

Finfun

Andreas Lochbihler

May 26, 2024

Contents

1	Almost everywhere constant functions	1
1.1	The <i>map-default</i> operation	1
1.2	The finfun type	2
1.3	Kernel functions for type $'a \Rightarrow f 'b$	7
1.4	Code generator setup	7
1.5	Setup for quickcheck	7
1.6	<i>finfun-update</i> as instance of <i>comp-fun-commute</i>	7
1.7	Default value for FinFuns	8
1.8	Recursion combinator and well-formedness conditions	10
1.9	Weak induction rule and case analysis for FinFuns	19
1.10	Function application	20
1.11	Function composition	21
1.12	Universal quantification	23
1.13	A diagonal operator for FinFuns	24
1.14	Currying for FinFuns	27
1.15	Executable equality for FinFuns	28
1.16	An operator that explicitly removes all redundant updates in the generated representations	29
1.17	The domain of a FinFun as a FinFun	29
1.18	The domain of a FinFun as a sorted list	30
1.18.1	Bundles for concrete syntax	35
2	Predicates modelled as FinFuns	36

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 \rightarrow '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 = ($\lambda a.$ b)
by(simp add: *fun-eq-iff map-default-def*)

lemma *map-default-inject*:
fixes g g' :: 'a \rightarrow '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 ≠ 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*⇒'*b*. ∃ *b*. *finite* {*a*. *f a* ≠ *b*}}

typedef (*'a, 'b*) *finfun* ((- ⇒*f* /-) [22, 21] 21) = *finfun* :: (*'a* => '*b*) *set*
morphisms *finfun-apply* *Abs-finfun*

proof –
have ∃*f*. *finite* {*x*. *f x* ≠ *undefined*}
proof
show *finite* {*x*. (λ*y*. *undefined*) *x* ≠ *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) ∈ finfun* ↔ *y ∈ finfun*

proof –
{ **fix** *b'*
have *finite* {*a'*. (*y(a := b)*) *a' ≠ b'*} = *finite* {*a'*. *y a' ≠ b'*}
proof(*cases b = b'*)
case *True*
hence {*a'*. (*y(a := b)*) *a' ≠ b'*} = {*a'*. *y a' ≠ b'*} – {*a*} **by** *auto*
thus *?thesis* **by** *simp*
next
case *False*
hence {*a'*. (*y(a := b)*) *a' ≠ b'*} = *insert a* {*a'*. *y a' ≠ b'*} **by** *auto*
thus *?thesis* **by** *simp*
qed }
thus *?thesis* **unfolding** *finfun-def* **by** *blast*

qed

lemma *const-finfun*: $(\lambda x. a) \in \text{finfun}$
by(*auto simp add: finfun-def*)

lemma *finfun-left-compose*:

assumes $y \in \text{finfun}$

shows $g \circ y \in \text{finfun}$

proof –

from *assms* **obtain** b **where** $\text{finite } \{a. y a \neq b\}$

unfolding *finfun-def* **by** *blast*

hence $\text{finite } \{c. g (y c) \neq g b\}$

proof(*induct* $\{a. y a \neq b\}$ *arbitrary: y*)

case *empty*

hence $y = (\lambda a. b)$ **by**(*auto*)

thus *?case* **by**(*simp*)

next

case (*insert x F*)

note $IH = \langle \bigwedge y. F = \{a. y a \neq b\} \implies \text{finite } \{c. g (y c) \neq g b\} \rangle$

from $\langle \text{insert } x F = \{a. y a \neq b\} \rangle \langle x \notin F \rangle$

have $F: F = \{a. (y(x := b)) a \neq b\}$ **by**(*auto*)

show *?case*

proof(*cases* $g (y x) = g b$)

case *True*

hence $\{c. g ((y(x := b)) c) \neq g b\} = \{c. g (y c) \neq g b\}$ **by** *auto*

with $IH[OF F]$ **show** *?thesis* **by** *simp*

next

case *False*

hence $\{c. g (y c) \neq g b\} = \text{insert } x \{c. g ((y(x := b)) c) \neq g b\}$ **by** *auto*

with $IH[OF F]$ **show** *?thesis* **by**(*simp*)

qed

qed

thus *?thesis* **unfolding** *finfun-def* **by** *auto*

qed

lemma *assumes* $y \in \text{finfun}$

shows *fst-finfun*: $\text{fst} \circ y \in \text{finfun}$

and *snd-finfun*: $\text{snd} \circ y \in \text{finfun}$

proof –

from *assms* **obtain** $b c$ **where** $bc: \text{finite } \{a. y a \neq (b, c)\}$

unfolding *finfun-def* **by** *auto*

have $\{a. \text{fst } (y a) \neq b\} \subseteq \{a. y a \neq (b, c)\}$

and $\{a. \text{snd } (y a) \neq c\} \subseteq \{a. y a \neq (b, c)\}$ **by** *auto*

hence $\text{finite } \{a. \text{fst } (y a) \neq b\}$

and $\text{finite } \{a. \text{snd } (y a) \neq c\}$ **using** bc **by**(*auto intro: finite-subset*)

thus $\text{fst} \circ y \in \text{finfun}$ $\text{snd} \circ y \in \text{finfun}$

unfolding *finfun-def* **by** *auto*

qed

lemma *map-of-finfun*: $\text{map-of } xs \in \text{finfun}$
unfolding *finfun-def*
by(*induct xs*)(*auto simp add: Collect-neg-eq Collect-conj-eq Collect-imp-eq intro: finite-subset*)

lemma *Diag-finfun*: $(\lambda x. (f x, g x)) \in \text{finfun} \iff f \in \text{finfun} \wedge g \in \text{finfun}$
by(*auto intro: finite-subset simp add: Collect-neg-eq Collect-imp-eq Collect-conj-eq finfun-def*)

lemma *finfun-right-compose*:
assumes $g: g \in \text{finfun}$ **and** $\text{inj}: \text{inj } f$
shows $g \circ f \in \text{finfun}$
proof –
from g **obtain** b **where** $b: \text{finite } \{a. g a \neq b\}$ **unfolding** *finfun-def* **by** *blast*
moreover **have** $f \text{ ' } \{a. g (f a) \neq b\} \subseteq \{a. g a \neq b\}$ **by** *auto*
moreover **from** inj **have** $\text{inj-on } f \{a. g (f a) \neq b\}$ **by**(*rule subset-inj-on*) *blast*
ultimately **have** $\text{finite } \{a. g (f a) \neq b\}$
by(*blast intro: finite-imageD[where f=f] finite-subset*)
thus *?thesis* **unfolding** *finfun-def* **by** *auto*
qed

lemma *finfun-curry*:
assumes $\text{fin}: f \in \text{finfun}$
shows $\text{curry } f \in \text{finfun}$ $\text{curry } f a \in \text{finfun}$
proof –
from fin **obtain** c **where** $c: \text{finite } \{ab. f ab \neq c\}$ **unfolding** *finfun-def* **by** *blast*
moreover **have** $\{a. \exists b. f (a, b) \neq c\} = \text{fst ' } \{ab. f ab \neq c\}$ **by**(*force*)
hence $\{a. \text{curry } f a \neq (\lambda b. c)\} = \text{fst ' } \{ab. f ab \neq c\}$
by(*auto simp add: curry-def fun-eq-iff*)
ultimately **have** $\text{finite } \{a. \text{curry } f a \neq (\lambda b. c)\}$ **by** *simp*
thus $\text{curry } f \in \text{finfun}$ **unfolding** *finfun-def* **by** *blast*

have $\text{snd ' } \{ab. f ab \neq c\} = \{b. \exists a. f (a, b) \neq c\}$ **by**(*force*)
hence $\{b. f (a, b) \neq c\} \subseteq \text{snd ' } \{ab. f ab \neq c\}$ **by** *auto*
hence $\text{finite } \{b. f (a, b) \neq c\}$ **by**(*rule finite-subset*)(*rule finite-imageI[OF c]*)
thus $\text{curry } f a \in \text{finfun}$ **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-funfun-inject-finite*:

fixes $x\ y :: 'a \Rightarrow 'b$

assumes $fin: finite\ (UNIV :: 'a\ set)$

shows $Abs-funfun\ x = Abs-funfun\ y \longleftrightarrow x = y$

proof

assume $Abs-funfun\ x = Abs-funfun\ y$

moreover have $x \in finfun\ y \in finfun$ **unfolding** *finfun-def*

by(*auto intro: finite-subset[OF - fin]*)

ultimately show $x = y$ **by**(*simp add: Abs-funfun-inject*)

qed *simp*

lemma *Abs-funfun-inject-finite-class*:

fixes $x\ y :: ('a :: finite) \Rightarrow 'b$

shows $Abs-funfun\ x = Abs-funfun\ y \longleftrightarrow x = y$

using *finite-UNIV*

by(*simp add: Abs-funfun-inject-finite*)

lemma *Abs-funfun-inj-finite*:

assumes $fin: finite\ (UNIV :: 'a\ set)$

shows $inj\ (Abs-funfun :: ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow f\ 'b)$

proof(*rule inj-onI*)

fix $x\ y :: 'a \Rightarrow 'b$

assume $Abs-funfun\ x = Abs-funfun\ y$

moreover have $x \in finfun\ y \in finfun$ **unfolding** *finfun-def*

by(*auto intro: finite-subset[OF - fin]*)

ultimately show $x = y$ **by**(*simp add: Abs-funfun-inject*)

qed

lemma *Abs-funfun-inverse-finite*:

fixes $x :: 'a \Rightarrow 'b$

assumes $fin: finite\ (UNIV :: 'a\ set)$

shows $finfun-apply\ (Abs-funfun\ x) = x$

including *finfun*

proof –

from *fin* **have** $x \in finfun$

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

thus *?thesis* **by** *simp*

qed

lemma *Abs-funfun-inverse-finite-class*:

fixes $x :: ('a :: finite) \Rightarrow 'b$

shows $finfun-apply\ (Abs-funfun\ x) = x$

using *finite-UNIV* **by**(*simp add: Abs-funfun-inverse-finite*)

lemma *finfun-eq-finite-UNIV*: $finite\ (UNIV :: 'a\ set) \implies (finfun :: ('a \Rightarrow 'b)\ set) = UNIV$

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

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

lemma *map-default-in-finfun*:

assumes *fin*: finite (*dom f*)

shows *map-default b f* \in *finfun*

unfolding *finfun-def*

proof(*intro CollectI exI*)

from *fin* **show** finite {*a. map-default b f a* \neq *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* \notin *ran g*

proof –

obtain *y* **where** *f*: *f* = *Abs-finfun y* **and** *y*: *y* \in *finfun* **by**(*cases f*)

from *y* **obtain** *b* **where** *b*: finite {*a. y a* \neq *b*} **unfolding** *finfun-def* **by** *auto*

let *?g* = ($\lambda a.$ 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* \notin *ran ?g* **by**(*auto simp add: ran-def*)

ultimately show *?thesis* **by**(*rule that*)

qed

1.3 Kernel functions for type 'a \Rightarrow_f 'b

lift-definition *finfun-const* :: 'b \Rightarrow 'a \Rightarrow_f 'b (*K\$ / - [0] 1*)

is $\lambda b x. b$ **by** (*rule const-finfun*)

lift-definition *finfun-update* :: 'a \Rightarrow_f 'b \Rightarrow 'a \Rightarrow 'b \Rightarrow 'a \Rightarrow_f 'b (*-'(- \$:= -)*)

[*1000,0,0*] *1000*) **is** *fun-upd*

by (*simp add: fun-upd-finfun*)

lemma *finfun-update-twist*: *a* \neq *a'* \Longrightarrow *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 \Rightarrow_f 'b \Rightarrow 'a \Rightarrow 'b \Rightarrow 'a \Rightarrow_f 'b

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

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

lemma *finfun-update-const-code* [code]:

$(K\$ b)(a \$:= b') = (if\ b = b'\ then\ (K\$ b)\ else\ finfun-update-code\ (K\$ b)\ a\ b')$
by (*simp add: finfun-update-const-same*)

lemma *finfun-update-update-code* [code]:

$(finfun-update-code\ f\ a\ b)(a'\ \$:= b') = (if\ a = a'\ then\ f(a\ \$:= b')\ else\ finfun-update-code\ (f(a'\ \$:= b'))\ a\ b)$
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 $(finfun-apply\ b)(a := b', a' := b') = (finfun-apply\ b)(a' := b', a := b')$

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 $Finite-Set.fold\ (\lambda a\ f.\ f(a\ \$:= b'))\ (K\$ b)\ (UNIV :: 'a\ set) = (K\$ b)$

proof –

{ **fix** $A :: 'a\ set$

from *fin* **have** *finite A* **by** (*auto intro: finite-subset*)

hence $Finite-Set.fold\ (\lambda a\ f.\ f(a\ \$:= b'))\ (K\$ b)\ A = Abs-finfun\ (\lambda a.\ if\ a \in A\ then\ b'\ else\ b)$

proof (*induct*)

case (*insert x F*)

have $(\lambda a.\ if\ a = x\ then\ b'\ else\ (if\ a \in F\ then\ b'\ else\ b)) = (\lambda a.\ if\ a = x \vee a \in F\ then\ b'\ else\ b)$


```

    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 ⇒ 'b) ⇒ '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 ⇒ '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 ⇒ '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 ⇒ 'b

```

assumes $fin: f \in finfun$
shows $finfun\text{-default-}aux (f(a := b)) = finfun\text{-default-}aux f$
proof(*cases finite (UNIV :: 'a set)*)
case *False*
from fin **obtain** b' **where** $b': finite \{a. f a \neq b'\}$ **unfolding** $finfun\text{-def}$ **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 $finfun\text{-default} :: 'a \Rightarrow f 'b \Rightarrow 'b$
is $finfun\text{-default-}aux$.

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

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

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

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

lemma $finfun\text{-default-}update\text{-code}$ [*code*]:
 $finfun\text{-default} (finfun\text{-update-code } f a b) = finfun\text{-default } f$

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* \equiv
let *b* = *finfun-default* *f*;
g = *THE* *g*. *f* = *Abs-finfun* (*map-default* *b* *g*) \wedge *finite* (*dom* *g*) \wedge *b* \notin *ran* *g*
in *Finite-Set.fold* (λa . *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'* \implies *upd* *a* *b* (*upd* *a'* *b'* *c*) = *upd* *a'* *b'* (*upd* *a* *b* *c*)

and *upd-idemp*: *b* \neq *b'* \implies *upd* *a* *b''* (*upd* *a* *b'* (*cnst* *b*)) = *upd* *a* *b''* (*cnst* *b*)

begin

lemma *upd-left-comm*: *comp-fun-commute* (λa . *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* \neq *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* \notin *dom* *f*

and *fg*: *f* \subseteq_m *g*

shows *upd* *a* *d* (*Finite-Set.fold* (λa . *upd* *a* (*map-default* *d* *g* *a*)) (*cnst* *d*) (*dom* *f*)) =

Finite-Set.fold (λa . *upd* *a* (*map-default* *d* *g* *a*)) (*cnst* *d*) (*dom* *f*)

proof –

let *?upd* = λa . *upd* *a* (*map-default* *d* *g* *a*)

let *?fr* = λA . *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 $\langle \{\} = \text{dom } f \rangle$ **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* = $\langle \wedge f. \llbracket A = \text{dom } f; a \notin \text{dom } f; f \subseteq_m g \rrbracket \implies \text{upd } a \text{ } d \text{ } (?fr \text{ } (\text{dom } f)) = ?fr \text{ } (\text{dom } f) \rangle$

note *fin* = $\langle \text{finite } A \rangle$ **note** *anf* = $\langle a \notin \text{dom } f \rangle$ **note** *a'nA* = $\langle a' \notin A \rangle$

note *domf* = $\langle \text{insert } a' \text{ } A = \text{dom } f \rangle$ **note** *fg* = $\langle f \subseteq_m g \rangle$

```

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 ≠ a' by auto note upd-commute[OF this]
also from domf a'nA anf fg have a ∉ dom ?f' ?f' ⊆m g and A: A = dom ?f'
by(auto simp add: ran-def map-le-def)
note A also note IH[OF A ⟨a ∉ dom ?f'⟩ ⟨?f' ⊆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 ∉ dom f
and fg: f ⊆m g
shows upd a d'' (upd a d' (Finite-Set.fold (λa. upd a (map-default d g a)) (cnst
d) (dom f))) =
  upd a d'' (Finite-Set.fold (λa. upd a (map-default d g a)) (cnst d) (dom f))
proof –
let ?upd = λa. upd a (map-default d g a)
let ?fr = λA. 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 = ⟨∧f. [A = dom f; a ∉ dom f; f ⊆_m g] ⇒ upd a d'' (upd a d' (?fr
(dom f))) = upd a d'' (?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 ⊆_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 ⇒ d | Some b ⇒ 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** $a \notin \text{dom } ?f'$ $?f' \subseteq_m g$ **and** $A: A = \text{dom } ?f'$
by *(auto simp add: ran-def map-le-def)*
note A **also note** *IH[OF A ⟨a ∉ dom ?f'⟩ ⟨?f' ⊆_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 ≠ 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 : $\bigwedge a. a \in A \implies f a = g a$
shows $Finite\text{-Set.fold } f z A = Finite\text{-Set.fold } g z A$
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 $B \subseteq A \implies Finite\text{-Set.fold } f z B = Finite\text{-Set.fold } g z B$
proof *(induct)*
case *empty* **thus** *?case* **by** *simp*
next
case *(insert a B)*
note $finB = \langle finite B \rangle$ **note** $anB = \langle a \notin B \rangle$ **note** $sub = \langle insert a B \subseteq A \rangle$
note $IH = \langle B \subseteq A \implies Finite\text{-Set.fold } f z B = Finite\text{-Set.fold } g z B \rangle$
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** $Finite\text{-Set.fold } f z B = Finite\text{-Set.fold } g z B$ **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 =  $\lambda f g. f = \text{Abs-funfun} (\text{map-default } b \ g) \wedge \text{finite} (\text{dom } g) \wedge b \notin \text{ran } g$ 
    obtain g where g: g = The (?the f) by blast
    obtain y where f: f = Abs-funfun y and y: y  $\in$  finfun by (cases f)
    from f y b have bfin: finite {a. y a  $\neq$  b} by(simp add: finfun-default-def fi-
    nite-funfun-default-aux)

    let ?g = ( $\lambda a. \text{Some } (y \ a)$ ) |' {a. y a  $\neq$  b}
    from bfin have fing: finite (dom ?g) by auto
    have bran: b  $\notin$  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  $\notin$  ran g'
      and eq: Abs-funfun (map-default b ?g) = Abs-funfun (map-default b g') using
      f yg by auto
      from fin' fing have map-default b ?g  $\in$  finfun map-default b g'  $\in$  finfun by(blast
      intro: map-default-in-funfun)+
      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' = ( $\lambda a. \text{Some } ((y(a' := b)) \ a)$ ) |' {a. (y(a' := b)) a  $\neq$  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  $\notin$  ran ?g' by(auto simp add: ran-def restrict-map-def)

      let ?b' =  $\lambda a. \text{case } ?g' \ a \ \text{of } \text{None} \Rightarrow b \mid \text{Some } b \Rightarrow b$ 
      let ?b = map-default b ?g
      from upd-left-comm upd-left-comm fing'
      have Finite-Set.fold ( $\lambda a. \text{upd } a \ (?b' \ a)$ ) (cnst b) (dom ?g') = Finite-Set.fold
      ( $\lambda a. \text{upd } a \ (?b \ a)$ ) (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  $\lambda a. \text{upd } a \ (?b \ a)$  by(rule upd-left-comm)
      have Finite-Set.fold ( $\lambda a. \text{upd } a \ (?b \ a)$ ) (cnst b) (dom ?g') = upd a' b' (Finite-Set.fold

```

```

( $\lambda a. \text{upd } a \text{ } (?b \ a)$ ) ( $\text{cnst } b$ ) ( $\text{dom } ?g$ )
  proof ( $\text{cases } y \ a' = b$ )
    case True
      with  $b'b$  have  $g': ?g' = ?g$  by (auto simp add: restrict-map-def)
      from True have  $a' \text{ndomg}: a' \notin \text{dom } ?g$  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  $\text{domg}: \text{dom } ?g = \text{insert } a' \ (\text{dom } ?g')$  by auto
      from False  $b'b$  have  $a' \text{ndomg}': a' \notin \text{dom } ?g'$  by auto
      have  $\text{Finite-Set.fold } (\lambda a. \text{upd } a \text{ } (?b \ a)) \ (\text{cnst } b) \ (\text{insert } a' \ (\text{dom } ?g')) =$ 
         $\text{upd } a' \ (?b \ a') \ (\text{Finite-Set.fold } (\lambda a. \text{upd } a \text{ } (?b \ a)) \ (\text{cnst } b) \ (\text{dom } ?g'))$ 
        using  $\text{fing}' \ a' \text{ndomg}'$  unfolding  $b'b$  by (rule gwf.fold-insert)
      hence  $\text{upd } a' \ b \ (\text{Finite-Set.fold } (\lambda a. \text{upd } a \text{ } (?b \ a)) \ (\text{cnst } b) \ (\text{insert } a' \ (\text{dom } ?g')))) =$ 
         $\text{upd } a' \ b \ (\text{upd } a' \ (?b \ a') \ (\text{Finite-Set.fold } (\lambda a. \text{upd } a \text{ } (?b \ a)) \ (\text{cnst } b) \ (\text{dom } ?g')))$ 
        by simp
      also from  $b'b$  have  $g' \text{leg}: ?g' \subseteq_m ?g$  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  $f \ y \ b \ b'b \ \text{brang}' \ \text{fing}'$  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  $\text{fin}': \text{finite } (\text{dom } g')$  and  $\text{ran}': b \notin \text{ran } g'$ 
      and  $\text{eq}: f(a' \$:= b') = \text{Abs-funfun } (\text{map-default } b \ g')$ 
      by (auto simp del: fun-upd-apply)
    from  $\text{fin}' \ \text{fing}'$  have  $\text{map-default } b \ g' \in \text{finfun } \text{map-default } b \ ?g' \in \text{finfun}$ 
      by (blast intro: map-default-in-finfun)
    with  $\text{eq} \ f \ b'b \ b$  have  $\text{map-default } b \ ?g' = \text{map-default } b \ g'$ 
      by (simp del: fun-upd-apply add: finfun-update-def)
    with  $\text{fing}' \ \text{brang}' \ \text{fin}' \ \text{ran}'$  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  $b \ \text{fin} \ b'b \ b$ 
  by (simp only: finfun-default-update-const map-default-def)
next
  case False
  note  $b'b = \text{this}$ 
  let  $?g' = ?g(a' \mapsto b')$ 

```

```

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 fmg': 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-funfun (map-default b ?g') by (auto
simp add: funfun-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: funfun-update-def restrict-map-def)
  next
  fix g' assume ?the (f(a' $:= b')) g'
  hence f': f(a' $:= b') = Abs-funfun (map-default b g')
  and fin': finite (dom g') and brang': b ∉ ran g' by auto
  from fing' fin' have map-default b ?g' ∈ funfun map-default b g' ∈ funfun
  by (auto intro: map-default-in-funfun)
  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 (((λa. Some (y a)) |' {a. y a ≠ b})(a' ↦ b')) = insert a' (dom
((λa. 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-funfun (map-default b (?g' |' dom ?g)) using f by simp
  interpret g'wf: comp-fun-commute λa. upd a (?b' a) by (rule upd-left-comm)
  from upd-left-comm upd-left-comm fing
  have Finite-Set.fold (λa. upd a (?b' a)) (cst b) (dom ?g) = Finite-Set.fold
(λa. upd a (?b' a)) (cst b) (dom ?g)
  by (rule finite-rec-cong1)(auto simp add: restrict-map-def b'b True map-default-def)
  thus ?thesis unfolding funfun-rec-def Let-def funfun-default-update-const
b[symmetric]
  unfolding g' g[symmetric] gg g'wf.fold-insert[OF fing a'ndomg, of cst 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'':  $a' \notin \text{dom } ?g''$  by auto
have finng'': finite ( $\text{dom } ?g''$ ) by(rule finite-subset[OF - finng]) auto
have bnrang'':  $b \notin \text{ran } ?g''$  by(auto simp add: ran-def restrict-map-def)
interpret gwf: comp-fun-commute  $\lambda a. \text{upd } a \text{ } (?b \ a)$  by(rule upd-left-comm)
interpret g'wf: comp-fun-commute  $\lambda a. \text{upd } a \text{ } (?b' \ a)$  by(rule upd-left-comm)
have  $\text{upd } a' \ b' \ (Finite-Set.fold \ (\lambda a. \text{upd } a \text{ } (?b \ a)) \ (cnst \ b) \ (\text{insert } a' \ (\text{dom } ?g'')))$  =
       $\text{upd } a' \ b' \ (\text{upd } a' \ (?b \ a') \ (Finite-Set.fold \ (\lambda a. \text{upd } a \text{ } (?b \ a)) \ (cnst \ b) \ (\text{dom } ?g')))$ 
unfolding gwf.fold-insert[OF finng'' a'ndomg''] f ..
also have g''leg:  $?g \mid' \text{dom } ?g'' \subseteq_m ?g$  by(auto simp add: map-le-def)
have  $\text{dom } (?g \mid' \text{dom } ?g') = \text{dom } ?g''$  by auto
note map-default-update-twice[where  $d=b$  and  $f = ?g \mid' \text{dom } ?g''$  and  $a=a'$  and  $d'=?b \ a'$  and  $d''=b'$  and  $g=?g$ ,
      unfolded this, OF finng'' a'ndomg'' g''leg]
also have b':  $b' = ?b' \ a'$  by(auto simp add: map-default-def)
from upd-left-comm upd-left-comm finng''
have  $Finite-Set.fold \ (\lambda a. \text{upd } a \text{ } (?b \ a)) \ (cnst \ b) \ (\text{dom } ?g'') =$ 
       $Finite-Set.fold \ (\lambda a. \text{upd } a \text{ } (?b' \ a)) \ (cnst \ b) \ (\text{dom } ?g'')$ 
by(rule finite-rec-cong1)(auto simp add: restrict-map-def b'b map-default-def)
with b' have  $\text{upd } a' \ b' \ (Finite-Set.fold \ (\lambda a. \text{upd } a \text{ } (?b \ a)) \ (cnst \ b) \ (\text{dom } ?g'')) =$ 
       $\text{upd } a' \ (?b' \ a') \ (Finite-Set.fold \ (\lambda a. \text{upd } a \text{ } (?b' \ a)) \ (cnst \ b) \ (\text{dom } ?g''))$ 
by simp
also note g'wf.fold-insert[OF finng'' a'ndomg'', symmetric]
finally have  $\text{upd } a' \ b' \ (Finite-Set.fold \ (\lambda a. \text{upd } a \text{ } (?b \ a)) \ (cnst \ b) \ (\text{dom } ?g))$ 
=
       $Finite-Set.fold \ (\lambda a. \text{upd } a \text{ } (?b' \ a)) \ (cnst \ b) \ (\text{dom } ?g)$ 
unfolding domg . }
ultimately have  $Finite-Set.fold \ (\lambda a. \text{upd } a \text{ } (?b' \ a)) \ (cnst \ b) \ (\text{insert } a' \ (\text{dom } ?g)) =$ 
       $\text{upd } a' \ b' \ (Finite-Set.fold \ (\lambda a. \text{upd } a \text{ } (?b \ a)) \ (cnst \ b) \ (\text{dom } ?g))$ 
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:
       $finite \ (UNIV :: 'a \ set) \implies Finite-Set.fold \ (\lambda a. \text{upd } a \ b') \ (cnst \ b) \ (UNIV :: 'a \ set) = cnst \ b'$ 
begin

lemma finfun-rec-const [simp]:  $finfun-rec \ cnst \ \text{upd} \ (K\$ \ c) = cnst \ c$ 

```

```

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 = (λa. c)
      by(simp add: finfun-const-def)
      moreover have map-default c Map.empty = (λa. 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 = λg :: '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' = (λa. 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 = ( $\lambda a :: 'a$ . Some c)
proof (rule the-equality)
  from False True show ?the ( $\lambda a :: 'a$ . Some c)
  by(auto simp add: map-default-def [abs-def] finfun-const-def dom-def ran-def)
next
  fix g' ::  $'a \rightarrow 'b$ 
  assume ?the g'
  hence fg: ( $K \$ c$ ) = Abs-funfun (map-default undefined g')
    and fin: finite (dom g') and g: undefined  $\notin$  ran g' by simp-all
from fin have map-default undefined g' \in finfun by(rule map-default-in-funfun)
with fg have map-default undefined g' = (\lambda a. c)
  by(auto simp add: finfun-const-def intro: Abs-funfun-inject [THEN iffD1])
with True False show g' = (\lambda a::'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:  $\bigwedge b. P (K \$ b)$ 
and update:  $\bigwedge f a b. P f \implies P (f(a \$:= b))$ 
shows  $P x$ 
including finfun
proof(induct x rule: Abs-funfun-induct)
case (Abs-funfun y)
then obtain b where finite {a. y a \neq b} unfolding finfun-def by blast
thus ?case using  $\langle y \in \text{finfun} \rangle$ 
proof(induct {a. y a \neq b} arbitrary: y rule: finite-induct)
case empty
  hence  $\bigwedge a. y a = b$  by blast
  hence  $y = (\lambda a. b)$  by(auto)
  hence Abs-funfun y = finfun-const b unfolding finfun-const-def by simp
  thus ?case by(simp add: const)
next
case (insert a A)
note  $IH = \langle \bigwedge y. \llbracket A = \{a. y a \neq b\}; y \in \text{finfun} \rrbracket \implies P (Abs-funfun y) \rangle$ 
note  $y = \langle y \in \text{finfun} \rangle$ 
with  $\langle \text{insert } a A = \{a. y a \neq b\} \rangle \langle a \notin A \rangle$ 
have  $A = \{a'. (y(a := b)) a' \neq b\} \cup \{a\}$  by auto
  from  $IH[OF \text{this}]$  have  $P (finfun-update (Abs-funfun (y(a := b))) a (y a))$ 
by(rule update)

```

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

lemma *finfun-exhaust-disj*: $(\exists b. x = \text{finfun-const } b) \vee (\exists f a b. x = \text{finfun-update } f a b)$
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 $f :: 'a \Rightarrow f 'b \Rightarrow 'c$
assumes $c: \bigwedge c. f (K\$ c) = \text{cnst } c$
and $u: \bigwedge g a b. f (g(a \$:= b)) = \text{upd } g a b (f g)$
and $c': \bigwedge c. f' (K\$ c) = \text{cnst } c$
and $u': \bigwedge g a b. f' (g(a \$:= b)) = \text{upd } g a b (f' g)$
shows $f = f'$

proof
fix $g :: 'a \Rightarrow f 'b$
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* $\lambda b. b \lambda a' b c. \text{if } (a = a') \text{ then } b \text{ else } c$
by(*unfold-locales*) *auto*

interpretation *finfun-apply*: *finfun-rec-wf* $\lambda b. b \lambda a' b c. \text{if } (a = a') \text{ then } b \text{ else } c$
proof(*unfold-locales*)

fix $b' b :: 'a$
assume $\text{fin}: \text{finite } (UNIV :: 'b \text{ set})$
{ **fix** $A :: 'b \text{ set}$
interpret *comp-fun-commute* $\lambda a'. \text{If } (a = a') b'$ **by**(*rule finfun-apply-aux.upd-left-comm*)
from fin **have** *finite* A **by**(*auto intro: finite-subset*)
hence $\text{Finite-Set.fold } (\lambda a'. \text{If } (a = a') b') b A = (\text{if } a \in A \text{ then } b' \text{ else } b)$
by *induct auto* **}**
from *this*[*of UNIV*] **show** $\text{Finite-Set.fold } (\lambda a'. \text{If } (a = a') b') b UNIV = b'$ **by**
simp
qed

lemma *finfun-apply-def*: $(\$) = (\lambda f a. \text{finfun-rec } (\lambda b. b) (\lambda a' b c. \text{if } (a = a') \text{ then } b \text{ else } c) f)$

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

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

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

lemma *finfun-ext*: $(\bigwedge a. f \$ a = g \$ a) \implies f = g$
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*]: $f(x \text{\$} := f \$ x) = f$
by(*simp add: expand-finfun-eq fun-eq-iff finfun-upd-apply*)

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

lemma *finfun-const-eq-update*:
 $((K\$ b) = f(a \text{\$} := b')) = (b = b' \wedge (\forall a'. a \neq a' \implies f \$ a' = b))$
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** $\circ\$$ 55)
where [*code del*]: $g \circ\$ f = \text{finfun-rec } (\lambda b. (K\$ g \ b)) (\lambda a \ b \ c. c(a \text{\$} := g \ b)) f$

notation (*ASCII*)
 finfun-comp (**infixr** $\circ\$$ 55)

interpretation *finfun-comp-aux*: $\text{finfun-rec-wf-aux } (\lambda b. (K\$ g \ b)) (\lambda a \ b \ c. c(a \text{\$} := g \ b))$

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

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

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* ($\lambda(a :: 'c) c. c(a \$:= g b')$) ($K\$ g b$) $A =$

Abs-finfun ($\lambda a. \text{if } a \in A \text{ then } g b' \text{ else } g b$)

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* ($\lambda(a :: 'c) c. c(a \$:= g b')$) ($K\$ g b$) $UNIV = (K\$ g b')$

by(*simp add: finfun-const-def*)

qed

lemma *finfun-comp-const* [*simp, code*]:

$g \circ\$ (K\$ c) = (K\$ g c)$

by(*simp add: finfun-comp-def*)

lemma *finfun-comp-update* [*simp*]: $g \circ\$ (f(a \$:= b)) = (g \circ\$ f)(a \$:= g b)$

and *finfun-comp-update-code* [*code*]:

$g \circ\$ (\text{finfun-update-code } f a b) = \text{finfun-update-code } (g \circ\$ f) a (g b)$

by(*simp-all add: finfun-comp-def*)

lemma *finfun-comp-apply* [*simp*]:

$(\$) (g \circ\$ f) = g \circ (\$) f$

by(*induct f rule: finfun-weak-induct*)(*auto simp add: finfun-upd-apply*)

lemma *finfun-comp-comp-collapse* [*simp*]: $f \circ\$ g \circ\$ h = (f \circ g) \circ\$ h$

by(*induct h rule: finfun-weak-induct simp-all*)

lemma *finfun-comp-const1* [*simp*]: $(\lambda x. c) \circ\$ f = (K\$ c)$

by(*induct f rule: finfun-weak-induct*)(*auto intro: finfun-ext simp add: finfun-upd-apply*)

lemma *finfun-comp-id1* [*simp*]: $(\lambda x. x) \circ\$ f = f \text{ id} \circ\$ f = f$

by(*induct f rule: finfun-weak-induct auto*)

lemma *finfun-comp-conv-comp*: $g \circ\$ f = \text{Abs-finfun } (g \circ (\$) f)$

including *finfun*

proof –

have ($\lambda f. g \circ\$ f$) = ($\lambda f. \text{Abs-finfun } (g \circ (\$) f)$)

proof(*rule finfun-rec-unique*)

{ **fix** c **show** $\text{Abs-finfun } (g \circ (\$) (K\$ c)) = (K\$ g c)$

by(*simp add: finfun-comp-def o-def*)(*simp add: finfun-const-def*) }

{ **fix** $g' a b$ **show** $\text{Abs-finfun } (g \circ (\$) g'(a \$:= b)) = (\text{Abs-finfun } (g \circ (\$) g'))(a \$:= g b)$

proof –

```

    obtain  $y$  where  $y: y \in \text{finfun}$  and  $g': g' = \text{Abs-finfun } y$  by(cases  $g'$ )
  moreover from  $g'$  have  $(g \circ (\$) g') \in \text{finfun}$  by(simp add: finfun-left-compose)
  moreover have  $g \circ y(a := b) = (g \circ y)(a := g b)$  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* :: $'b \Rightarrow f 'c \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow f 'c$ (**infixr** $\$ \circ$ 55)
where [code del]: $g \$ \circ f = \text{Abs-finfun } ((\$) g \circ f)$

notation (*ASCII*)
finfun-comp2 (**infixr** $\$ \circ$ 55)

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

lemma *finfun-comp2-update*:
assumes *inj*: $\text{inj } f$
shows $\text{finfun-comp2 } (g(b \$:= c)) f = (\text{if } b \in \text{range } f \text{ then } (\text{finfun-comp2 } g f)(\text{inv } f b \$:= c) \text{ else } \text{finfun-comp2 } g f)$
including *finfun*
proof(cases $b \in \text{range } f$)
case *True*
from *inj* have $\bigwedge x. ((\$) g)(f x := c) \circ f = ((\$) g \circ f)(x := c)$ **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 $((\$) g)(b := c) \circ f = (\$) g \circ f$ **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* :: $'a \text{ list} \Rightarrow 'a \Rightarrow f \text{ bool} \Rightarrow \text{bool}$
where [code del]: $\text{finfun-All-except } A P \equiv \forall a. a \in \text{set } A \vee P \$ a$

lemma *finfun-All-except-const*: $\text{finfun-All-except } A (K \$ b) \longleftrightarrow b \vee \text{set } A = \text{UNIV}$
by(auto simp add: finfun-All-except-def)

lemma *finfun-All-except-const-finfun-UNIV-code* [code]:
 $\text{finfun-All-except } A (K \$ b) = (b \vee \text{is-list-UNIV } A)$
by(simp add: finfun-All-except-const is-list-UNIV-iff)

lemma *finfun-All-except-update*:

finfun-All-except $A f(a \text{ \$} := b) = ((a \in \text{set } A \vee b) \wedge \text{finfun-All-except } (a \# A) f)$
by(*fastforce simp add: finfun-All-except-def finfun-upd-apply*)

lemma *finfun-All-except-update-code* [*code*]:

fixes $a :: 'a :: \text{card-UNIV}$

shows *finfun-All-except* $A (\text{finfun-update-code } f a b) = ((a \in \text{set } A \vee b) \wedge \text{finfun-All-except } (a \# A) f)$

by(*simp add: finfun-All-except-update*)

definition *finfun-All* :: $'a \Rightarrow f \text{ bool} \Rightarrow \text{bool}$

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* $f(a \text{ \$} := b) = (b \wedge \text{finfun-All-except } [a] f)$

by(*simp add: finfun-All-def finfun-All-except-update*)

lemma *finfun-All-All*: *finfun-All* $P = \text{All } ((\$) P)$

by(*simp add: finfun-All-def finfun-All-except-def*)

definition *finfun-Ex* :: $'a \Rightarrow f \text{ bool} \Rightarrow \text{bool}$

where *finfun-Ex* $P = \text{Not } (\text{finfun-All } (\text{Not } \circ \$ P))$

lemma *finfun-Ex-Ex*: *finfun-Ex* $P = \text{Ex } ((\$) P)$

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* :: $'a \Rightarrow f 'b \Rightarrow 'a \Rightarrow f 'c \Rightarrow 'a \Rightarrow f ('b \times 'c) ((1'(\$-/-\$'))$
 $[0, 0] 1000)$

where [*code del*]: $(\$f, g\$) = \text{finfun-rec } (\lambda b. \text{Pair } b \circ \$ g) (\lambda a b c. c(a \text{ \$} := (b, g \$ a))) f$

interpretation *finfun-Diag-aux*: *finfun-rec-wf-aux* $\lambda b. \text{Pair } b \circ \$ g \lambda a b c. c(a \text{ \$} := (b, g \$ a))$

by(*unfold-locales*)(*simp-all add: expand-finfun-eq fun-eq-iff finfun-upd-apply*)

interpretation *finfun-Diag*: *finfun-rec-wf* $\lambda b. \text{Pair } b \circ \$ g \lambda a b c. c(a \text{ \$} := (b, g \$ a))$

proof

fix $b' b :: 'a$

assume *fin*: *finite* (*UNIV* :: $'c \text{ set}$)

{ **fix** $A :: 'c \text{ set}$

interpret *comp-fun-commute* $\lambda a c. c(a \text{ \$} := (b', g \$ a))$ **by**(*rule finfun-Diag-aux.upd-left-comm*)

from *fin* **have** *finite A* **by**(*auto intro: finite-subset*)
hence *Finite-Set.fold* ($\lambda a c. c(a \text{ \$} := (b', g \text{ \$ } a))$) (*Pair b* \circ *g*) *A* =
Abs-finfun ($\lambda a. (if a \in A \text{ then } b' \text{ else } b, g \text{ \$ } a)$)
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* ($\lambda a c. c(a \text{ \$} := (b', g \text{ \$ } a))$) (*Pair b* \circ *g*) *UNIV* = *Pair b'* \circ *g*
by(*simp add: finfun-const-def finfun-comp-conv-comp o-def*)
qed

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

Do not use ($\text{\$K\$ } ?b, ?g\text{\$}$) = *Pair ?b* \circ *?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 (\circ *g*).

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

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

lemma *finfun-Diag-const2*: ($\text{\$f}, K\text{\$ } c\text{\$}$) = ($\lambda b. (b, c)$) \circ *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*: ($\text{\$f}, g(a \text{ \$} := c)\text{\$}$) = ($\text{\$f}, g\text{\$}$)(*a* $\text{\$} := (f \text{ \$ } a, c)$)
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*]: ($\text{\$K\$ } b, K\text{\$ } c\text{\$}$) = (*K* $\text{\$}$ (*b*, *c*))
by(*simp add: finfun-Diag-const1*)

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

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

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

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

lemma *finfun-Diag-apply* [*simp*]: $(\$) (\$f, g\$) = (\lambda x. (f \$ x, g \$ x))$
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*:

$(\$f, g\$) = \text{Abs-finfun } ((\lambda x. (f \$ x, g \$ x)))$

including *finfun*

proof –

have $(\lambda f :: 'a \Rightarrow f 'b. (\$f, g\$)) = (\lambda f. \text{Abs-finfun } ((\lambda x. (f \$ x, g \$ x))))$

proof(*rule finfun-rec-unique*)

{ **fix** *c* **show** $\text{Abs-finfun } (\lambda x. ((K\$ c) \$ x, g \$ x)) = \text{Pair } c \circ \$ g$
by(*simp add: finfun-comp-conv-comp o-def finfun-const-def*) }

{ **fix** *g' a b*

show $\text{Abs-finfun } (\lambda x. (g'(a \$:= b) \$ x, g \$ x)) =$

$(\text{Abs-finfun } (\lambda x. (g' \$ x, g \$ x)))(a \$:= (b, g \$ a))$

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*: $(\$f, g\$) = (\$f', g' \$) \longleftrightarrow f = f' \wedge g = g'$

by(*auto simp add: expand-finfun-eq fun-eq-iff*)

definition *finfun-fst* :: $'a \Rightarrow f ('b \times 'c) \Rightarrow 'a \Rightarrow f 'b$

where [*code*]: $\text{finfun-fst } f = \text{fst} \circ \$ f$

lemma *finfun-fst-const*: $\text{finfun-fst } (K\$ bc) = (K\$ \text{fst } bc)$

by(*simp add: finfun-fst-def*)

lemma *finfun-fst-update*: $\text{finfun-fst } (f(a \$:= bc)) = (\text{finfun-fst } f)(a \$:= \text{fst } bc)$

and *finfun-fst-update-code*: $\text{finfun-fst } (\text{finfun-update-code } f a bc) = (\text{finfun-fst } f)(a \$:= \text{fst } bc)$

by(*simp-all add: finfun-fst-def*)

lemma *finfun-fst-comp-conv*: $\text{finfun-fst } (f \circ \$ g) = (\text{fst} \circ f) \circ \$ g$

by(*simp add: finfun-fst-def*)

lemma *finfun-fst-conv* [*simp*]: $\text{finfun-fst } (\$f, g\$) = f$

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*: $\text{finfun-fst} = (\lambda f. \text{Abs-finfun } (\text{fst} \circ (\$) f))$

by(*simp add: finfun-fst-def [abs-def] finfun-comp-conv-comp*)

definition *finfun-snd* :: $'a \Rightarrow f ('b \times 'c) \Rightarrow 'a \Rightarrow f 'c$

where [code]: $\text{finfun-snd } f = \text{snd} \circ \$ f$

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

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

lemma *finfun-snd-comp-conv*: $\text{finfun-snd } (f \circ \$ g) = (\text{snd} \circ f) \circ \$ g$
by(*simp add: finfun-snd-def*)

lemma *finfun-snd-conv* [simp]: $\text{finfun-snd } (\$f, g\$) = g$
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*: $\text{finfun-snd} = (\lambda f. \text{Abs-finfun } (\text{snd} \circ (\$) f))$
by(*simp add: finfun-snd-def [abs-def] finfun-comp-conv-comp*)

lemma *finfun-Diag-collapse* [simp]: $(\$ \text{finfun-fst } f, \text{finfun-snd } f\$) = f$
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* :: $('a \times 'b) \Rightarrow f 'c \Rightarrow 'a \Rightarrow f 'b \Rightarrow f 'c$
where [code del]: $\text{finfun-curry} = \text{finfun-rec } (\text{finfun-const} \circ \text{finfun-const}) (\lambda(a, b) c f. f(a \$:= (f \$ a)(b \$:= c)))$

interpretation *finfun-curry-aux*: $\text{finfun-rec-wf-aux } \text{finfun-const} \circ \text{finfun-const } \lambda(a, b) c f. f(a \$:= (f \$ a)(b \$:= c))$
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*: $\text{finfun-rec-wf } \text{finfun-const} \circ \text{finfun-const } \lambda(a, b) c f. f(a \$:= (f \$ a)(b \$:= c))$

proof(*unfold-locales*)

fix $b' b :: 'b$

assume $\text{fin}: \text{finite } (\text{UNIV} :: ('c \times 'a) \text{ set})$

hence $\text{fin1}: \text{finite } (\text{UNIV} :: 'c \text{ set})$ **and** $\text{fin2}: \text{finite } (\text{UNIV} :: 'a \text{ set})$

unfolding *UNIV-Times-UNIV*[*symmetric*]

by(*fastforce dest: finite-cartesian-productD1 finite-cartesian-productD2*)**+**

note [simp] = $\text{Abs-finfun-inverse-finite}[OF \text{fin}] \text{Abs-finfun-inverse-finite}[OF \text{fin1}] \text{Abs-finfun-inverse-finite}[OF \text{fin2}]$

```

{ fix A :: ('c × 'a) set
  interpret comp-fun-commute λa :: '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 (λa :: 'c × 'a. (λ(a, b) c f. f(a $:= (f $ a)(b $:= c)))
a b') ((finfun-const ∘ finfun-const) b) A = Abs-finfun (λa. Abs-finfun (λb''. 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 (λa :: '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 (λa. 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 ≠ (λx. 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 (λa. Abs-finfun (curry (finfun-apply f) a))
  including finfun
proof –
  have finfun-curry = (λf :: ('a × 'b) ⇒f 'c. Abs-finfun (λa. 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 $:= c))
    by(cases a) simp
  show Abs-finfun (λa. Abs-finfun (curry (finfun-apply (K$ c)) a)) = (K$ K$ c)
    by(simp add: finfun-curry-def finfun-const-def curry-def)
  fix g b
  show Abs-finfun (λaa. Abs-finfun (curry (($) g(a $:= b)) aa)) =
    (Abs-finfun (λa. Abs-finfun (curry (($) g) a)))(
    fst a $:= ((Abs-finfun (λa. Abs-finfun (curry (($) g) a))) $ (fst a))(snd a $:=
b))
    by(cases a)(auto intro!: ext arg-cong[where f=Abs-finfun] simp add: fin-
fun-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*: $(f = g) \longleftrightarrow \text{finfun-All } ((\lambda(x, y). x = y) \circ \$ (\$f, g\$))$
by(simp add: expand-finfun-eq fun-eq-iff finfun-All-All o-def)

instantiation *finfun* :: $(\{\text{card-UNIV}, \text{equal}\}, \text{equal})$ *equal* **begin**

definition *eq-finfun-def* [code]: $\text{HOL.equal } f \ g \longleftrightarrow \text{finfun-All } ((\lambda(x, y). x = y) \circ \$ (\$f, g\$))$

instance **by**(intro-classes)(simp add: eq-finfun-All-ext eq-finfun-def)
end

lemma [code nbe]:
 $\text{HOL.equal } (f :: - \Rightarrow f -) \ f \longleftrightarrow \text{True}$
by (fact equal-refl)

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

definition *finfun-clearjunk* :: $'a \Rightarrow f 'b \Rightarrow 'a \Rightarrow f 'b$
where [simp, code del]: *finfun-clearjunk* = *id*

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

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

1.17 The domain of a FinFun as a FinFun

definition *finfun-dom* :: $('a \Rightarrow f 'b) \Rightarrow ('a \Rightarrow f \text{bool})$
where [code del]: *finfun-dom* $f = \text{Abs-finfun } (\lambda a. f \$ a \neq \text{finfun-default } f)$

lemma *finfun-dom-const*:
 $finfun-dom ((K\$ c) :: 'a \Rightarrow f 'b) = (K\$ finite (UNIV :: 'a set) \wedge c \neq undefined)$
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*]:
 $finfun-dom ((K\$ c) :: ('a :: card-UNIV) \Rightarrow f 'b) =$
(if CARD('a) = 0 then (K\$ False) else Code.abort (STR "finfun-dom called on finite type")) ($\lambda\cdot. finfun-dom (K\$ c)$)
by(*simp add: finfun-dom-const card-UNIV card-eq-0-iff*)

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

lemma *finfun-dom-update* [*simp*]:
 $finfun-dom (f(a \$:= b)) = (finfun-dom f)(a \$:= (b \neq finfun-default f))$
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*]:
 $finfun-dom (finfun-update-code f a b) = finfun-update-code (finfun-dom f) a (b \neq finfun-default f)$
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) \wedge b \neq 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* :: $('a :: linorder) \Rightarrow f 'b \Rightarrow 'a list$

where

finfun-to-list $f = (THE\ xs.\ set\ xs = \{x.\ finfun\text{-}dom\ f\ \$\ x\} \wedge\ sorted\ xs \wedge\ distinct\ xs)$

lemma *set-finfun-to-list* [*simp*]: $set\ (finfun\text{-}to\text{-}list\ f) = \{x.\ finfun\text{-}dom\ f\ \$\ x\}$ (**is** *?thesis1*)

and *sorted-finfun-to-list*: $sorted\ (finfun\text{-}to\text{-}list\ f)$ (**is** *?thesis2*)

and *distinct-finfun-to-list*: $distinct\ (finfun\text{-}to\text{-}list\ f)$ (**is** *?thesis3*)

proof (*atomize* (*full*))

show *?thesis1* \wedge *?thesis2* \wedge *?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 $((K\ \$\ c) :: ('a :: \{linorder\} \Rightarrow f\ 'b)) =$

$(if\ \neg\ finite\ (UNIV :: 'a\ set) \vee\ c = undefined\ then\ []\ else\ THE\ xs.\ set\ xs = UNIV \wedge\ sorted\ xs \wedge\ distinct\ xs)$

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 $((K\ \$\ c) :: ('a :: \{linorder,\ card\ UNIV\} \Rightarrow f\ 'b)) =$

$(if\ CARD('a) = 0\ then\ []\ else\ Code.abort\ (STR\ "finfun-to-list\ called\ on\ finite\ type"))\ (\lambda.\ finfun\text{-}to\text{-}list\ ((K\ \$\ c) :: ('a \Rightarrow f\ 'b)))$

by(*auto* *simp* *add*: *finfun-to-list-const* *card-UNIV* *card-eq-0-iff*)

lemma *remove1-insort-insert-same*:

$x \notin set\ xs \implies remove1\ x\ (insort\text{-}insert\ x\ xs) = xs$

by (*metis* *insort-insert-insort* *remove1-insort-key*)

lemma *finfun-dom-conv*:

finfun-dom $f\ \$\ x \longleftrightarrow f\ \$\ x \neq finfun\text{-}default\ f$

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 $(f(a\ \$:=\ b)) =$

$(if\ b = finfun\text{-}default\ f\ then\ List.remove1\ a\ (finfun\text{-}to\text{-}list\ f)\ else\ List.insort\text{-}insert\ a\ (finfun\text{-}to\text{-}list\ f))$

proof(*subst* *finfun-to-list-def*, *rule* *the-equality*)

fix *xs*

assume $set\ xs = \{x.\ finfun\text{-}dom\ f(a\ \$:=\ b)\ \$\ x\} \wedge\ sorted\ xs \wedge\ distinct\ xs$

```

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 in-
sort-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 = insert-insert a xs
      unfolding finfun-to-list-def
    proof(rule the-equality)
    have set (insert-insert a xs) = insert a (set xs) by(simp add: set-insert-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 (insert-insert a xs) = {x. finfun-dom f $ x}  $\wedge$  sorted
(insert-insert a xs)  $\wedge$  distinct (insert-insert a xs)
      by(simp add: sorted-insert-insert distinct-insert-insert)

    fix xs'
    assume set xs' = {x. finfun-dom f $ x}  $\wedge$  sorted xs'  $\wedge$  distinct xs'
    thus xs' = insert-insert a xs using 1 by(auto dest: sorted-distinct-set-unique)
    qed
  with eq True show ?thesis by(simp add: remove1-insert-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 fin-
fun-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}  $\wedge$  sorted xs  $\wedge$  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: insert-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: insert-insert-insort insort-remove1)
qed
qed
qed (auto simp add: distinct-funfun-to-list sorted-funfun-to-list sorted-remove1 set-insert-insert
sorted-insert-insort distinct-insort-insort 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.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  $card \dots = 1$  by simp
  finally show ?lhs .
qed

lemma card-UNIV-funfun:
  defines  $F == finfun :: ('a \Rightarrow 'b)\ set$ 
  shows  $CARD('a \Rightarrow_f 'b) = (if\ CARD('a) \neq 0 \wedge\ CARD('b) \neq 0 \vee\ CARD('b) = 1\ then\ CARD('b) \wedge\ CARD('a)\ else\ 0)$ 
  proof(cases  $0 < CARD('a) \wedge 0 < CARD('b) \vee CARD('b) = 1$ )
    case True
      from True have  $F = UNIV$ 
      proof
        assume  $b: CARD('b) = 1$ 
        hence  $\forall x :: 'b.\ x = undefined$ 
          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  $CARD('a \Rightarrow_f 'b) = card\ F$ 
        unfolding type-definition.Abs-image[OF type-definition-funfun, symmetric]
        by(auto intro!: card-image inj-onI simp add: Abs-funfun-inject F-def)
      ultimately show ?thesis by(simp add: card-fun)
    next
      case False
        hence infinite:  $\neg (finite\ (UNIV :: 'a\ set) \wedge finite\ (UNIV :: 'b\ set))$ 
          and  $b: CARD('b) \neq 1$  by(simp-all add: card-eq-0-iff)
        from b obtain  $b1\ b2 :: 'b$  where  $b1 \neq b2$ 
          by(auto simp add: card-eq-1-iff simp del: One-nat-def)
        let  $?f = \lambda a\ a' :: 'a.\ if\ a = a'\ then\ b1\ else\ b2$ 
        from infinite have  $\neg finite\ (UNIV :: ('a \Rightarrow_f 'b)\ set)$ 
          proof(rule contrapos-nn[OF - conjI])
            assume finite:  $finite\ (UNIV :: ('a \Rightarrow_f 'b)\ set)$ 
            hence finite F
              unfolding type-definition.Abs-image[OF type-definition-funfun, symmetric]
              F-def
              by(rule finite-imageD)(auto intro: inj-onI simp add: Abs-funfun-inject)
            hence finite (range ?f)
              by(rule finite-subset[rotated 1])(auto simp add: F-def finfun-def  $\langle b1 \neq b2 \rangle$ 
              intro!: exI[where  $x=b2$ ])
            thus finite (UNIV :: 'a set)
              by(rule finite-imageD)(auto intro: inj-onI simp add: fun-eq-iff  $\langle b1 \neq b2 \rangle$  split:
              if-split-asm)
          qed
        from finite have finite (range  $(\lambda b.\ ((K\$ b) :: 'a \Rightarrow_f 'b))$ )
          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-funfun*:

finite (UNIV :: ('a \Rightarrow f 'b) set) \longleftrightarrow
(*finite* (UNIV :: 'a set) \wedge *finite* (UNIV :: 'b set) \vee *CARD*('b) = 1)
(is ?lhs \longleftrightarrow ?rhs)

proof –

have ?lhs \longleftrightarrow *CARD*('a \Rightarrow f 'b) > 0 **by**(simp add: *card-gt-0-iff*)
also have ... \longleftrightarrow *CARD*('a) > 0 \wedge *CARD*('b) > 0 \vee *CARD*('b) = 1
by(simp add: *card-UNIV-funfun*)
also have ... = ?rhs **by**(simp add: *card-gt-0-iff*)
finally show ?thesis .

qed

instantiation *funfun* :: (*finite-UNIV*, *card-UNIV*) *finite-UNIV* **begin**

definition *finite-UNIV* = *Phantom*('a \Rightarrow f 'b)
(let *cb* = *of-phantom* (*card-UNIV* :: 'b *card-UNIV*)
in *cb* = 1 \vee *of-phantom* (*finite-UNIV* :: 'a *finite-UNIV*) \wedge *cb* \neq 0)

instance

by *intro-classes* (auto simp add: *finite-UNIV-funfun-def* *Let-def* *card-UNIV* *finite-UNIV* *finite-UNIV-funfun* *card-gt-0-iff*)

end

instantiation *funfun* :: (*card-UNIV*, *card-UNIV*) *card-UNIV* **begin**

definition *card-UNIV* = *Phantom*('a \Rightarrow f 'b)
(let *ca* = *of-phantom* (*card-UNIV* :: 'a *card-UNIV*);
cb = *of-phantom* (*card-UNIV* :: 'b *card-UNIV*)
in if *ca* \neq 0 \wedge *cb* \neq 0 \vee *cb* = 1 then *cb* \wedge *ca* else 0)

instance **by** *intro-classes* (simp add: *card-UNIV-funfun-def* *card-UNIV* *Let-def* *card-UNIV-funfun*)

end

1.18.1 Bundles for concrete syntax

bundle *funfun-syntax*

begin

type-notation *funfun* ((- \Rightarrow f /-) [22, 21] 21)

notation

funfun-const (*K*\$/ - [0] 1) **and**
funfun-update (-'(- \$:= -') [1000, 0, 0] 1000) **and**
funfun-apply (**infixl** \$ 999) **and**
funfun-comp (**infixr** \circ \$ 55) **and**
funfun-comp2 (**infixr** \$ \circ 55) **and**
funfun-Diag ((1'(\$-/ -\$')) [0, 0] 1000)

```

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

end

bundle no-finfun-syntax
begin

no-type-notation
  finfun ((-  $\Rightarrow$  f /-) [22, 21] 21)

no-notation
  finfun-const (K$/ - [0] 1) and
  finfun-update (-'(- $:= -') [1000, 0, 0] 1000) and
  finfun-apply (infixl $ 999) and
  finfun-comp (infixr o$ 55) and
  finfun-comp2 (infixr $o 55) and
  finfun-Diag ((1'($-/ -$')) [0, 0] 1000)

no-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 predf = '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-funfun-code* [code]:

$f \leq g \iff \text{finfun-All } ((\lambda(x, y). x \leq y) \circ \$ (\$f, g\$))$
by(*simp add: le-funfun-def finfun-All-All o-def*)

end

instance *finfun* :: (type, preorder) preorder

by(*intro-classes*)(*auto simp add: less-funfun-def le-funfun-def intro: order-trans*)

instance *finfun* :: (type, order) order

by(*intro-classes*)(*auto simp add: le-funfun-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-funfun-def le-funfun-def*)

end

lemma *bot-funfun-apply* [*simp*]: ($\$$) *bot* = (λ -. *bot*)

by(*auto simp add: bot-funfun-def*)

instantiation *finfun* :: (type, order-top) order-top **begin**

definition *top* = *finfun-const top*

instance **by**(*intro-classes*)(*simp add: top-funfun-def le-funfun-def*)

end

lemma *top-funfun-apply* [*simp*]: ($\$$) *top* = (λ -. *top*)

by(*auto simp add: top-funfun-def*)

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

definition [code]: *inf f g* = ($\lambda(x, y). \text{inf } x \ y$) $\circ \$ (\$f, g\$)$

instance ..

end

lemma *inf-funfun-apply* [*simp*]: ($\$$) (*inf f g*) = *inf* (($\$$) *f*) (($\$$) *g*)

by(*auto simp add: inf-funfun-def o-def inf-fun-def*)

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

definition [code]: *sup f g* = ($\lambda(x, y). \text{sup } x \ y$) $\circ \$ (\$f, g\$)$

instance ..

end

lemma *sup-funfun-apply* [*simp*]: ($\$$) (*sup f g*) = *sup* (($\$$) *f*) (($\$$) *g*)

by(*auto simp add: sup-funfun-def o-def sup-fun-def*)

instance *finfun* :: (type, semilattice-inf) semilattice-inf

by(*intro-classes*)(*simp-all add: inf-funfun-def le-funfun-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 $f - g = \text{case-prod } (-) \circ \$ (f, g)$
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 $- A = \text{uminus } \circ \$ A$
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-fin-def inf-fin-def fun-Compl-def o-def inf-compl-bot sup-compl-top diff-eq*)

Replicate predicate operations for FinFuns

abbreviation *finfun-empty* :: 'a $\text{pred}_f (\{\}_f)$
where $\{\}_f \equiv \text{bot}$

abbreviation *finfun-UNIV* :: 'a pred_f
where *finfun-UNIV* $\equiv \text{top}$

definition *finfun-single* :: 'a \Rightarrow 'a pred_f
where [*code*]: *finfun-single* $x = \text{finfun-empty}(x \ \$:= \text{True})$

lemma *finfun-single-apply* [*simp*]:
 $\text{finfun-single } x \ \$ y \longleftrightarrow x = y$
by(*simp add: finfun-single-def finfun-upd-apply*)

lemma [*iff*]:
shows *finfun-single-neq-bot*: $\text{finfun-single } x \neq \text{bot}$
and *bot-neq-finfun-single*: $\text{bot} \neq \text{finfun-single } x$

by(*simp-all add: expand-funfun-eq fun-eq-iff*)

lemma *finfun-leI* [*intro!*]: $(!!x. A \$ x \implies B \$ x) \implies A \leq B$
by(*simp add: le-funfun-def*)

lemma *finfun-leD* [*elim*]: $\llbracket A \leq B; A \$ x \rrbracket \implies B \$ x$
by(*simp add: le-funfun-def*)

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

definition *finfun-Ball-except* :: $'a \text{ list} \Rightarrow 'a \text{ pred}_f \Rightarrow ('a \Rightarrow \text{bool}) \Rightarrow \text{bool}$
where [*code del*]: *finfun-Ball-except* *xs A P* = $(\forall a. A \$ a \longrightarrow a \in \text{set } xs \vee P a)$

lemma *finfun-Ball-except-const*:
finfun-Ball-except *xs (K\\$ b) P* $\longleftrightarrow \neg b \vee \text{set } xs = \text{UNIV} \vee \text{Code.abort } (\text{STR } "finfun-ball-except") (\lambda u. finfun-Ball-except } xs (K\$ b) P)$
by(*auto simp add: finfun-Ball-except-def*)

lemma *finfun-Ball-except-const-funfun-UNIV-code* [*code*]:
finfun-Ball-except *xs (K\\$ b) P* $\longleftrightarrow \neg b \vee \text{is-list-UNIV } xs \vee \text{Code.abort } (\text{STR } "finfun-ball-except") (\lambda u. finfun-Ball-except } xs (K\$ b) P)$
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* = $((a \in \text{set } xs \vee (b \longrightarrow P a)) \wedge finfun-Ball-except (a \# xs) A P)$
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* = $((a \in \text{set } xs \vee (b \longrightarrow P a)) \wedge finfun-Ball-except (a \# xs) f P)$
by(*simp add: finfun-Ball-except-update*)

definition *finfun-Ball* :: $'a \text{ pred}_f \Rightarrow ('a \Rightarrow \text{bool}) \Rightarrow \text{bool}$
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* :: $'a \text{ list} \Rightarrow 'a \text{ pred}_f \Rightarrow ('a \Rightarrow \text{bool}) \Rightarrow \text{bool}$
where [*code del*]: *finfun-Bex-except* *xs A P* = $(\exists a. A \$ a \wedge a \notin \text{set } xs \wedge P a)$

lemma *finfun-Bex-except-const*:
finfun-Bex-except *xs (K\\$ b) P* $\longleftrightarrow b \wedge \text{set } xs \neq \text{UNIV} \wedge \text{Code.abort } (\text{STR } "finfun-Bex-except") (\lambda u. finfun-Bex-except } xs (K\$ b) P)$
by(*auto simp add: finfun-Bex-except-def*)

lemma *finfun-Bex-except-const-finfun-UNIV-code* [code]:
 $\text{finfun-Bex-except } xs \ (K \$ b) \ P \longleftrightarrow b \wedge \neg \text{is-list-UNIV } xs \wedge \text{Code.abort } (STR$
"finfun-Bex-except") $(\lambda u. \text{finfun-Bex-except } xs \ (K \$ b) \ P)$
by(*auto simp add: finfun-Bex-except-def is-list-UNIV-iff*)

lemma *finfun-Bex-except-update*:
 $\text{finfun-Bex-except } xs \ (A(a \$ = b)) \ P \longleftrightarrow (a \notin \text{set } xs \wedge b \wedge P a) \vee \text{finfun-Bex-except}$
 $(a \# xs) \ A \ P$
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 $a :: 'a :: \text{card-UNIV}$
shows $\text{finfun-Bex-except } xs \ (\text{finfun-update-code } f \ a \ b) \ P \longleftrightarrow ((a \notin \text{set } xs \wedge b \wedge$
 $P \ a) \vee \text{finfun-Bex-except } (a \# xs) \ f \ P)$
by(*simp add: finfun-Bex-except-update*)

definition *finfun-Bex* $:: 'a \text{ pred}_f \Rightarrow ('a \Rightarrow \text{bool}) \Rightarrow \text{bool}$
where [code del]: $\text{finfun-Bex } A \ P = \text{Bex } \{x. A \ \$ \ x\} \ P$

lemma *finfun-Bex-code* [code]: $\text{finfun-Bex} = \text{finfun-Bex-except } []$
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]:
 $\text{sup } ((\$) \ A) \ ((\$) \ B) = (\$) \ (\text{sup } A \ B)$
by(*simp*)

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

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

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

$A \$ x \wedge B \$ x \longleftrightarrow \text{inf } A B \$ x$
by(*simp add: inf-fun-def*)

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

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

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

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

lemma *iso-funfun-minus* [*code-unfold*]:
fixes $A :: 'a \text{ pred}_f$
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-funfun-Ball-Ball*:
 $(\forall x. A \$ x \longrightarrow P x) \longleftrightarrow \text{finfun-Ball } A P$
by(*simp add: finfun-Ball-def*)

lemma *iso-funfun-Bex-Bex*:
 $(\exists x. A \$ x \wedge P x) \longleftrightarrow \text{finfun-Bex } A P$
by(*simp add: finfun-Bex-def*)

Test code setup

notepad begin
have $\text{inf } ((\lambda- :: \text{nat. False})(1 := \text{True}, 2 := \text{True})) ((\lambda-. \text{True})(3 := \text{False})) \leq$
 $\text{sup } ((\lambda-. \text{False})(1 := \text{True}, 5 := \text{True})) (- ((\lambda-. \text{True})(2 := \text{False}, 3 :=$
 $\text{False}))$
by *eval*
end

declare *iso-funfun-Ball-Ball*[*code-unfold*]
notepad begin
have $(\forall x. ((\lambda- :: \text{nat. False})(1 := \text{True}, 2 := \text{True})) x \longrightarrow x < 3)$

```
  by eval
end
declare iso-funfun-Ball-Ball[code-unfold del]

end
```