $ *x*, *g* $ *x*)) = *Pair c* ∘$ *g*
        by(*simp add*: *finfun-comp-conv-comp o-def finfun-const-def*) **}**
    { fix *g*′ *a b*
      show *Abs-finfun* (λ*x*. (*g*′(*a* $:= *b*) $ *x*, *g* $ *x*)) =
            (*Abs-finfun* (λ*x*. (*g*′ $ *x*, *g* $ *x*)))(*a* $:= (*b*, *g* $ *a*))
        by(*auto simp add*: *finfun-update-def fun-eq-iff simp del*: *fun-upd-apply*) *simp*
}
  qed(*simp-all add*: *finfun-Diag-const1 finfun-Diag-update1*)
  thus *?thesis* **by**(*auto simp add*: *fun-eq-iff*)
qed

lemma *finfun-Diag-eq*: ($*f*, *g*$) = ($*f*′, *g*′$) ⟷ *f* = *f*′ ∧ *g* = *g*′
by(*auto simp add*: *expand-finfun-eq fun-eq-iff*)

definition *finfun-fst* :: ′*a* ⇒*f* (′*b* × ′*c*) ⇒ ′*a* ⇒*f* ′*b*
where [*code*]: *finfun-fst f* = *fst* ∘$ *f*

lemma *finfun-fst-const*: *finfun-fst* (*K*$ *bc*) = (*K*$ *fst bc*)
by(*simp add*: *finfun-fst-def*)

lemma *finfun-fst-update*: *finfun-fst* (*f*(*a* $:= *bc*)) = (*finfun-fst f*)(*a* $:= *fst bc*)
  and *finfun-fst-update-code*: *finfun-fst* (*finfun-update-code f a bc*) = (*finfun-fst f*)(*a* $:= *fst bc*)
by(*simp-all add*: *finfun-fst-def*)

lemma *finfun-fst-comp-conv*: *finfun-fst* (*f* ∘$ *g*) = (*fst* ∘ *f*) ∘$ *g*
by(*simp add*: *finfun-fst-def*)

lemma *finfun-fst-conv* [*simp*]: *finfun-fst* ($*f*, *g*$) = *f*
by(*induct f rule*: *finfun-weak-induct*)(*simp-all add*: *finfun-Diag-const1 finfun-fst-comp-conv o-def finfun-Diag-update1 finfun-fst-update*)

lemma *finfun-fst-conv-Abs-finfun*: *finfun-fst* = (λ*f*. *Abs-finfun* (*fst* ∘ ($) *f*))
by(*simp add*: *finfun-fst-def* [*abs-def*] *finfun-comp-conv-comp*)

definition *finfun-snd* :: ′*a* ⇒*f* (′*b* × ′*c*) ⇒ ′*a* ⇒*f* ′*c*

where [*code*]: *finfun-snd f* = *snd* ∘$ *f*

lemma *finfun-snd-const*: *finfun-snd* (*K$ bc*) = (*K$ snd bc*)
by(*simp add*: *finfun-snd-def*)

lemma *finfun-snd-update*: *finfun-snd* (*f*(*a* $:= *bc*)) = (*finfun-snd f*)(*a* $:= *snd bc*)
and *finfun-snd-update-code* [*code*]: *finfun-snd* (*finfun-update-code f a bc*) = (*finfun-snd f*)(*a* $:= *snd bc*)
by(*simp-all add*: *finfun-snd-def*)

lemma *finfun-snd-comp-conv*: *finfun-snd* (*f* ∘$ *g*) = (*snd* ∘ *f*) ∘$ *g*
by(*simp add*: *finfun-snd-def*)

lemma *finfun-snd-conv* [*simp*]: *finfun-snd* ($*f*, *g*$) = *g*
apply(*induct f rule*: *finfun-weak-induct*)
apply(*auto simp add*: *finfun-Diag-const1 finfun-snd-comp-conv o-def finfun-Diag-update1 finfun-snd-update finfun-upd-apply intro*: *finfun-ext*)
done

lemma *finfun-snd-conv-Abs-finfun*: *finfun-snd* = (λ*f*. *Abs-finfun* (*snd* ∘ ($) *f*))
by(*simp add*: *finfun-snd-def* [*abs-def*] *finfun-comp-conv-comp*)

lemma *finfun-Diag-collapse* [*simp*]: ($*finfun-fst f*, *finfun-snd f*$) = *f*
by(*induct f rule*: *finfun-weak-induct*)(*simp-all add*: *finfun-fst-const finfun-snd-const finfun-fst-update finfun-snd-update finfun-Diag-update-update*)

1.14 Currying for FinFuns

definition *finfun-curry* :: (′*a* × ′*b*) ⇒*f* ′*c* ⇒ ′*a* ⇒*f* ′*b* ⇒*f* ′*c*
where [*code del*]: *finfun-curry* = *finfun-rec* (*finfun-const* ∘ *finfun-const*) (λ(*a*, *b*) *c f*. *f*(*a* $:= (*f* $ *a*)(*b* $:= *c*)))

interpretation *finfun-curry-aux*: *finfun-rec-wf-aux finfun-const* ∘ *finfun-const* λ(*a*, *b*) *c f*. *f*(*a* $:= (*f* $ *a*)(*b* $:= *c*))
apply(*unfold-locales*)
apply(*auto simp add*: *split-def finfun-update-twist finfun-upd-apply split-paired-all finfun-update-const-same*)
done

interpretation *finfun-curry*: *finfun-rec-wf finfun-const* ∘ *finfun-const* λ(*a*, *b*) *c f*. *f*(*a* $:= (*f* $ *a*)(*b* $:= *c*))
proof(*unfold-locales*)
  fix *b*′ *b* :: ′*b*
  assume *fin*: *finite* (*UNIV* :: (′*c* × ′*a*) *set*)
  hence *fin1*: *finite* (*UNIV* :: ′*c set*) **and** *fin2*: *finite* (*UNIV* :: ′*a set*)
    unfolding *UNIV-Times-UNIV*[*symmetric*]
    by(*fastforce dest*: *finite-cartesian-productD1 finite-cartesian-productD2*)+
  note [*simp*] = *Abs-finfun-inverse-finite*[*OF fin*] *Abs-finfun-inverse-finite*[*OF fin1*] *Abs-finfun-inverse-finite*[*OF fin2*]

{ **fix** *A* :: ('*c* × '*a*) *set*
interpret *comp-fun-commute* λ*a* :: '*c* × '*a*. (λ(*a*, *b*) *c f*. *f*(*a* $:= (*f* $ *a*)(*b* $:= *c*))) *a b'*
by(*rule finfun-curry-aux.upd-left-comm*)
from *fin* **have** *finite A* **by**(*auto intro*: *finite-subset*)
hence *Finite-Set.fold* (λ*a* :: '*c* × '*a*. (λ(*a*, *b*) *c f*. *f*(*a* $:= (*f* $ *a*)(*b* $:= *c*))) *a b'*) ((*finfun-const* ∘ *finfun-const*) *b*) *A* = *Abs-finfun* (λ*a*. *Abs-finfun* (λ*b''*. *if* (*a*, *b''*) ∈ *A then b' else b*))
by *induct* (*simp-all*, *auto simp add*: *finfun-update-def finfun-const-def split-def intro*!: *arg-cong*[**where** *f*=*Abs-finfun*] *ext*) }
from *this*[*of UNIV*]
show *Finite-Set.fold* (λ*a* :: '*c* × '*a*. (λ(*a*, *b*) *c f*. *f*(*a* $:= (*f* $ *a*)(*b* $:= *c*))) *a b'*) ((*finfun-const* ∘ *finfun-const*) *b*) *UNIV* = (*finfun-const* ∘ *finfun-const*) *b'*
by(*simp add*: *finfun-const-def*)
qed

lemma *finfun-curry-const* [*simp*, *code*]: *finfun-curry* (*K*$ *c*) = (*K*$ *K*$ *c*)
by(*simp add*: *finfun-curry-def*)

lemma *finfun-curry-update* [*simp*]:
finfun-curry (*f*((*a*, *b*) $:= *c*)) = (*finfun-curry f*)(*a* $:= (*finfun-curry f* $ *a*)(*b* $:= *c*))
and *finfun-curry-update-code* [*code*]:
finfun-curry (*finfun-update-code f* (*a*, *b*) *c*) = (*finfun-curry f*)(*a* $:= (*finfun-curry f* $ *a*)(*b* $:= *c*))
by(*simp-all add*: *finfun-curry-def*)

lemma *finfun-Abs-finfun-curry*: **assumes** *fin*: *f* ∈ *finfun*
shows (λ*a*. *Abs-finfun* (*curry f a*)) ∈ *finfun*
including *finfun*
proof −
from *fin* **obtain** *c* **where** *c*: *finite* {*ab*. *f ab* ≠ *c*} **unfolding** *finfun-def* **by** *blast*
have {*a*. ∃ *b*. *f* (*a*, *b*) ≠ *c*} = *fst* ' {*ab*. *f ab* ≠ *c*} **by**(*force*)
hence {*a*. *curry f a* ≠ (λ*x*. *c*)} = *fst* ' {*ab*. *f ab* ≠ *c*}
by(*auto simp add*: *curry-def fun-eq-iff*)
with *fin c* **have** *finite* {*a*. *Abs-finfun* (*curry f a*) ≠ (*K*$ *c*)}
by(*simp add*: *finfun-const-def finfun-curry*)
thus *?thesis* **unfolding** *finfun-def* **by** *auto*
qed

lemma *finfun-curry-conv-curry*:
fixes *f* :: ('*a* × '*b*) ⇒*f* '*c*
shows *finfun-curry f* = *Abs-finfun* (λ*a*. *Abs-finfun* (*curry* (*finfun-apply f*) *a*))
including *finfun*
proof −
have *finfun-curry* = (λ*f* :: ('*a* × '*b*) ⇒*f* '*c*. *Abs-finfun* (λ*a*. *Abs-finfun* (*curry* (*finfun-apply f*) *a*)))
proof(*rule finfun-rec-unique*)
fix *c* **show** *finfun-curry* (*K*$ *c*) = (*K*$ *K*$ *c*) **by** *simp*

fix *f a*
show *finfun-curry* (*f*(*a* $:= *c*)) = (*finfun-curry f*)(*fst a* $:= (*finfun-curry f* $ (*fst a*))(*snd a* $:= *c*))
by(*cases a*) *simp*
show *Abs-finfun* (λ*a*. *Abs-finfun* (*curry* (*finfun-apply* (*K*$ *c*)) *a*)) = (*K*$ *K*$ *c*)
by(*simp add*: *finfun-curry-def finfun-const-def curry-def*)
fix *g b*
show *Abs-finfun* (λ*aa*. *Abs-finfun* (*curry* (($) *g*(*a* $:= *b*)) *aa*)) =
(*Abs-finfun* (λ*a*. *Abs-finfun* (*curry* (($) *g*) *a*)))(
fst a $:= ((*Abs-finfun* (λ*a*. *Abs-finfun* (*curry* (($) *g*) *a*))) $ (*fst a*))(*snd a* $:= *b*))
by(*cases a*)(*auto intro*!: *ext arg-cong*[**where** *f*=*Abs-finfun*] *simp add*: *finfun-curry-def finfun-update-def finfun-Abs-finfun-curry*)
qed
thus *?thesis* **by**(*auto simp add*: *fun-eq-iff*)
qed

1.15 Executable equality for FinFuns

lemma *eq-finfun-All-ext*: (*f* = *g*) ⟷ *finfun-All* ((λ(*x*, *y*). *x* = *y*) ∘$ ($*f*, *g*$))
by(*simp add*: *expand-finfun-eq fun-eq-iff finfun-All-All o-def*)

instantiation *finfun* :: ({*card-UNIV*,*equal*},*equal*) *equal* **begin**
definition *eq-finfun-def* [*code*]: *HOL.equal f g* ⟷ *finfun-All* ((λ(*x*, *y*). *x* = *y*) ∘$ ($*f*, *g*$))
instance by(*intro-classes*)(*simp add*: *eq-finfun-All-ext eq-finfun-def*)
end

lemma [*code nbe*]:
HOL.equal (*f* :: - ⇒*f* -) *f* ⟷ *True*
by (*fact equal-refl*)

1.16 An operator that explicitly removes all redundant updates in the generated representations

definition *finfun-clearjunk* :: ′*a* ⇒*f* ′*b* ⇒ ′*a* ⇒*f* ′*b*
where [*simp*, *code del*]: *finfun-clearjunk* = *id*

lemma *finfun-clearjunk-const* [*code*]: *finfun-clearjunk* (*K*$ *b*) = (*K*$ *b*)
by *simp*

lemma *finfun-clearjunk-update* [*code*]:
finfun-clearjunk (*finfun-update-code f a b*) = *f*(*a* $:= *b*)
by *simp*

1.17 The domain of a FinFun as a FinFun

definition *finfun-dom* :: (′*a* ⇒*f* ′*b*) ⇒ (′*a* ⇒*f bool*)
where [*code del*]: *finfun-dom f* = *Abs-finfun* (λ*a*. *f* $ *a* ≠ *finfun-default f*)

lemma *finfun-dom-const*:
finfun-dom ((*K\$ c*) :: *'a* ⇒*f* *'b*) = (*K\$ finite* (*UNIV* :: *'a set*) ∧ *c* ≠ *undefined*)
unfolding *finfun-dom-def finfun-default-const*
by(*auto*)(*simp-all add*: *finfun-const-def*)

finfun-dom raises an exception when called on a FinFun whose domain is a finite type. For such FinFuns, the default value (and as such the domain) is undefined.

lemma *finfun-dom-const-code* [*code*]:
finfun-dom ((*K\$ c*) :: (*'a* :: *card-UNIV*) ⇒*f* *'b*) =
(*if CARD*(*'a*) = *0 then* (*K\$ False*) *else Code.abort* (*STR ''finfun-dom called on finite type''*) (λ-. *finfun-dom* (*K\$ c*)))
by(*simp add*: *finfun-dom-const card-UNIV card-eq-0-iff*)

lemma *finfun-dom-finfunI*: (λ*a*. *f* \$ *a* ≠ *finfun-default f*) ∈ *finfun*
using *finite-finfun-default*[*of f*]
by(*simp add*: *finfun-def exI*[**where** *x*=*False*])

lemma *finfun-dom-update* [*simp*]:
finfun-dom (*f*(*a* \$:= *b*)) = (*finfun-dom f*)(*a* \$:= (*b* ≠ *finfun-default f*))
including *finfun* **unfolding** *finfun-dom-def finfun-update-def*
apply(*simp add*: *finfun-default-update-const finfun-dom-finfunI*)
apply(*fold finfun-update.rep-eq*)
apply(*simp add*: *finfun-upd-apply fun-eq-iff fun-upd-def finfun-default-update-const*)
done

lemma *finfun-dom-update-code* [*code*]:
finfun-dom (*finfun-update-code f a b*) = *finfun-update-code* (*finfun-dom f*) *a* (*b* ≠ *finfun-default f*)
by(*simp*)

lemma *finite-finfun-dom*: *finite* {*x*. *finfun-dom f* \$ *x*}
proof(*induct f rule*: *finfun-weak-induct*)
case (*const b*)
thus *?case*
by (*cases finite* (*UNIV* :: *'a set*) ∧ *b* ≠ *undefined*)
(*auto simp add*: *finfun-dom-const UNIV-def* [*symmetric*] *Set.empty-def* [*symmetric*])
next
case (*update f a b*)
have {*x*. *finfun-dom f*(*a* \$:= *b*) \$ *x*} =
(*if b* = *finfun-default f then* {*x*. *finfun-dom f* \$ *x*} − {*a*} *else insert a* {*x*. *finfun-dom f* \$ *x*})
by (*auto simp add*: *finfun-upd-apply split*: *if-split-asm*)
thus *?case* **using** *update* **by** *simp*
qed

1.18 The domain of a FinFun as a sorted list

definition *finfun-to-list* :: (*'a* :: *linorder*) ⇒*f* *'b* ⇒ *'a list*

where
finfun-to-list f = (*THE xs. set xs* = {*x. finfun-dom f* $ *x*} ∧ *sorted xs* ∧ *distinct xs*)

lemma *set-finfun-to-list* [*simp*]: *set* (*finfun-to-list f*) = {*x. finfun-dom f* $ *x*} (**is** *?thesis1*)
and *sorted-finfun-to-list*: *sorted* (*finfun-to-list f*) (**is** *?thesis2*)
and *distinct-finfun-to-list*: *distinct* (*finfun-to-list f*) (**is** *?thesis3*)
proof (*atomize* (*full*))
show *?thesis1* ∧ *?thesis2* ∧ *?thesis3*
unfolding *finfun-to-list-def*
by(*rule theI'*)(*rule finite-sorted-distinct-unique finite-finfun-dom*)+
qed

lemma *finfun-const-False-conv-bot*: ($) (*K*$ *False*) = *bot*
by *auto*

lemma *finfun-const-True-conv-top*: ($) (*K*$ *True*) = *top*
by *auto*

lemma *finfun-to-list-const*:
finfun-to-list ((*K*$ *c*) :: ('*a* :: {*linorder*} ⇒*f* '*b*)) =
(*if* ¬ *finite* (*UNIV* :: '*a set*) ∨ *c* = *undefined then* [] *else THE xs. set xs* = *UNIV* ∧ *sorted xs* ∧ *distinct xs*)
by(*auto simp add*: *finfun-to-list-def finfun-const-False-conv-bot finfun-const-True-conv-top finfun-dom-const*)

lemma *finfun-to-list-const-code* [*code*]:
finfun-to-list ((*K*$ *c*) :: ('*a* :: {*linorder*, *card-UNIV*} ⇒*f* '*b*)) =
(*if CARD*('*a*) = *0 then* [] *else Code.abort* (*STR* ''*finfun-to-list called on finite type*'') (λ-. *finfun-to-list* ((*K*$ *c*) :: ('*a* ⇒*f* '*b*))))
by(*auto simp add*: *finfun-to-list-const card-UNIV card-eq-0-iff*)

lemma *remove1-insort-insert-same*:
x ∉ *set xs* ⟹ *remove1 x* (*insort-insert x xs*) = *xs*
by (*metis insort-insert-insort remove1-insort-key*)

lemma *finfun-dom-conv*:
finfun-dom f $ *x* ⟷ *f* $ *x* ≠ *finfun-default f*
by(*induct f rule*: *finfun-weak-induct*)(*auto simp add*: *finfun-dom-const finfun-default-const finfun-default-update-const finfun-upd-apply*)

lemma *finfun-to-list-update*:
finfun-to-list (*f*(*a* $:= *b*)) =
(*if b* = *finfun-default f then List.remove1 a* (*finfun-to-list f*) *else List.insort-insert a* (*finfun-to-list f*))
proof(*subst finfun-to-list-def*, *rule the-equality*)
fix *xs*
assume *set xs* = {*x. finfun-dom f*(*a* $:= *b*) $ *x*} ∧ *sorted xs* ∧ *distinct xs*

hence *eq*: *set xs* = {*x*. *finfun-dom f*(*a* $:= *b*) $ *x*}
and [*simp*]: *sorted xs distinct xs* **by** *simp-all*
show *xs* = (*if b* = *finfun-default f then remove1 a* (*finfun-to-list f*) *else insort-insert a* (*finfun-to-list f*))
proof(*cases b* = *finfun-default f*)
case [*simp*]: *True*
show *?thesis*
proof(*cases finfun-dom f* $ *a*)
case *True*
have *finfun-to-list f* = *insort-insert a xs*
unfolding *finfun-to-list-def*
proof(*rule the-equality*)
have *set* (*insort-insert a xs*) = *insert a* (*set xs*) **by**(*simp add*: *set-insort-insert*)
also note *eq* **also**
have *insert a* {*x*. *finfun-dom f*(*a* $:= *b*) $ *x*} = {*x*. *finfun-dom f* $ *x*} **using** *True*
by(*auto simp add*: *finfun-upd-apply split*: *if-split-asm*)
finally show *1*: *set* (*insort-insert a xs*) = {*x*. *finfun-dom f* $ *x*} ∧ *sorted* (*insort-insert a xs*) ∧ *distinct* (*insort-insert a xs*)
by(*simp add*: *sorted-insort-insert distinct-insort-insert*)

fix *xs′*
assume *set xs′* = {*x*. *finfun-dom f* $ *x*} ∧ *sorted xs′* ∧ *distinct xs′*
thus *xs′* = *insort-insert a xs* **using** *1* **by**(*auto dest*: *sorted-distinct-set-unique*)
qed
with *eq True* **show** *?thesis* **by**(*simp add*: *remove1-insort-insert-same*)
next
case *False*
hence *f* $ *a* = *b* **by**(*auto simp add*: *finfun-dom-conv*)
hence *f*: *f*(*a* $:= *b*) = *f* **by**(*simp add*: *expand-finfun-eq fun-eq-iff finfun-upd-apply*)
from *eq* **have** *finfun-to-list f* = *xs* **unfolding** *f finfun-to-list-def*
by(*auto elim*: *sorted-distinct-set-unique intro*!: *the-equality*)
with *eq False* **show** *?thesis* **unfolding** *f* **by**(*simp add*: *remove1-idem*)
qed
next
case *False*
show *?thesis*
proof(*cases finfun-dom f* $ *a*)
case *True*
have *finfun-to-list f* = *xs*
unfolding *finfun-to-list-def*
proof(*rule the-equality*)
have *finfun-dom f* = *finfun-dom f*(*a* $:= *b*) **using** *False True*
by(*simp add*: *expand-finfun-eq fun-eq-iff finfun-upd-apply*)
with *eq* **show** *1*: *set xs* = {*x*. *finfun-dom f* $ *x*} ∧ *sorted xs* ∧ *distinct xs*
by(*simp del*: *finfun-dom-update*)

fix *xs′*

```
      assume set xs' = {x. finfun-dom f $ x} ∧ sorted xs' ∧ distinct xs'
      thus xs' = xs using 1 by(auto elim: sorted-distinct-set-unique)
    qed
    thus ?thesis using False True eq by(simp add: insort-insert-triv)
  next
    case False
    have finfun-to-list f = remove1 a xs
      unfolding finfun-to-list-def
    proof(rule the-equality)
      have set (remove1 a xs) = set xs − {a} by simp
      also note eq also
      have {x. finfun-dom f(a $:= b) $ x} − {a} = {x. finfun-dom f $ x} using
False
        by(auto simp add: finfun-upd-apply split: if-split-asm)
        finally show 1: set (remove1 a xs) = {x. finfun-dom f $ x} ∧ sorted
(remove1 a xs) ∧ distinct (remove1 a xs)
        by(simp add: sorted-remove1)

      fix xs'
      assume set xs' = {x. finfun-dom f $ x} ∧ sorted xs' ∧ distinct xs'
      thus xs' = remove1 a xs using 1 by(blast intro: sorted-distinct-set-unique)
    qed
    thus ?thesis using False eq ‹b ≠ finfun-default f›
      by (simp add: insort-insert-insort insort-remove1)
  qed
 qed
qed (auto simp add: distinct-finfun-to-list sorted-finfun-to-list sorted-remove1 set-insort-insert
sorted-insort-insert distinct-insort-insert finfun-upd-apply split: if-split-asm)

lemma finfun-to-list-update-code [code]:
 finfun-to-list (finfun-update-code f a b) =
 (if b = finfun-default f then List.remove1 a (finfun-to-list f) else List.insort-insert
a (finfun-to-list f))
by(simp add: finfun-to-list-update)
```

More type class instantiations

```
lemma card-eq-1-iff: card A = 1 ⟷ A ≠ {} ∧ (∀x∈A. ∀y∈A. x = y)
 (is ?lhs ⟷ ?rhs)
proof
 assume ?lhs
 moreover {
  fix x y
  assume A: x ∈ A y ∈ A and neq: x ≠ y
  have finite A using ‹?lhs› by(simp add: card-ge-0-finite)
  from neq have 2 = card {x, y} by simp
  also have . . . ≤ card A using A ‹finite A›
   by(auto intro: card-mono)
  finally have False using ‹?lhs› by simp }
 ultimately show ?rhs by auto
```

next
  assume *?rhs*
  hence $A = \{THE\ x.\ x \in A\}$
    by *safe* (*auto intro*: *theI the-equality*[*symmetric*])
  also have *card* ... = *1* **by** *simp*
  finally show *?lhs* **.**
qed

lemma *card-UNIV-finfun*:
  defines *F* == *finfun* :: ('*a* ⇒ '*b*) *set*
  shows *CARD*('*a* ⇒*f* '*b*) = (*if CARD*('*a*) ≠ *0* ∧ *CARD*('*b*) ≠ *0* ∨ *CARD*('*b*) =
1 then CARD('*b*) ^ *CARD*('*a*) *else 0*)
proof(*cases 0* < *CARD*('*a*) ∧ *0* < *CARD*('*b*) ∨ *CARD*('*b*) = *1*)
  case *True*
  from *True* **have** *F* = *UNIV*
  proof
    assume *b*: *CARD*('*b*) = *1*
    hence ∀ *x* :: '*b*. *x* = *undefined*
      by(*auto simp add*: *card-eq-1-iff simp del*: *One-nat-def*)
   thus *?thesis* **by**(*auto simp add*: *finfun-def F-def intro*: *exI*[**where** *x*=*undefined*])
 qed(*auto simp add*: *finfun-def card-gt-0-iff F-def intro*: *finite-subset*[**where** *B*=*UNIV*])
  moreover have *CARD*('*a* ⇒*f* '*b*) = *card F*
    unfolding *type-definition.Abs-image*[*OF type-definition-finfun*, *symmetric*]
    by(*auto intro*!: *card-image inj-onI simp add*: *Abs-finfun-inject F-def*)
  ultimately show *?thesis* **by**(*simp add*: *card-fun*)
next
  case *False*
  hence *infinite*: ¬ (*finite* (*UNIV* :: '*a set*) ∧ *finite* (*UNIV* :: '*b set*))
    and *b*: *CARD*('*b*) ≠ *1* **by**(*simp-all add*: *card-eq-0-iff*)
  from *b* **obtain** *b1 b2* :: '*b* **where** *b1* ≠ *b2*
    by(*auto simp add*: *card-eq-1-iff simp del*: *One-nat-def*)
  let *?f* = λ*a a'* :: '*a*. *if a* = *a' then b1 else b2*
  from *infinite* **have** ¬ *finite* (*UNIV* :: ('*a* ⇒*f* '*b*) *set*)
  proof(*rule contrapos-nn*[*OF - conjI*])
    assume *finite*: *finite* (*UNIV* :: ('*a* ⇒*f* '*b*) *set*)
    hence *finite F*
        unfolding *type-definition.Abs-image*[*OF type-definition-finfun*, *symmetric*]
F-def
      by(*rule finite-imageD*)(*auto intro*: *inj-onI simp add*: *Abs-finfun-inject*)
    hence *finite* (*range ?f*)
       by(*rule finite-subset*[*rotated 1*])(*auto simp add*: *F-def finfun-def* ‹*b1* ≠ *b2*›
intro!: *exI*[**where** *x*=*b2*])
    thus *finite* (*UNIV* :: '*a set*)
     by(*rule finite-imageD*)(*auto intro*: *inj-onI simp add*: *fun-eq-iff* ‹*b1* ≠ *b2*› *split*:
if-split-asm)

    from *finite* **have** *finite* (*range* (λ*b*. ((*K$ b*) :: '*a* ⇒*f* '*b*)))
      by(*rule finite-subset*[*rotated 1*]) *simp*
    thus *finite* (*UNIV* :: '*b set*)

by(*rule finite-imageD*)(*auto intro*!: *inj-onI*)
qed
with *False* **show** *?thesis* **by** *auto*
qed

lemma *finite-UNIV-finfun*:
finite (*UNIV* :: ('*a* ⇒*f* '*b*) *set*) ⟷
(*finite* (*UNIV* :: '*a set*) ∧ *finite* (*UNIV* :: '*b set*) ∨ *CARD*('*b*) = *1*)
(**is** *?lhs* ⟷ *?rhs*)
proof −
have *?lhs* ⟷ *CARD*('*a* ⇒*f* '*b*) > *0* **by**(*simp add*: *card-gt-0-iff*)
also have ... ⟷ *CARD*('*a*) > *0* ∧ *CARD*('*b*) > *0* ∨ *CARD*('*b*) = *1*
by(*simp add*: *card-UNIV-finfun*)
also have ... = *?rhs* **by**(*simp add*: *card-gt-0-iff*)
finally show *?thesis* **.**
qed

instantiation *finfun* :: (*finite-UNIV*, *card-UNIV*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*('*a* ⇒*f* '*b*)
(*let cb* = *of-phantom* (*card-UNIV* :: '*b card-UNIV*)
in cb = *1* ∨ *of-phantom* (*finite-UNIV* :: '*a finite-UNIV*) ∧ *cb* ≠ *0*)
instance
by *intro-classes* (*auto simp add*: *finite-UNIV-finfun-def Let-def card-UNIV finite-UNIV finite-UNIV-finfun card-gt-0-iff*)
end

instantiation *finfun* :: (*card-UNIV*, *card-UNIV*) *card-UNIV* **begin**
definition *card-UNIV* = *Phantom*('*a* ⇒*f* '*b*)
(*let ca* = *of-phantom* (*card-UNIV* :: '*a card-UNIV*);
cb = *of-phantom* (*card-UNIV* :: '*b card-UNIV*)
in if ca ≠ *0* ∧ *cb* ≠ *0* ∨ *cb* = *1 then cb* ^ *ca else 0*)
instance by *intro-classes* (*simp add*: *card-UNIV-finfun-def card-UNIV Let-def card-UNIV-finfun*)
end

1.18.1 Bundles for concrete syntax

bundle *finfun-syntax*
begin

type-notation *finfun* (‹(- ⇒*f* /-)› [*22*, *21*] *21*)

notation
finfun-const (‹*K*$/ -› [*0*] *1*) **and**
finfun-update (‹-'(- $:= -')› [*1000*, *0*, *0*] *1000*) **and**
finfun-apply (**infixl** ‹$› *999*) **and**
finfun-comp (**infixr** ‹∘$› *55*) **and**
finfun-comp2 (**infixr** ‹$∘› *55*) **and**
finfun-Diag (‹(*1*'($-,/ -$'))› [*0*, *0*] *1000*)

notation (*ASCII*)
finfun-comp (**infixr** ‹*o*$› *55*) **and**
finfun-comp2 (**infixr** ‹$*o*› *55*)

end

unbundle *no finfun-syntax*

end

2 Predicates modelled as FinFuns

theory *FinFunPred*
imports *FinFun*
begin

unbundle *finfun-syntax*

Instantiate FinFun predicates just like predicates

type-synonym $'a\ pred_f = 'a \Rightarrow f\ bool$

instantiation *finfun* :: (*type*, *ord*) *ord*
begin

definition *le-finfun-def* [*code del*]: $f \leq g \longleftrightarrow (\forall x.\ f\ \$\ x \leq g\ \$\ x)$

definition [*code del*]: $(f::'a \Rightarrow f\ 'b) < g \longleftrightarrow f \leq g \wedge \neg\ g \leq f$

instance ..

lemma *le-finfun-code* [*code*]:
$f \leq g \longleftrightarrow$ *finfun-All* $((\lambda(x, y).\ x \leq y) \circ\$\ (\$f, g\$))$
by(*simp add*: *le-finfun-def finfun-All-All o-def*)

end

instance *finfun* :: (*type*, *preorder*) *preorder*
by(*intro-classes*)(*auto simp add*: *less-finfun-def le-finfun-def intro*: *order-trans*)

instance *finfun* :: (*type*, *order*) *order*
by(*intro-classes*)(*auto simp add*: *le-finfun-def order-antisym-conv intro*: *finfun-ext*)

instantiation *finfun* :: (*type*, *order-bot*) *order-bot* **begin**
definition $bot = $ *finfun-const bot*
instance by(*intro-classes*)(*simp add*: *bot-finfun-def le-finfun-def*)
end

lemma *bot-finfun-apply* [*simp*]: $(\$)\ bot = (\lambda\text{-}.\ bot)$
by(*auto simp add*: *bot-finfun-def*)

instantiation *finfun* :: (*type*, *order-top*) *order-top* **begin**
definition *top* = *finfun-const top*
instance by(*intro-classes*)(*simp add*: *top-finfun-def le-finfun-def*)
end

lemma *top-finfun-apply* [*simp*]: ($) *top* = (λ-. *top*)
by(*auto simp add*: *top-finfun-def*)

instantiation *finfun* :: (*type*, *inf*) *inf* **begin**
definition [*code*]: *inf f g* = (λ(*x*, *y*). *inf x y*) ∘$ ($*f*, *g*$)
instance ..
end

lemma *inf-finfun-apply* [*simp*]: ($) (*inf f g*) = *inf* (($) *f*) (($) *g*)
by(*auto simp add*: *inf-finfun-def o-def inf-fun-def*)

instantiation *finfun* :: (*type*, *sup*) *sup* **begin**
definition [*code*]: *sup f g* = (λ(*x*, *y*). *sup x y*) ∘$ ($*f*, *g*$)
instance ..
end

lemma *sup-finfun-apply* [*simp*]: ($) (*sup f g*) = *sup* (($) *f*) (($) *g*)
by(*auto simp add*: *sup-finfun-def o-def sup-fun-def*)

instance *finfun* :: (*type*, *semilattice-inf*) *semilattice-inf*
by(*intro-classes*)(*simp-all add*: *inf-finfun-def le-finfun-def*)

instance *finfun* :: (*type*, *semilattice-sup*) *semilattice-sup*
by(*intro-classes*)(*simp-all add*: *sup-finfun-def le-finfun-def*)

instance *finfun* :: (*type*, *lattice*) *lattice* **..**

instance *finfun* :: (*type*, *bounded-lattice*) *bounded-lattice*
by(*intro-classes*)

instance *finfun* :: (*type*, *distrib-lattice*) *distrib-lattice*
by(*intro-classes*)(*simp add*: *sup-finfun-def inf-finfun-def expand-finfun-eq o-def sup-inf-distrib1*)

instantiation *finfun* :: (*type*, *minus*) *minus* **begin**
definition *f* − *g* = *case-prod* (−) ∘$ ($*f*, *g*$)
instance ..
end

lemma *minus-finfun-apply* [*simp*]: ($) (*f* − *g*) = ($) *f* − ($) *g*
by(*simp add*: *minus-finfun-def o-def fun-diff-def*)

instantiation *finfun* :: (*type*, *uminus*) *uminus* **begin**
definition − *A* = *uminus* ∘$ *A*

instance ..
end

lemma *uminus-finfun-apply* [*simp*]: ($) (− *g*) = − ($) *g*
by(*simp add*: *uminus-finfun-def o-def fun-Compl-def*)

instance *finfun* :: (*type*, *boolean-algebra*) *boolean-algebra*
by(*intro-classes*)
(*simp-all add*: *uminus-finfun-def inf-finfun-def expand-finfun-eq sup-fun-def inf-fun-def fun-Compl-def o-def inf-compl-bot sup-compl-top diff-eq*)

Replicate predicate operations for FinFuns

abbreviation *finfun-empty* :: *'a* $pred_f$ (‹$\{\}_f$›)
where $\{\}_f$ ≡ *bot*

abbreviation *finfun-UNIV* :: *'a* $pred_f$
where *finfun-UNIV* ≡ *top*

definition *finfun-single* :: *'a* ⇒ *'a* $pred_f$
where [*code*]: *finfun-single x* = *finfun-empty*(*x* $:= *True*)

lemma *finfun-single-apply* [*simp*]:
finfun-single x $ *y* ⟷ *x* = *y*
by(*simp add*: *finfun-single-def finfun-upd-apply*)

lemma [*iff*]:
shows *finfun-single-neq-bot*: *finfun-single x* ≠ *bot*
and *bot-neq-finfun-single*: *bot* ≠ *finfun-single x*
by(*simp-all add*: *expand-finfun-eq fun-eq-iff*)

lemma *finfun-leI* [*intro*!]: (!!*x*. *A* $ *x* ⟹ *B* $ *x*) ⟹ *A* ≤ *B*
by(*simp add*: *le-finfun-def*)

lemma *finfun-leD* [*elim*]: ⟦ *A* ≤ *B*; *A* $ *x* ⟧ ⟹ *B* $ *x*
by(*simp add*: *le-finfun-def*)

Bounded quantification. Warning: *finfun-Ball* and *finfun-Ex* may raise an exception, they should not be used for quickcheck

definition *finfun-Ball-except* :: *'a list* ⇒ *'a* $pred_f$ ⇒ (*'a* ⇒ *bool*) ⇒ *bool*
where [*code del*]: *finfun-Ball-except xs A P* = (∀ *a*. *A* $ *a* ⟶ *a* ∈ *set xs* ∨ *P a*)

lemma *finfun-Ball-except-const*:
finfun-Ball-except xs (*K*$ *b*) *P* ⟷ ¬ *b* ∨ *set xs* = *UNIV* ∨ *Code.abort* (*STR ''finfun-ball-except''*) (λ*u*. *finfun-Ball-except xs* (*K*$ *b*) *P*)
by(*auto simp add*: *finfun-Ball-except-def*)

lemma *finfun-Ball-except-const-finfun-UNIV-code* [*code*]:
finfun-Ball-except xs (*K*$ *b*) *P* ⟷ ¬ *b* ∨ *is-list-UNIV xs* ∨ *Code.abort* (*STR ''finfun-ball-except''*) (λ*u*. *finfun-Ball-except xs* (*K*$ *b*) *P*)

by(*auto simp add*: *finfun-Ball-except-def is-list-UNIV-iff*)

lemma *finfun-Ball-except-update*:
finfun-Ball-except xs (*A*(*a* $:= *b*)) *P* = ((*a* ∈ *set xs* ∨ (*b* ⟶ *P a*)) ∧ *finfun-Ball-except* (*a* # *xs*) *A P*)
by(*fastforce simp add*: *finfun-Ball-except-def finfun-upd-apply split*: *if-split-asm*)

lemma *finfun-Ball-except-update-code* [*code*]:
fixes *a* :: *'a* :: *card-UNIV*
shows *finfun-Ball-except xs* (*finfun-update-code f a b*) *P* = ((*a* ∈ *set xs* ∨ (*b* ⟶ *P a*)) ∧ *finfun-Ball-except* (*a* # *xs*) *f P*)
by(*simp add*: *finfun-Ball-except-update*)

definition *finfun-Ball* :: *'a* $pred_f$ ⇒ (*'a* ⇒ *bool*) ⇒ *bool*
where [*code del*]: *finfun-Ball A P* = *Ball* {*x*. *A* $ *x*} *P*

lemma *finfun-Ball-code* [*code*]: *finfun-Ball* = *finfun-Ball-except* []
by(*auto intro*!: *ext simp add*: *finfun-Ball-except-def finfun-Ball-def*)

definition *finfun-Bex-except* :: *'a list* ⇒ *'a* $pred_f$ ⇒ (*'a* ⇒ *bool*) ⇒ *bool*
where [*code del*]: *finfun-Bex-except xs A P* = (∃ *a*. *A* $ *a* ∧ *a* ∉ *set xs* ∧ *P a*)

lemma *finfun-Bex-except-const*:
finfun-Bex-except xs (*K*$ *b*) *P* ⟷ *b* ∧ *set xs* ≠ *UNIV* ∧ *Code.abort* (*STR ''finfun-Bex-except''*) (λ*u*. *finfun-Bex-except xs* (*K*$ *b*) *P*)
by(*auto simp add*: *finfun-Bex-except-def*)

lemma *finfun-Bex-except-const-finfun-UNIV-code* [*code*]:
finfun-Bex-except xs (*K*$ *b*) *P* ⟷ *b* ∧ ¬ *is-list-UNIV xs* ∧ *Code.abort* (*STR ''finfun-Bex-except''*) (λ*u*. *finfun-Bex-except xs* (*K*$ *b*) *P*)
by(*auto simp add*: *finfun-Bex-except-def is-list-UNIV-iff*)

lemma *finfun-Bex-except-update*:
finfun-Bex-except xs (*A*(*a* $:= *b*)) *P* ⟷ (*a* ∉ *set xs* ∧ *b* ∧ *P a*) ∨ *finfun-Bex-except* (*a* # *xs*) *A P*
by(*fastforce simp add*: *finfun-Bex-except-def finfun-upd-apply dest*: *bspec split*: *if-split-asm*)

lemma *finfun-Bex-except-update-code* [*code*]:
fixes *a* :: *'a* :: *card-UNIV*
shows *finfun-Bex-except xs* (*finfun-update-code f a b*) *P* ⟷ ((*a* ∉ *set xs* ∧ *b* ∧ *P a*) ∨ *finfun-Bex-except* (*a* # *xs*) *f P*)
by(*simp add*: *finfun-Bex-except-update*)

definition *finfun-Bex* :: *'a* $pred_f$ ⇒ (*'a* ⇒ *bool*) ⇒ *bool*
where [*code del*]: *finfun-Bex A P* = *Bex* {*x*. *A* $ *x*} *P*

lemma *finfun-Bex-code* [*code*]: *finfun-Bex* = *finfun-Bex-except* []
by(*auto intro*!: *ext simp add*: *finfun-Bex-except-def finfun-Bex-def*)

Automatically replace predicate operations by finfun predicate operations where possible

lemma *iso-finfun-le* [*code-unfold*]:
$(\$)\ A \leq (\$)\ B \longleftrightarrow A \leq B$
by (*metis le-finfun-def le-funD le-funI*)

lemma *iso-finfun-less* [*code-unfold*]:
$(\$)\ A < (\$)\ B \longleftrightarrow A < B$
by (*metis iso-finfun-le less-finfun-def less-fun-def*)

lemma *iso-finfun-eq* [*code-unfold*]:
$(\$)\ A = (\$)\ B \longleftrightarrow A = B$
by(*simp only*: *expand-finfun-eq*)

lemma *iso-finfun-sup* [*code-unfold*]:
sup $((\$)\ A)\ ((\$)\ B) = (\$)$ (*sup* $A\ B$)
by(*simp*)

lemma *iso-finfun-disj* [*code-unfold*]:
$A\ \$\ x \lor B\ \$\ x \longleftrightarrow$ *sup* $A\ B\ \$\ x$
by(*simp add*: *sup-fun-def*)

lemma *iso-finfun-inf* [*code-unfold*]:
inf $((\$)\ A)\ ((\$)\ B) = (\$)$ (*inf* $A\ B$)
by(*simp*)

lemma *iso-finfun-conj* [*code-unfold*]:
$A\ \$\ x \land B\ \$\ x \longleftrightarrow$ *inf* $A\ B\ \$\ x$
by(*simp add*: *inf-fun-def*)

lemma *iso-finfun-empty-conv* [*code-unfold*]:
$(\lambda\text{-}.\ \mathit{False}) = (\$)\ \{\}_f$
by *simp*

lemma *iso-finfun-UNIV-conv* [*code-unfold*]:
$(\lambda\text{-}.\ \mathit{True}) = (\$)$ *finfun-UNIV*
by *simp*

lemma *iso-finfun-upd* [*code-unfold*]:
fixes $A :: {'a}\ \mathit{pred}_f$
shows $((\$)\ A)(x := b) = (\$)\ (A(x\ \${:=}\ b))$
by(*simp add*: *fun-eq-iff*)

lemma *iso-finfun-uminus* [*code-unfold*]:
fixes $A :: {'a}\ \mathit{pred}_f$
shows $-\ (\$)\ A = (\$)\ (-\ A)$
by(*simp*)

lemma *iso-finfun-minus* [*code-unfold*]:

fixes A :: $'a\ pred_f$
shows ($) A − ($) B = ($) ($A$ − B)
by(*simp*)

Do not declare the following two theorems as [*code-unfold*], because this causes quickcheck to fail frequently when bounded quantification is used which raises an exception. For code generation, the same problems occur, but then, no randomly generated FinFun is usually around.

lemma *iso-finfun-Ball-Ball*:
(∀ *x*. *A* $ *x* ⟶ *P x*) ⟷ *finfun-Ball A P*
by(*simp add*: *finfun-Ball-def*)

lemma *iso-finfun-Bex-Bex*:
(∃ *x*. *A* $ *x* ∧ *P x*) ⟷ *finfun-Bex A P*
by(*simp add*: *finfun-Bex-def*)

Test code setup

notepad begin
have *inf* ((λ- :: *nat*. *False*)(*1* := *True*, *2* := *True*)) ((λ-. *True*)(*3* := *False*)) ≤
sup ((λ-. *False*)(*1* := *True*, *5* := *True*)) (− ((λ-. *True*)(*2* := *False*, *3* := *False*)))
by *eval*
end

declare *iso-finfun-Ball-Ball*[*code-unfold*]
notepad begin
have (∀ *x*. ((λ- :: *nat*. *False*)(*1* := *True*, *2* := *True*)) *x* ⟶ *x* < *3*)
by *eval*
end
declare *iso-finfun-Ball-Ball*[*code-unfold del*]

end