Finfuns

Andreas Lochbihler

December 12, 2009

Contents

1 Almost everywhere constant functions

theory *FinFun*
imports *Main Infinite-Set Enum*
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 $\mathit{map\text{-}default} :: 'b \Rightarrow ('a \rightharpoonup 'b) \Rightarrow 'a \Rightarrow 'b$
where $\mathit{map\text{-}default}\ b\ f\ a \equiv \mathit{case}\ f\ a\ \mathit{of}\ \mathit{None} \Rightarrow b \mid \mathit{Some}\ b' \Rightarrow b'$

lemma *map-default-delete* [*simp*]:
  $\mathit{map\text{-}default}\ b\ (f(a := \mathit{None})) = (\mathit{map\text{-}default}\ b\ f)(a := b)$
by(*simp add*: *map-default-def expand-fun-eq*)

lemma *map-default-insert*:
  $\mathit{map\text{-}default}\ b\ (f(a \mapsto b')) = (\mathit{map\text{-}default}\ b\ f)(a := b')$
by(*simp add*: *map-default-def expand-fun-eq*)

lemma *map-default-empty* [*simp*]: $\mathit{map\text{-}default}\ b\ \mathit{empty} = (\lambda a.\ b)$
by(*simp add*: *expand-fun-eq map-default-def*)

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

typedef $('a,'b)$ *finfun* $= \{f::'a \Rightarrow 'b.\ \exists\, b.\ \mathit{finite}\ \{a.\ f\ a \neq b\}\}$
proof $-$
have $\exists f.\ \mathit{finite}\ \{x.\ f\ x \neq \mathit{undefined}\}$
proof
show $\mathit{finite}\ \{x.\ (\lambda y.\ \mathit{undefined})\ x \neq \mathit{undefined}\}$ **by** *auto*
qed
then show *?thesis* **by** *auto*
qed

syntax
finfun :: *type* $\Rightarrow$ *type* $\Rightarrow$ *type* $((\text{-} \Rightarrow_f /\text{-})\ [22,\ 21]\ 21)$

lemma *fun-upd-finfun*: $y(a := b) \in \mathit{finfun} \longleftrightarrow y \in \mathit{finfun}$
proof $-$
{ fix b'
have $\mathit{finite}\ \{a'.\ (y(a := b))\ a' \neq b'\} = \mathit{finite}\ \{a'.\ y\ a' \neq b'\}$
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' ≠ 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: (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 x≡{a. y a ≠ b} arbitrary: y)
  case empty
  hence y = (λa. b) by(auto intro: ext)
  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 expand-fun-eq*)
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

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

lemma *Abs-finfun-inject-finite*:

fixes $x\ y :: 'a \Rightarrow 'b$
assumes *fin*: *finite* $(UNIV :: 'a\ set)$
shows *Abs-finfun* $x =$ *Abs-finfun* $y \longleftrightarrow x = y$
proof
assume *Abs-finfun* $x =$ *Abs-finfun* 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-finfun-inject*)
qed *simp*

lemma *Abs-finfun-inject-finite-class*:
fixes $x\ y :: ('a :: finite) \Rightarrow 'b$
shows *Abs-finfun* $x =$ *Abs-finfun* $y \longleftrightarrow x = y$
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* :: $('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow_f 'b$)
proof(*rule inj-onI*)
fix $x\ y :: 'a \Rightarrow 'b$
assume *Abs-finfun* $x =$ *Abs-finfun* 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-finfun-inject*)
qed

declare *finfun-simp* [*simp*] *finfun-iff* [*iff*] *finfun-intro* [*intro*]

lemma *Abs-finfun-inverse-finite*:
fixes $x :: 'a \Rightarrow 'b$
assumes *fin*: *finite* $(UNIV :: 'a\ set)$
shows *Rep-finfun* (*Abs-finfun* x) $= x$
proof −
from *fin* **have** $x \in$ *finfun*
by(*auto simp add*: *finfun-def intro*: *finite-subset*)
thus *?thesis* **by** *simp*
qed

declare *finfun-simp* [*simp del*] *finfun-iff* [*iff del*] *finfun-intro* [*rule del*]

lemma *Abs-finfun-inverse-finite-class*:
fixes $x :: ('a :: finite) \Rightarrow 'b$
shows *Rep-finfun* (*Abs-finfun* x) $= x$
using *finite-UNIV* **by**(*simp add*: *Abs-finfun-inverse-finite*)

lemma *finfun-eq-finite-UNIV*: *finite* $(UNIV :: 'a\ set) \Longrightarrow$ (*finfun* :: $('a \Rightarrow 'b)\ set$) $= UNIV$
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*: *expand-fun-eq 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 $'a \Rightarrow_f 'b$

definition *finfun-const* :: $'b \Rightarrow 'a \Rightarrow_f 'b$ (λ^f/ - [*0*] *1*)
where [*code del*]: (λ^f *b*) = *Abs-finfun* (λ*x*. *b*)

definition *finfun-update* :: $'a \Rightarrow_f 'b \Rightarrow 'a \Rightarrow 'b \Rightarrow 'a \Rightarrow_f 'b$ (-'(^{f}/ - := -')
[*1000*,*0*,*0*] *1000*)
where [*code del*]: *f*(^{f} *a* := *b*) = *Abs-finfun* ((*Rep-finfun f*)(*a* := *b*))

declare *finfun-simp* [*simp*] *finfun-iff* [*iff*] *finfun-intro* [*intro*]

lemma *finfun-update-twist*: *a* ≠ *a'* ⟹ *f*(^{f} *a* := *b*)(^{f} *a'* := *b'*) = *f*(^{f} *a'* := *b'*)(^{f}
a := *b*)
by(*simp add*: *finfun-update-def fun-upd-twist*)

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

lemma *finfun-update-const-same*: (λ^f *b*)(^{f} *a* := *b*) = (λ^f *b*)
by(*simp add*: *finfun-update-def finfun-const-def expand-fun-eq*)

declare *finfun-simp* [*simp del*] *finfun-iff* [*iff del*] *finfun-intro* [*rule del*]

1.4 Code generator setup

definition *finfun-update-code* :: $'a \Rightarrow_f 'b \Rightarrow 'a \Rightarrow 'b \Rightarrow 'a \Rightarrow_f 'b$ (-$'$($^{f\,c}$/ - := -$'$) [*1000*,*0*,*0*] *1000*)
where [*simp*, *code del*]: *finfun-update-code* = *finfun-update*

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

lemma *finfun-update-const-code* [*code*]:
$(\lambda^f\ b)(^f\ a := b') = (\mathit{if}\ b = b'\ \mathit{then}\ (\lambda^f\ b)\ \mathit{else}\ \mathit{finfun\text{-}update\text{-}code}\ (\lambda^f\ b)\ a\ b')$
by(*simp add*: *finfun-update-const-same*)

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

1.5 Setup for quickcheck

notation *fcomp* (**infixl** *o>* *60*)
notation *scomp* (**infixl** *o→* *60*)

definition (**in** *term-syntax*) *valtermify-finfun-const* ::
$'b$::*typerep* × (*unit* ⇒ *Code-Evaluation.term*) ⇒ ($'a$::*typerep* $\Rightarrow_f$ $'b$) × (*unit* ⇒ *Code-Evaluation.term*) **where**
valtermify-finfun-const y = *Code-Evaluation.valtermify finfun-const* {·} *y*

definition (**in** *term-syntax*) *valtermify-finfun-update-code* ::
$'a$::*typerep* × (*unit* ⇒ *Code-Evaluation.term*) ⇒ $'b$::*typerep* × (*unit* ⇒ *Code-Evaluation.term*) ⇒ ($'a \Rightarrow_f 'b$) × (*unit* ⇒ *Code-Evaluation.term*) ⇒ ($'a \Rightarrow_f 'b$) × (*unit* ⇒ *Code-Evaluation.term*)
where
valtermify-finfun-update-code x y f = *Code-Evaluation.valtermify finfun-update-code* {·} *f* {·} *x* {·} *y*

instantiation *finfun* :: (*random*, *random*) *random*
begin

primrec *random-finfun-aux* :: *code-numeral* ⇒ *code-numeral* ⇒ *Random.seed* ⇒ ($'a \Rightarrow_f 'b$ × (*unit* ⇒ *Code-Evaluation.term*)) × *Random.seed* **where**
random-finfun-aux 0 j = *Quickcheck.collapse* (*Random.select-weight*
[(*1*, *Quickcheck.random j o→* (λ*y*. *Pair* (*valtermify-finfun-const y*)))])
| *random-finfun-aux* (*Suc-code-numeral i*) *j* = *Quickcheck.collapse* (*Random.select-weight*
[(*Suc-code-numeral i*, *Quickcheck.random j o→* (λ*x*. *Quickcheck.random j o→*
(λ*y*. *random-finfun-aux i j o→* (λ*f*. *Pair* (*valtermify-finfun-update-code x y f*))))),
(*1*, *Quickcheck.random j o→* (λ*y*. *Pair* (*valtermify-finfun-const y*)))])

definition
Quickcheck.random i = *random-finfun-aux i i*

instance ..

end

lemma *random-finfun-aux-code* [*code*]:
random-finfun-aux i j = *Quickcheck.collapse* (*Random.select-weight*
[(*i*, *Quickcheck.random j o→* (λx. *Quickcheck.random j o→* (λy. *random-finfun-aux*
(*i* − *1*) *j o→* (λf. *Pair* (*valtermify-finfun-update-code x y f*))))),
(*1*, *Quickcheck.random j o→* (λy. *Pair* (*valtermify-finfun-const y*)))])
apply (*cases i rule*: *code-numeral.exhaust*)
apply (*simp-all only*: *random-finfun-aux.simps code-numeral-zero-minus-one Suc-code-numeral-minus-one*)
apply (*subst select-weight-cons-zero*) **apply** (*simp only*:)
done

no-notation *fcomp* (**infixl** *o> 60*)
no-notation *scomp* (**infixl** *o→ 60*)

1.6 *finfun-update* as instance of *fun-left-comm*

declare *finfun-simp* [*simp*] *finfun-iff* [*iff*] *finfun-intro* [*intro*]

interpretation *finfun-update*: *fun-left-comm* $\lambda a\, f.\ f({}^f\ a :: {}'a := b')$
proof
fix $a'\ a :: {}'a$
fix b
have (*Rep-finfun b*)$(a := b',\ a' := b')$ = (*Rep-finfun b*)$(a' := b',\ a := b')$
by(*cases* $a = a'$)(*auto simp add*: *fun-upd-twist*)
thus $b({}^f\ a := b')({}^f\ a' := b') = b({}^f\ a' := b')({}^f\ a := b')$
by(*auto simp add*: *finfun-update-def fun-upd-twist*)
qed

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

lemma *finfun-default-aux-infinite*:
  fixes f :: $'a \Rightarrow 'b$
  assumes *infin*: *infinite* (*UNIV* :: *'a set*)
  and *fin*: *finite* $\{a.\ f\ a \neq b\}$
  shows *finfun-default-aux* $f = b$
proof −
  let $?B = \{a.\ f\ a \neq b\}$
  from *fin* **have** (*THE* b. *finite* $\{a.\ f\ a \neq b\}$) $= b$
  proof(*rule the-equality*)
    fix b'
    assume *finite* $\{a.\ f\ a \neq b'\}$ (**is** *finite* $?B'$)
    with *infin fin* **have** *UNIV* − $(?B' \cup ?B) \neq \{\}$ **by**(*auto dest*: *finite-subset*)
    then obtain a **where** a: $a \notin ?B' \cup ?B$ **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*: $f \in$ *finfun*
  shows *finite* $\{a.\ f\ a \neq$ *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* $\{a.\ f\ a \neq b\}$ (**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

definition *finfun-default* :: $'a \Rightarrow_f\ 'b \Rightarrow\ 'b$
where [*code del*]: *finfun-default f* = *finfun-default-aux* (*Rep-finfun f*)

lemma *finite-finfun-default*: *finite* {*a*. *Rep-finfun f a* ≠ *finfun-default f*}
unfolding *finfun-default-def* **by**(*simp add*: *finite-finfun-default-aux*)

lemma *finfun-default-const*: *finfun-default* $((\lambda^f\ b) :: 'a \Rightarrow_f\ 'b)$ = (*if finite* (*UNIV* :: $'a$ *set*) *then undefined else b*)
apply(*auto simp add*: *finfun-default-def finfun-const-def finfun-default-aux-infinite*)
apply(*simp add*: *finfun-default-aux-def*)
done

lemma *finfun-default-update-const*:
finfun-default $(f(^f\ a := b))$ = *finfun-default f*
unfolding *finfun-default-def finfun-update-def*
by(*simp add*: *finfun-default-aux-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*) ∧ $b \notin$ *ran g*
in 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*: *fun-left-comm* ($\lambda a.\ upd\ a\ (f\ a)$)
by(*unfold-locales*)(*auto intro*: *upd-commute*)

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*)

declare *finfun-simp* [*simp*] *finfun-iff* [*iff*] *finfun-intro* [*intro*]

lemma *map-default-update-const*:
assumes *fin*: *finite* (*dom f*)
and *anf*: $a \notin dom\ f$
and *fg*: $f \subseteq_m g$
shows $upd\ a\ d\ (fold\ (\lambda a.\ upd\ a\ (map\text{-}default\ d\ g\ a))\ (cnst\ d)\ (dom\ f)) =$
$fold\ (\lambda a.\ upd\ a\ (map\text{-}default\ d\ g\ a))\ (cnst\ d)\ (dom\ f)$
proof −
let $?upd = \lambda a.\ upd\ a\ (map\text{-}default\ d\ g\ a)$
let $?fr = \lambda A.\ fold\ ?upd\ (cnst\ d)\ A$
interpret *gwf*: *fun-left-comm ?upd* **by**(*rule upd-left-comm*)

from *fin anf fg* **show** *?thesis*
proof(*induct* $A \equiv dom\ f$ *arbitrary*: *f*)
case *empty*
from ‹$\{\} = dom\ f$› **have** $f = empty$ **by**(*auto simp add*: *dom-def intro*: *ext*)
thus *?case* **by**(*simp add*: *finfun-const-def upd-const-same*)
next
case (*insert* a' *A*)
note *IH* = ‹$\bigwedge f.\ \llbracket a \notin dom\ f;\ f \subseteq_m g;\ A = dom\ f \rrbracket \Longrightarrow upd\ a\ d\ (?fr\ (dom\ f)) = ?fr\ (dom\ f)$›
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 $?f' = f(a' := None)$
have $upd\ a\ d\ (?fr\ (insert\ a'\ A)) = upd\ a\ d\ (upd\ a'\ (map\text{-}default\ d\ g\ a')\ (?fr\ A))$
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'*: $map\text{-}default\ d\ g\ a' = map\text{-}default\ d\ f\ a'$ **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* ‹$a \notin dom\ ?f'$› ‹$?f' \subseteq_m g$› *A*]
also have $upd\ a'\ (map\text{-}default\ d\ f\ a')\ (?fr\ (dom\ (f(a' := None)))) = ?fr\ (dom\ f)$

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'* (*fold* (λa. *upd a* (*map-default d g a*)) (*cnst d*) (*dom f*))) =
upd a d'' (*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. *fold ?upd* (*cnst d*) *A*
interpret *gwf*: *fun-left-comm ?upd* **by**(*rule upd-left-comm*)

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

declare *finfun-simp* [*simp del*] *finfun-iff* [*iff del*] *finfun-intro* [*rule del*]

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 intro*: *ext*)

lemma *finite-rec-cong1*:
  assumes *f*: *fun-left-comm f* **and** *g*: *fun-left-comm g*
  and *fin*: *finite A*
  and *eq*: $\bigwedge a.\ a \in A \Longrightarrow f\ a = g\ a$
  shows *fold f z A* = *fold g z A*
proof –
  interpret *f*: *fun-left-comm f* **by**(*rule f*)
  interpret *g*: *fun-left-comm g* **by**(*rule g*)
  { fix *B*
    assume *BsubA*: $B \subseteq A$
    with *fin* **have** *finite B* **by**(*blast intro*: *finite-subset*)
    hence $B \subseteq A \Longrightarrow$ *fold f z B* = *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* = ‹$B \subseteq A \Longrightarrow$ *fold f z B* = *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** *fold f z B* = *fold g z B* **by** *blast* **}**
  thus *?thesis* **by** *blast*
qed

declare *finfun-simp* [*simp*] *finfun-iff* [*iff*] *finfun-intro* [*intro*]

lemma *finfun-rec-upd* [*simp*]:
  finfun-rec cnst upd ($f(^f\ a' := b')$) = *upd* a' b' (*finfun-rec cnst upd f*)
proof –
  obtain *b* **where** *b*: *b* = *finfun-default f* **by** *auto*
  let *?the* = $\lambda f\ g.\ f$ = *Abs-finfun* (*map-default b g*) $\wedge$ *finite* (*dom g*) $\wedge$ $b \notin$ *ran g*
  obtain *g* **where** *g*: *g* = *The* (*?the f*) **by** *blast*
  obtain *y* **where** *f*: *f* = *Abs-finfun 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 finite-finfun-default-aux*)

  let *?g* = (λa. *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*: *split-if-asm*)
next
fix g'
assume *?the f* g'
hence *fin'*: *finite* (*dom* g') **and** *ran'*: $b \notin$ *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* $\in$ *finfun map-default b* $g' \in$ *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 $?g' = (\lambda a.\ \mathit{Some}\ ((y(a' := b))\ a))$ |' $\{a.\ (y(a' := b))\ a \neq b\}$
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 $?b' = \lambda a.$ *case* $?g'$ *a of None* $\Rightarrow b$ | *Some* $b \Rightarrow b$
let $?b =$ *map-default b ?g*
from *upd-left-comm upd-left-comm fing'*
have *fold* ($\lambda a.$ *upd a* ($?b'$ *a*)) (*cnst b*) (*dom* $?g'$) $=$ *fold* ($\lambda a.$ *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*: *fun-left-comm* $\lambda a.$ *upd a* (*?b a*) **by**(*rule upd-left-comm*)
have *fold* ($\lambda a.$ *upd a* (*?b a*)) (*cnst b*) (*dom* $?g'$) $=$ *upd* a' b' (*fold* ($\lambda a.$ *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 intro*: *ext*)
from *True* **have** *a'ndomg*: $a' \notin$ *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'*: $a' \notin$ *dom* $?g'$ **by** *auto*

```
      have fold (λa. upd a (?b a)) (cnst b) (insert a' (dom ?g')) =
            upd a' (?b a') (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 (fold (λa. upd a (?b a)) (cnst b) (insert a' (dom ?g'))) =
              upd a' b (upd a' (?b a') (fold (λa. upd a (?b a)) (cnst b) (dom ?g')))
by simp
       also from b'b have g'leg: ?g' ⊆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(f a' := b'))) = ?g'
    proof(rule the-equality)
      from f y b b'b brang' fing' show ?the (f(f a' := b')) ?g'
        by(auto simp del: fun-upd-apply simp add: finfun-update-def)
    next
      fix g'
      assume ?the (f(f a' := b')) g'
      hence fin': finite (dom g') and ran': b ∉ ran g'
        and eq: f(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 intro: ext simp add:
map-default-def restrict-map-def)
    with f y have f-Abs: f(f a' := b') = Abs-finfun (map-default b ?g') by(auto
simp add: finfun-update-def)
    have g': The (?the (f(f a' := b'))) = ?g'
    proof
       from fing' bnrang' f-Abs show ?the (f(f a' := b')) ?g' by(auto simp add:
finfun-update-def restrict-map-def)
    next
```

fix g' **assume** *?the* $(f(^f\ a' := b'))\ g'$
hence f': $f(^f\ a' := b') = \textit{Abs-finfun}\ (\textit{map-default}\ b\ g')$
and *fin'*: *finite* $(\textit{dom}\ g')$ **and** *brang'*: $b \notin \textit{ran}\ g'$ **by** *auto*
from *fing' fin'* **have** *map-default b ?g'* $\in$ *finfun map-default b g'* $\in$ *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* $(((\lambda a.\ \textit{Some}\ (y\ a))\ |^{\backprime}\ \{a.\ y\ a \neq b\})(a' \mapsto b')) = \textit{insert}\ a'$ $(\textit{dom}\ ((\lambda a.\ \textit{Some}\ (y\ a))\ |^{\backprime}\ \{a.\ y\ a \neq b\}))$
by *auto*
show *?thesis*
proof(*cases* $y\ a' = b$)
case *True*
hence *a'ndomg*: $a' \notin$ *dom ?g* **by** *auto*
from *f y b'b True* **have** *yff*: y = *map-default b* (*?g'* $|^{\backprime}$ *dom ?g*)
by(*auto simp add*: *restrict-map-def map-default-def intro*!: *ext*)
hence f': f = *Abs-finfun* (*map-default b* (*?g'* $|^{\backprime}$ *dom ?g*)) **using** *f* **by** *simp*
interpret *g'wf*: *fun-left-comm* $\lambda a.$ *upd a* (*?b' a*) **by**(*rule upd-left-comm*)
from *upd-left-comm upd-left-comm fing*
have *fold* ($\lambda a.$ *upd a* (*?b a*)) (*cnst b*) (*dom ?g*) = *fold* ($\lambda a.$ *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*: *fun-left-comm* $\lambda a.$ *upd a* (*?b a*) **by**(*rule upd-left-comm*)
interpret *g'wf*: *fun-left-comm* $\lambda a.$ *upd a* (*?b' a*) **by**(*rule upd-left-comm*)
have *upd a' b'* (*fold* ($\lambda a.$ *upd a* (*?b a*)) (*cnst b*) (*insert a'* (*dom ?g''*))) =
upd a' b' (*upd a'* (*?b a'*) (*fold* ($\lambda a.$ *upd a* (*?b a*)) (*cnst b*) (*dom ?g''*)))
unfolding *gwf.fold-insert*[*OF fing'' a'ndomg''*] *f* **..**
also have *g''leg*: *?g* $|^{\backprime}$ *dom ?g''* $\subseteq_m$ *?g* **by**(*auto simp add*: *map-le-def*)
have *dom* (*?g* $|^{\backprime}$ *dom ?g''*) = *dom ?g''* **by** *auto*
note *map-default-update-twice*[**where** *d=b* **and** *f* = *?g* $|^{\backprime}$ *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 *fold* ($\lambda a.$ *upd a* (*?b a*)) (*cnst b*) (*dom ?g''*) = *fold* ($\lambda a.$ *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'* (*fold* ($\lambda a.$ *upd a* (*?b a*)) (*cnst b*) (*dom ?g''*)) =
upd a' (*?b' a'*) (*fold* ($\lambda a.$ *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'* (*fold* ($\lambda a.$ *upd a* (*?b a*)) (*cnst b*) (*dom ?g*)) =
fold ($\lambda a.$ *upd a* (*?b' a*)) (*cnst b*) (*dom ?g*)
unfolding *domg* **. }**
ultimately have *fold* ($\lambda a.$ *upd a* (*?b' a*)) (*cnst b*) (*insert a'* (*dom ?g*)) =
upd a' b' (*fold* ($\lambda a.$ *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

declare *finfun-simp* [*simp del*] *finfun-iff* [*iff del*] *finfun-intro* [*rule del*]

end

locale *finfun-rec-wf* = *finfun-rec-wf-aux* +
assumes *const-update-all*:
finite (*UNIV* :: *'a set*) $\Longrightarrow$ *fold* ($\lambda a.$ *upd a b'*) (*cnst b*) (*UNIV* :: *'a set*) = *cnst b'*
begin

declare *finfun-simp* [*simp*] *finfun-iff* [*iff*] *finfun-intro* [*intro*]

lemma *finfun-rec-const* [*simp*]:
finfun-rec cnst upd (λ^f *c*) = *cnst c*
proof(*cases finite* (*UNIV* :: *'a set*))
case *False*
hence *finfun-default* ((λ^f *c*) :: *'a* $\Rightarrow_f$ *'b*) = *c* **by**(*simp add*: *finfun-default-const*)
moreover have (*THE g* :: *'a* $\rightharpoonup$ *'b*. (λ^f *c*) = *Abs-finfun* (*map-default c g*) $\wedge$ *finite* (*dom g*) $\wedge$ $c \notin$ *ran g*) = *empty*
proof
show (λ^f *c*) = *Abs-finfun* (*map-default c empty*) $\wedge$ *finite* (*dom empty*) $\wedge$ $c \notin$ *ran empty*
by(*auto simp add*: *finfun-const-def*)
next
fix *g* :: *'a* $\rightharpoonup$ *'b*
assume (λ^f *c*) = *Abs-finfun* (*map-default c g*) $\wedge$ *finite* (*dom g*) $\wedge$ $c \notin$ *ran g*
hence *g*: (λ^f *c*) = *Abs-finfun* (*map-default c g*) **and** *fin*: *finite* (*dom g*) **and** *ran*: $c \notin$ *ran g* **by** *blast+*

from *g map-default-in-finfun*[*OF fin*, *of c*] **have** *map-default c g* = ($\lambda a.$ *c*)
by(*simp add*: *finfun-const-def*)
moreover have *map-default c empty* = ($\lambda a.$ *c*) **by** *simp*
ultimately show *g* = *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* ((λ^f *c*) :: $'a \Rightarrow_f 'b$) = *undefined* **by**(*simp add*: *finfun-default-const*)
let *?the* = $\lambda g :: 'a \rightharpoonup 'b.$ (λ^f *c*) = *Abs-finfun* (*map-default undefined g*) $\wedge$ *finite* (*dom g*) $\wedge$ *undefined* $\notin$ *ran g*
show *?thesis*
proof(*cases c* = *undefined*)
case *True*
have *the*: *The ?the* = *empty*
proof
from *True* **show** *?the empty* **by**(*auto simp add*: *finfun-const-def*)
next
fix *g'*
assume *?the g'*
hence *fg*: (λ^f *c*) = *Abs-finfun* (*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-finfun*)
with *fg* **have** *map-default undefined g'* = ($\lambda a.$ *c*)
by(*auto simp add*: *finfun-const-def intro*: *Abs-finfun-inject*[*THEN iffD1*])
with *True* **show** *g'* = *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
from *False True* **show** *?the* ($\lambda a :: 'a.$ *Some c*)
by(*auto simp add*: *map-default-def-raw finfun-const-def dom-def ran-def*)
next
fix *g'* :: $'a \rightharpoonup 'b$
assume *?the g'*
hence *fg*: (λ^f *c*) = *Abs-finfun* (*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-finfun*)
with *fg* **have** *map-default undefined g'* = ($\lambda a.$ *c*)
by(*auto simp add*: *finfun-const-def intro*: *Abs-finfun-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-raw*)

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

declare *finfun-simp* [*simp del*] *finfun-iff* [*iff del*] *finfun-intro* [*rule del*]

end

1.9 Weak induction rule and case analysis for FinFuns

declare *finfun-simp* [*simp*] *finfun-iff* [*iff*] *finfun-intro* [*intro*]

lemma *finfun-weak-induct* [*consumes 0*, *case-names const update*]:
assumes *const*: $\bigwedge b.\ P\ (\lambda^f\ b)$
and *update*: $\bigwedge f\ a\ b.\ P\ f \Longrightarrow P\ (f(^f\ a := b))$
shows $P\ x$
proof(*induct x rule*: *Abs-finfun-induct*)
case (*Abs-finfun y*)
then obtain b **where** *finite* $\{a.\ y\ a \neq b\}$ **unfolding** *finfun-def* **by** *blast*
thus *?case* **using** ‹$y \in$ *finfun*›
proof(*induct* $x \equiv \{a.\ y\ a \neq b\}$ *arbitrary*: y *rule*: *finite-induct*)
case *empty*
hence $\bigwedge a.\ y\ a = b$ **by** *blast*
hence $y = (\lambda a.\ b)$ **by**(*auto intro*: *ext*)
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 = ‹$\bigwedge y.\ [\![\ y \in finfun;\ A = \{a.\ y\ a \neq b\}\ ]\!] \Longrightarrow P\ (Abs\text{-}finfun\ y)$›
note y = ‹$y \in$ *finfun*›
with ‹*insert* $a\ A = \{a.\ y\ a \neq b\}$› ‹$a \notin A$›
have $y(a := b) \in finfun\ A = \{a'.\ (y(a := b))\ a' \neq b\}$ **by** *auto*
from $IH[OF\ this]$ **have** $P\ (finfun\text{-}update\ (Abs\text{-}finfun\ (y(a := b)))\ a\ (y\ a))$
by(*rule update*)
thus *?case* **using** y **unfolding** *finfun-update-def* **by** *simp*
qed
qed

declare *finfun-simp* [*simp del*] *finfun-iff* [*iff del*] *finfun-intro* [*rule del*]

lemma *finfun-exhaust-disj*: $(\exists\, b.\ x = finfun\text{-}const\ b) \lor (\exists\, f\ a\ b.\ x = finfun\text{-}update\ f\ a\ b)$
by(*induct x rule*: *finfun-weak-induct*) *blast+*

lemma *finfun-exhaust*:
obtains b **where** $x = (\lambda^f\ b)$
| $f\ a\ b$ **where** $x = f(^f\ a := b)$

by(*atomize-elim*)(*rule finfun-exhaust-disj*)

lemma *finfun-rec-unique*:
  fixes $f :: 'a \Rightarrow_f 'b \Rightarrow 'c$
  assumes c: $\bigwedge c.\ f\ (\lambda^f\ c) = cnst\ c$
  and u: $\bigwedge g\ a\ b.\ f\ (g(^f\ a := b)) = upd\ g\ a\ b\ (f\ g)$
  and c': $\bigwedge c.\ f'\ (\lambda^f\ c) = cnst\ c$
  and u': $\bigwedge g\ a\ b.\ f'\ (g(^f\ a := b)) = upd\ g\ a\ b\ (f'\ g)$
  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

definition *finfun-apply* :: $'a \Rightarrow_f 'b \Rightarrow 'a \Rightarrow 'b$ (-_f [*1000*] *1000*)
where [*code del*]: *finfun-apply* = ($\lambda f\ a.$ *finfun-rec* ($\lambda b.\ b$) ($\lambda a'\ b\ c.$ *if* ($a = a'$) *then* b *else* c) f)

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

interpretation *finfun-apply*: *finfun-rec-wf* $\lambda b.\ b\ \lambda a'\ b\ c.$ *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 *fun-left-comm* $\lambda a'.$ *If* ($a = a'$) b' **by**(*rule finfun-apply-aux.upd-left-comm*)
    from *fin* **have** *finite A* **by**(*auto intro*: *finite-subset*)
    hence *fold* ($\lambda a'.$ *If* ($a = a'$) b') b A = (*if* $a \in A$ *then* b' *else* b)
      by *induct auto* **}**
  from *this*[*of UNIV*] **show** *fold* ($\lambda a'.$ *If* ($a = a'$) b') b *UNIV* = b' **by** *simp*
qed

lemma *finfun-const-apply* [*simp*, *code*]: $(\lambda^f\ b)_f\ a = b$
by(*simp add*: *finfun-apply-def*)

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

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

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

declare *finfun-simp* [*simp*] *finfun-iff* [*iff*] *finfun-intro* [*intro*]

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

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

declare *finfun-simp* [*simp del*] *finfun-iff* [*iff del*] *finfun-intro* [*rule del*]

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

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

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

1.11 Function composition

definition *finfun-comp* :: $('a \Rightarrow 'b) \Rightarrow 'c \Rightarrow_f 'a \Rightarrow 'c \Rightarrow_f 'b$ (**infixr** $\circ_f$ *55*)
where [*code del*]: $g \circ_f f = \mathit{finfun\text{-}rec}\ (\lambda b.\ (\lambda^f\ g\ b))\ (\lambda a\ b\ c.\ c(^f\ a := g\ b))\ f$

interpretation *finfun-comp-aux*: *finfun-rec-wf-aux* $(\lambda b.\ (\lambda^f\ g\ b))\ (\lambda a\ b\ c.\ c(^f\ a := g\ b))$
by(*unfold-locales*)(*auto simp add*: *finfun-upd-apply intro*: *finfun-ext*)

interpretation *finfun-comp*: *finfun-rec-wf* $(\lambda b.\ (\lambda^f\ g\ b))\ (\lambda a\ b\ c.\ c(^f\ a := g\ b))$
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 *fold* $(\lambda(a :: 'c)\ c.\ c(^f\ a := g\ b'))\ (\lambda^f\ g\ b)\ A =$
Abs-finfun $(\lambda a.\ \mathit{if}\ a \in A\ \mathit{then}\ g\ b'\ \mathit{else}\ g\ b)$

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 expand-fun-eq fin*) **}**
from *this*[*of UNIV*] **show** *fold* $(\lambda(a :: 'c)\ c.\ c({}^f\ a := g\ b'))\ (\lambda^f\ g\ b)\ UNIV = (\lambda^f\ g\ b')$
by(*simp add*: *finfun-const-def*)
qed

lemma *finfun-comp-const* [*simp*, *code*]:
$g \circ_f (\lambda^f\ c) = (\lambda^f\ g\ c)$
by(*simp add*: *finfun-comp-def*)

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

lemma *finfun-comp-apply* [*simp*]:
$(g \circ_f f)_f = g \circ f_f$
by(*induct f rule*: *finfun-weak-induct*)(*auto simp add*: *finfun-upd-apply intro*: *ext*)

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

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

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

declare *finfun-simp* [*simp*] *finfun-iff* [*iff*] *finfun-intro* [*intro*]

lemma *finfun-comp-conv-comp*: $g \circ_f f =$ *Abs-finfun* ($g \circ$ *finfun-apply f*)
proof −
have $(\lambda f.\ g \circ_f f) = (\lambda f.$ *Abs-finfun* ($g \circ$ *finfun-apply f*))
proof(*rule finfun-rec-unique*)
{ fix *c* **show** *Abs-finfun* $(g \circ (\lambda^f\ c)_f) = (\lambda^f\ g\ c)$
by(*simp add*: *finfun-comp-def o-def*)(*simp add*: *finfun-const-def*) **}**
{ fix g' a b **show** *Abs-finfun* $(g \circ g'({}^f\ a := b)_f) =$ (*Abs-finfun* $(g \circ g'_f))({}^f\ a := g\ b)$
proof −
obtain *y* **where** *y*: $y \in$ *finfun* **and** g': $g' =$ *Abs-finfun y* **by**(*cases* g')
moreover hence $(g \circ g'_f) \in$ *finfun* **by**(*simp add*: *finfun-apply-Rep-finfun finfun-left-compose*)
moreover have $g \circ y(a := b) = (g \circ y)(a := g\ b)$ **by**(*auto intro*: *ext*)
ultimately show *?thesis* **by**(*simp add*: *finfun-comp-def finfun-update-def finfun-apply-Rep-finfun*)
qed }
qed *auto*
thus *?thesis* **by**(*auto simp add*: *expand-fun-eq*)

qed

declare *finfun-simp* [*simp del*] *finfun-iff* [*iff del*] *finfun-intro* [*rule del*]

definition *finfun-comp2* :: $'b \Rightarrow_f 'c \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow_f 'c$ (**infixr** $_f\circ$ *55*)
where [*code del*]: *finfun-comp2 g f* = *Abs-finfun* (*Rep-finfun g* ∘ *f*)

declare *finfun-simp* [*simp*] *finfun-iff* [*iff*] *finfun-intro* [*intro*]

lemma *finfun-comp2-const* [*code*, *simp*]: *finfun-comp2* (λ^f *c*) *f* = (λ^f *c*)
by(*simp add*: *finfun-comp2-def finfun-const-def comp-def*)

lemma *finfun-comp2-update*:
  assumes *inj*: *inj f*
  shows *finfun-comp2* (*g*(^{f} *b* := *c*)) *f* = (*if b* ∈ *range f then* (*finfun-comp2 g f*)(^{f} *inv f b* := *c*) *else finfun-comp2 g f*)
proof(*cases b* ∈ *range f*)
  case *True*
  from *inj* **have** $\bigwedge x$. (*Rep-finfun g*)(*f x* := *c*) ∘ *f* = (*Rep-finfun 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 (*Rep-finfun g*)(*b* := *c*) ∘ *f* = *Rep-finfun g* ∘ *f* **by**(*auto simp add*: *expand-fun-eq*)
  with *False* **show** *?thesis* **by**(*auto simp add*: *finfun-comp2-def finfun-update-def*)
qed

declare *finfun-simp* [*simp del*] *finfun-iff* [*iff del*] *finfun-intro* [*rule del*]

1.12 A type class for computing the cardinality of a type's universe

class *card-UNIV* =
  fixes *card-UNIV* :: *'a itself* ⇒ *nat*
  assumes *card-UNIV*: *card-UNIV x* = *card* (*UNIV* :: *'a set*)
begin

lemma *card-UNIV-neq-0-finite-UNIV*:
  card-UNIV x ≠ *0* ⟷ *finite* (*UNIV* :: *'a set*)
by(*simp add*: *card-UNIV card-eq-0-iff*)

lemma *card-UNIV-ge-0-finite-UNIV*:
  card-UNIV x > *0* ⟷ *finite* (*UNIV* :: *'a set*)
by(*auto simp add*: *card-UNIV intro*: *card-ge-0-finite finite-UNIV-card-ge-0*)

lemma *card-UNIV-eq-0-infinite-UNIV*:

card-UNIV x = 0 $\longleftrightarrow$ *infinite (UNIV :: 'a set)*
by(*simp add*: *card-UNIV card-eq-0-iff*)

definition *is-list-UNIV* :: *'a list* $\Rightarrow$ *bool*
where *is-list-UNIV xs* = (*let c* = *card-UNIV* (*TYPE('a)*) *in if c* = *0 then False else size* (*remdups xs*) = *c*)

lemma *is-list-UNIV-iff*:
  fixes *xs* :: *'a list*
  shows *is-list-UNIV xs* $\longleftrightarrow$ *set xs* = *UNIV*
proof
  assume *is-list-UNIV xs*
  hence *c*: *card-UNIV* (*TYPE('a)*) > *0* **and** *xs*: *size* (*remdups xs*) = *card-UNIV* (*TYPE('a)*)
    unfolding *is-list-UNIV-def* **by**(*simp-all add*: *Let-def split*: *split-if-asm*)
  from *c* **have** *fin*: *finite* (*UNIV* :: *'a set*) **by**(*auto simp add*: *card-UNIV-ge-0-finite-UNIV*)
  have *card* (*set* (*remdups xs*)) = *size* (*remdups xs*) **by**(*subst distinct-card*) *auto*
  also note *set-remdups*
  finally show *set xs* = *UNIV* **using** *fin* **unfolding** *xs card-UNIV* **by**−(*rule card-eq-UNIV-imp-eq-UNIV*)
next
  assume *xs*: *set xs* = *UNIV*
  from *finite-set*[*of xs*] **have** *fin*: *finite* (*UNIV* :: *'a set*) **unfolding** *xs* **.**
  hence *card-UNIV* (*TYPE* (*'a*)) $\neq$ *0* **unfolding** *card-UNIV-neq-0-finite-UNIV* **.**
  moreover have *size* (*remdups xs*) = *card* (*set* (*remdups xs*))
    by(*subst distinct-card*) *auto*
  ultimately show *is-list-UNIV xs* **using** *xs* **by**(*simp add*: *is-list-UNIV-def Let-def card-UNIV*)
qed

lemma *card-UNIV-eq-0-is-list-UNIV-False*:
  assumes *cU0*: *card-UNIV x* = *0*
  shows *is-list-UNIV* = (λ*xs*. *False*)
proof(*rule ext*)
  fix *xs* :: *'a list*
  from *cU0* **have** *infinite* (*UNIV* :: *'a set*)
    by(*auto simp only*: *card-UNIV-eq-0-infinite-UNIV*)
  moreover have *finite* (*set xs*) **by**(*rule finite-set*)
  ultimately have (*UNIV* :: *'a set*) $\neq$ *set xs* **by**(*auto simp del*: *finite-set*)
  thus *is-list-UNIV xs* = *False* **unfolding** *is-list-UNIV-iff* **by** *simp*
qed

end

1.13 Instantiations for *card-UNIV*

1.13.1 *nat*

instantiation *nat* :: *card-UNIV* **begin**

definition *card-UNIV-nat-def*:
card-UNIV-class.card-UNIV = (λa :: *nat itself*. *0*)

instance proof
fix *x* :: *nat itself*
show *card-UNIV x* = *card* (*UNIV* :: *nat set*)
unfolding *card-UNIV-nat-def* **by** *simp*
qed

end

1.13.2 *int*

instantiation *int* :: *card-UNIV* **begin**

definition *card-UNIV-int-def*:
card-UNIV-class.card-UNIV = (λa :: *int itself*. *0*)

instance proof
fix *x* :: *int itself*
show *card-UNIV x* = *card* (*UNIV* :: *int set*)
unfolding *card-UNIV-int-def* **by** *simp*
qed

end

1.13.3 *'a list*

instantiation *list* :: (*type*) *card-UNIV* **begin**

definition *card-UNIV-list-def*:
card-UNIV-class.card-UNIV = (λa :: *'a list itself*. *0*)

instance proof
fix *x* :: *'a list itself*
show *card-UNIV x* = *card* (*UNIV* :: *'a list set*)
unfolding *card-UNIV-list-def* **by**(*simp add*: *infinite-UNIV-listI*)
qed

end

1.13.4 *unit*

lemma *card-UNIV-unit*: *card* (*UNIV* :: *unit set*) = *1*
unfolding *UNIV-unit* **by** *simp*

instantiation *unit* :: *card-UNIV* **begin**

definition *card-UNIV-unit-def*:
card-UNIV-class.card-UNIV = (λa :: *unit itself*. *1*)

instance proof
  fix *x* :: *unit itself*
  show *card-UNIV x* = *card* (*UNIV* :: *unit set*)
    by(*simp add*: *card-UNIV-unit-def card-UNIV-unit*)
qed

end

1.13.5 *bool*

lemma *card-UNIV-bool*: *card* (*UNIV* :: *bool set*) = *2*
  unfolding *UNIV-bool* **by** *simp*

instantiation *bool* :: *card-UNIV* **begin**

definition *card-UNIV-bool-def*:
  card-UNIV-class.card-UNIV = (λ*a* :: *bool itself*. *2*)

instance proof
  fix *x* :: *bool itself*
  show *card-UNIV x* = *card* (*UNIV* :: *bool set*)
    by(*simp add*: *card-UNIV-bool-def card-UNIV-bool*)
qed

end

1.13.6 *char*

lemma *card-UNIV-char*: *card* (*UNIV* :: *char set*) = *256*
proof −
  from *enum-distinct*
  have *card* (*set* (*enum* :: *char list*)) = *length* (*enum* :: *char list*)
    by − (*rule distinct-card*)
  also have *set enum* = (*UNIV* :: *char set*) **by** *auto*
  also note *enum-chars*
  finally show *?thesis* **by** (*simp add*: *chars-def*)
qed

instantiation *char* :: *card-UNIV* **begin**

definition *card-UNIV-char-def*:
  card-UNIV-class.card-UNIV = (λ*a* :: *char itself*. *256*)

instance proof
  fix *x* :: *char itself*
  show *card-UNIV x* = *card* (*UNIV* :: *char set*)
    by(*simp add*: *card-UNIV-char-def card-UNIV-char*)
qed

end

1.13.7 $'a \times 'b$

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

definition *card-UNIV-product-def*:
card-UNIV-class.card-UNIV = (λa :: ($'a \times 'b$) *itself*. *card-UNIV* (*TYPE*($'a$)) $*$ *card-UNIV* (*TYPE*($'b$)))

instance proof
fix x :: ($'a \times 'b$) *itself*
show *card-UNIV* x = *card* (*UNIV* :: ($'a \times 'b$) *set*)
by(*simp add*: *card-UNIV-product-def card-UNIV UNIV-Times-UNIV*[*symmetric*] *card-cartesian-product del*: *UNIV-Times-UNIV*)
qed

end

1.13.8 $'a + 'b$

instantiation $+$ :: (*card-UNIV*, *card-UNIV*) *card-UNIV* **begin**

definition *card-UNIV-sum-def*:
card-UNIV-class.card-UNIV = (λa :: ($'a + 'b$) *itself*. *let ca* = *card-UNIV* (*TYPE*($'a$)); *cb* = *card-UNIV* (*TYPE*($'b$))
in if ca $\neq$ *0* $\wedge$ *cb* $\neq$ *0 then ca* + *cb else 0*)

instance proof
fix x :: ($'a + 'b$) *itself*
show *card-UNIV* x = *card* (*UNIV* :: ($'a + 'b$) *set*)
by (*auto simp add*: *card-UNIV-sum-def card-UNIV card-eq-0-iff UNIV-Plus-UNIV*[*symmetric*] *finite-Plus-iff Let-def card-Plus simp del*: *UNIV-Plus-UNIV dest*!: *card-ge-0-finite*)
qed

end

1.13.9 $'a \Rightarrow 'b$

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

definition *card-UNIV-fun-def*:
card-UNIV-class.card-UNIV = (λa :: ($'a \Rightarrow 'b$) *itself*. *let ca* = *card-UNIV* (*TYPE*($'a$)); *cb* = *card-UNIV* (*TYPE*($'b$))
in if ca $\neq$ *0* $\wedge$ *cb* $\neq$ *0* $\vee$ *cb* = *1 then cb* ^ *ca else 0*)

instance proof
fix x :: ($'a \Rightarrow 'b$) *itself*

{ **assume** *0* < *card* (*UNIV* :: $'a$ *set*)

```
    and 0 < card (UNIV :: 'b set)
    hence fina: finite (UNIV :: 'a set) and finb: finite (UNIV :: 'b set)
      by(simp-all only: card-ge-0-finite)
    from finite-distinct-list[OF finb] obtain bs
      where bs: set bs = (UNIV :: 'b set) and distb: distinct bs by blast
    from finite-distinct-list[OF fina] obtain as
      where as: set as = (UNIV :: 'a set) and dista: distinct as by blast
    have cb: card (UNIV :: 'b set) = length bs
      unfolding bs[symmetric] distinct-card[OF distb] ..
    have ca: card (UNIV :: 'a set) = length as
      unfolding as[symmetric] distinct-card[OF dista] ..
    let ?xs = map (λys. the o map-of (zip as ys)) (n-lists (length as) bs)
    have UNIV = set ?xs
    proof(rule UNIV-eq-I)
      fix f :: 'a ⇒ 'b
      from as have f = the ∘ map-of (zip as (map f as))
        by(auto simp add: map-of-zip-map intro: ext)
      thus f ∈ set ?xs using bs by(auto simp add: set-n-lists)
    qed
    moreover have distinct ?xs unfolding distinct-map
    proof(intro conjI distinct-n-lists distb inj-onI)
      fix xs ys :: 'b list
      assume xs: xs ∈ set (n-lists (length as) bs)
        and ys: ys ∈ set (n-lists (length as) bs)
        and eq: the ∘ map-of (zip as xs) = the ∘ map-of (zip as ys)
      from xs ys have [simp]: length xs = length as length ys = length as
        by(simp-all add: length-n-lists-elem)
      have map-of (zip as xs) = map-of (zip as ys)
      proof
        fix x
        from as bs have ∃y. map-of (zip as xs) x = Some y ∃y. map-of (zip as
ys) x = Some y
          by(simp-all add: map-of-zip-is-Some[symmetric])
        with eq show map-of (zip as xs) x = map-of (zip as ys) x
          by(auto dest: fun-cong[where x=x])
      qed
      with dista show xs = ys by(simp add: map-of-zip-inject)
    qed
    hence card (set ?xs) = length ?xs by(simp only: distinct-card)
    moreover have length ?xs = length bs ^ length as by(simp add: length-n-lists)
    ultimately have card (UNIV :: ('a ⇒ 'b) set) = card (UNIV :: 'b set) ^ card
(UNIV :: 'a set)
      using cb ca by simp }
  moreover {
    assume cb: card (UNIV :: 'b set) = Suc 0
    then obtain b where b: UNIV = {b :: 'b} by(auto simp add: card-Suc-eq)
    have eq: UNIV = {λx :: 'a. b ::'b}
    proof(rule UNIV-eq-I)
      fix x :: 'a ⇒ 'b
```

{ **fix** *y*
have *x y* ∈ *UNIV* **..**
hence *x y* = *b* **unfolding** *b* **by** *simp* }
thus *x* ∈ {λ*x*. *b*} **by**(*auto intro*: *ext*)
qed
have *card* (*UNIV* :: (′*a* ⇒ ′*b*) *set*) = *Suc 0* **unfolding** *eq* **by** *simp* }
ultimately show *card-UNIV x* = *card* (*UNIV* :: (′*a* ⇒ ′*b*) *set*)
unfolding *card-UNIV-fun-def card-UNIV Let-def*
by(*auto simp del*: *One-nat-def*)(*auto simp add*: *card-eq-0-iff dest*: *finite-fun-UNIVD2 finite-fun-UNIVD1*)
qed

end

1.13.10 ′*a option*

instantiation *option* :: (*card-UNIV*) *card-UNIV*
begin

definition *card-UNIV-option-def*:
card-UNIV-class.card-UNIV = (λ*a* :: ′*a option itself*. *let c* = *card-UNIV* (*TYPE*(′*a*))
in if c ≠ *0 then Suc c else 0*)

instance proof
fix *x* :: ′*a option itself*
show *card-UNIV x* = *card* (*UNIV* :: ′*a option set*)
unfolding *UNIV-option-conv*
by(*auto simp add*: *card-UNIV-option-def card-UNIV card-eq-0-iff Let-def intro*: *inj-Some dest*: *finite-imageD*)
(*subst card-insert-disjoint*, *auto simp add*: *card-eq-0-iff card-image inj-Some intro*: *finite-imageI card-ge-0-finite*)
qed

end

1.14 Universal quantification

definition *finfun-All-except* :: ′*a list* ⇒ ′*a* $\Rightarrow_f$ *bool* ⇒ *bool*
where [*code del*]: *finfun-All-except A P* ≡ ∀ *a*. *a* ∈ *set A* ∨ *P* $_f$ *a*

lemma *finfun-All-except-const*: *finfun-All-except A* (λ^f *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 (λ^f *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(^{f} *a* := *b*) = ((*a* ∈ *set A* ∨ *b*) ∧ *finfun-All-except* (*a* # *A*) *f*)

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

lemma *finfun-All-except-update-code* [*code*]:
  fixes $a :: 'a :: card\text{-}UNIV$
  shows *finfun-All-except* A (*finfun-update-code* $f\ a\ b$) $= ((a \in set\ A \lor b) \land$ *finfun-All-except* $(a \# A)\ f)$
by(*simp add*: *finfun-All-except-update*)

definition *finfun-All* :: $'a \Rightarrow_f bool \Rightarrow bool$
where *finfun-All* = *finfun-All-except* []

lemma *finfun-All-const* [*simp*]: *finfun-All* $(\lambda^f\ b) = b$
by(*simp add*: *finfun-All-def finfun-All-except-def*)

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

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

definition *finfun-Ex* :: $'a \Rightarrow_f bool \Rightarrow bool$
where *finfun-Ex* $P = Not$ (*finfun-All* $(Not \circ_f P))$

lemma *finfun-Ex-Ex*: *finfun-Ex* $P = Ex\ P_f$
unfolding *finfun-Ex-def finfun-All-All* **by** *simp*

lemma *finfun-Ex-const* [*simp*]: *finfun-Ex* $(\lambda^f\ b) = b$
by(*simp add*: *finfun-Ex-def*)

1.15 A diagonal operator for FinFuns

definition *finfun-Diag* :: $'a \Rightarrow_f 'b \Rightarrow 'a \Rightarrow_f 'c \Rightarrow 'a \Rightarrow_f ('b \times 'c)$ $((1'(\text{-},/\ \text{-}')^f)$ $[0,\ 0]\ 1000)$
where [*code del*]: *finfun-Diag* $f\ g$ = *finfun-rec* $(\lambda b.\ Pair\ b \circ_f g)\ (\lambda a\ b\ c.\ c(^f\ a := (b,\ g_f\ a)))\ f$

interpretation *finfun-Diag-aux*: *finfun-rec-wf-aux* $\lambda b.\ Pair\ b \circ_f g\ \lambda a\ b\ c.\ c(^f\ a := (b,\ g_f\ a))$
by(*unfold-locales*)(*simp-all add*: *expand-finfun-eq expand-fun-eq finfun-upd-apply*)

interpretation *finfun-Diag*: *finfun-rec-wf* $\lambda b.\ Pair\ b \circ_f g\ \lambda a\ b\ c.\ c(^f\ a := (b,\ g_f\ a))$
proof
  fix $b'\ b :: 'a$
  assume *fin*: *finite* $(UNIV :: 'c\ set)$
  { **fix** $A :: 'c\ set$
   interpret *fun-left-comm* $\lambda a\ c.\ c(^f\ a := (b',\ g_f\ a))$ **by**(*rule finfun-Diag-aux.upd-left-comm*)
    from *fin* **have** *finite A* **by**(*auto intro*: *finite-subset*)

hence *fold* $(\lambda a\ c.\ c({}^f\ a := (b',\ g_f\ a)))$ $(Pair\ b \circ_f g)$ $A =$
Abs-finfun $(\lambda a.\ (\mathit{if}\ a \in A\ \mathit{then}\ b'\ \mathit{else}\ b,\ g_f\ a))$
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 expand-fun-eq fin*) **}**
from *this*[*of UNIV*] **show** *fold* $(\lambda a\ c.\ c({}^f\ a := (b',\ g_f\ a)))$ $(Pair\ b \circ_f g)$ *UNIV* $=$ $Pair\ b' \circ_f g$
by(*simp add*: *finfun-const-def finfun-comp-conv-comp o-def*)
qed

lemma *finfun-Diag-const1*: $(\lambda^f\ b,\ g)^f = Pair\ b \circ_f g$
by(*simp add*: *finfun-Diag-def*)

Do not use $(\lambda^f\ ?b,\ ?g)^f = Pair\ ?b \circ_f\ ?g$ 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 $\circ_f$.

lemma *finfun-Diag-const-code* [*code*]:
$(\lambda^f\ b,\ \lambda^f\ c)^f = (\lambda^f\ (b,\ c))$
$(\lambda^f\ b,\ g({}^{fc}\ a := c))^f = (\lambda^f\ b,\ g)^f({}^{fc}\ a := (b,\ c))$
by(*simp-all add*: *finfun-Diag-const1*)

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

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

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

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

lemma *finfun-Diag-update-update*:
$(f({}^f\ a := b),\ g({}^f\ a' := c))^f = (\mathit{if}\ a = a'\ \mathit{then}\ (f,\ g)^f({}^f\ a := (b,\ c))\ \mathit{else}\ (f,\ g)^f({}^f\ a := (b,\ g_f\ a))({}^f\ a' := (f_f\ a',\ c)))$
by(*auto simp add*: *finfun-Diag-update1 finfun-Diag-update2*)

lemma *finfun-Diag-apply* [*simp*]: $(f, g)^f{}_f = (\lambda x.\ (f_f\ x,\ g_f\ x))$
by(*induct f rule*: *finfun-weak-induct*)(*auto simp add*: *finfun-Diag-const1 finfun-Diag-update1 finfun-upd-apply intro*: *ext*)

declare *finfun-simp* [*simp*] *finfun-iff* [*iff*] *finfun-intro* [*intro*]

lemma *finfun-Diag-conv-Abs-finfun*:
$(f, g)^f = \mathit{Abs\text{-}finfun}\ ((\lambda x.\ (\mathit{Rep\text{-}finfun}\ f\ x,\ \mathit{Rep\text{-}finfun}\ g\ x)))$
proof −
have $(\lambda f :: 'a \Rightarrow_f 'b.\ (f, g)^f) = (\lambda f.\ \mathit{Abs\text{-}finfun}\ ((\lambda x.\ (\mathit{Rep\text{-}finfun}\ f\ x,\ \mathit{Rep\text{-}finfun}\ g\ x))))$
proof(*rule finfun-rec-unique*)
{ **fix** c **show** $\mathit{Abs\text{-}finfun}\ (\lambda x.\ (\mathit{Rep\text{-}finfun}\ (\lambda^f\ c)\ x,\ \mathit{Rep\text{-}finfun}\ g\ x)) = \mathit{Pair}\ c \circ_f g$
by(*simp add*: *finfun-comp-conv-comp finfun-apply-Rep-finfun o-def finfun-const-def*)
}
{ **fix** $g'\ a\ b$
show $\mathit{Abs\text{-}finfun}\ (\lambda x.\ (\mathit{Rep\text{-}finfun}\ g'(^f\ a := b)\ x,\ \mathit{Rep\text{-}finfun}\ g\ x)) =$
$(\mathit{Abs\text{-}finfun}\ (\lambda x.\ (\mathit{Rep\text{-}finfun}\ g'\ x,\ \mathit{Rep\text{-}finfun}\ g\ x)))(^f\ a := (b,\ g_f\ a))$
by(*auto simp add*: *finfun-update-def expand-fun-eq finfun-apply-Rep-finfun simp del*: *fun-upd-apply*) *simp* }
qed(*simp-all add*: *finfun-Diag-const1 finfun-Diag-update1*)
thus *?thesis* **by**(*auto simp add*: *expand-fun-eq*)
qed

declare *finfun-simp* [*simp del*] *finfun-iff* [*iff del*] *finfun-intro* [*rule del*]

lemma *finfun-Diag-eq*: $(f, g)^f = (f', g')^f \longleftrightarrow f = f' \land g = g'$
by(*auto simp add*: *expand-finfun-eq expand-fun-eq*)

definition *finfun-fst* :: $'a \Rightarrow_f ('b \times 'c) \Rightarrow 'a \Rightarrow_f 'b$
where [*code*]: $\mathit{finfun\text{-}fst}\ f = \mathit{fst} \circ_f f$

lemma *finfun-fst-const*: $\mathit{finfun\text{-}fst}\ (\lambda^f\ bc) = (\lambda^f\ \mathit{fst}\ bc)$
by(*simp add*: *finfun-fst-def*)

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

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

lemma *finfun-fst-conv* [*simp*]: $\mathit{finfun\text{-}fst}\ (f, g)^f = f$
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* = (λf. *Abs-finfun* (*fst o Rep-finfun* f))
by(*simp add*: *finfun-fst-def-raw finfun-comp-conv-comp finfun-apply-Rep-finfun*)

definition *finfun-snd* :: $'a \Rightarrow_f ('b \times 'c) \Rightarrow 'a \Rightarrow_f 'c$
where [*code*]: *finfun-snd* f = *snd* $\circ_f$ f

lemma *finfun-snd-const*: *finfun-snd* (λ^f *bc*) = (λ^f *snd bc*)
by(*simp add*: *finfun-snd-def*)

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

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

lemma *finfun-snd-conv* [*simp*]: *finfun-snd* $(f, g)^f = g$
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* = (λf. *Abs-finfun* (*snd o Rep-finfun* f))
by(*simp add*: *finfun-snd-def-raw finfun-comp-conv-comp finfun-apply-Rep-finfun*)

lemma *finfun-Diag-collapse* [*simp*]: (*finfun-fst* f, *finfun-snd* $f)^f = f$
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*)

1.16 Currying for FinFuns

definition *finfun-curry* :: $('a \times 'b) \Rightarrow_f 'c \Rightarrow 'a \Rightarrow_f 'b \Rightarrow_f 'c$
where [*code del*]: *finfun-curry* = *finfun-rec* (*finfun-const* $\circ$ *finfun-const*) ($\lambda(a, b)$ c f. $f(^f$ a := $(f_f$ $a)(^f$ b := c)))

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

declare *finfun-simp* [*simp*] *finfun-iff* [*iff*] *finfun-intro* [*intro*]

interpretation *finfun-curry*: *finfun-rec-wf finfun-const* $\circ$ *finfun-const* $\lambda(a,\ b)\ c\ f.\ f(^f\ a := (f_f\ a)(^f\ b := c))$
proof(*unfold-locales*)
  fix $b'\ b :: 'b$
  assume *fin*: *finite* (*UNIV* :: ($'c \times 'a$) *set*)
  hence *fin1*: *finite* (*UNIV* :: $'c$ *set*) **and** *fin2*: *finite* (*UNIV* :: $'a$ *set*)
    unfolding *UNIV-Times-UNIV*[*symmetric*]
    by(*fastsimp 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 $A :: ('c \times 'a)$ *set*
    interpret *fun-left-comm* $\lambda a :: 'c \times 'a.\ (\lambda(a,\ b)\ c\ f.\ f(^f\ a := (f_f\ a)(^f\ b := c)))\ a\ b'$
      by(*rule finfun-curry-aux.upd-left-comm*)
    from *fin* **have** *finite A* **by**(*auto intro*: *finite-subset*)
    hence *fold* $(\lambda a :: 'c \times 'a.\ (\lambda(a,\ b)\ c\ f.\ f(^f\ a := (f_f\ a)(^f\ b := c)))\ a\ b')$ ((*finfun-const* $\circ$ *finfun-const*) b) A = *Abs-finfun* ($\lambda a.$ *Abs-finfun* ($\lambda b''.$ *if* $(a,\ b'') \in A$ *then* b' *else* b))
      by *induct* (*simp-all*, *auto simp add*: *finfun-update-def finfun-const-def split-def finfun-apply-Rep-finfun intro*!: *arg-cong*[**where** *f*=*Abs-finfun*] *ext*) **}**
  from *this*[*of UNIV*]
  show *fold* $(\lambda a :: 'c \times 'a.\ (\lambda(a,\ b)\ c\ f.\ f(^f\ a := (f_f\ a)(^f\ b := c)))\ a\ b')$ ((*finfun-const* $\circ$ *finfun-const*) b) *UNIV* = (*finfun-const* $\circ$ *finfun-const*) b'
    by(*simp add*: *finfun-const-def*)
qed

declare *finfun-simp* [*simp del*] *finfun-iff* [*iff del*] *finfun-intro* [*rule del*]

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

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

declare *finfun-simp* [*simp*] *finfun-iff* [*iff*] *finfun-intro* [*intro*]

lemma *finfun-Abs-finfun-curry*: **assumes** *fin*: $f \in$ *finfun*
  shows ($\lambda a.$ *Abs-finfun* (*curry f a*)) $\in$ *finfun*
proof −
  from *fin* **obtain** c **where** c: *finite* $\{ab.\ f\ ab \neq c\}$ **unfolding** *finfun-def* **by** *blast*
  have $\{a.\ \exists b.\ f\ (a,\ b) \neq c\}$ = *fst* ‘ $\{ab.\ f\ ab \neq c\}$ **by**(*force*)
  hence $\{a.$ *curry f a* $\neq (\lambda x.\ c)\}$ = *fst* ‘ $\{ab.\ f\ ab \neq c\}$
    by(*auto simp add*: *curry-def expand-fun-eq*)
  with *fin c* **have** *finite* $\{a.$ *Abs-finfun* (*curry f a*) $\neq (\lambda^f\ c)\}$

by(*simp add*: *finfun-const-def finfun-curry*)
thus *?thesis* **unfolding** *finfun-def* **by** *auto*
qed

lemma *finfun-curry-conv-curry*:
fixes f :: $('a \times 'b) \Rightarrow_f 'c$
shows *finfun-curry* f = *Abs-finfun* (λa. *Abs-finfun* (*curry* (*Rep-finfun* f) a))
proof −
have *finfun-curry* = (λf :: $('a \times 'b) \Rightarrow_f 'c$. *Abs-finfun* ($\lambda a$. *Abs-finfun* (*curry* (*Rep-finfun* f) a)))
proof(*rule finfun-rec-unique*)
{ fix c **show** *finfun-curry* (λ^f c) = (λ^f λ^f c) **by** *simp* **}**
{ fix f a c **show** *finfun-curry* (f(^{f} a := c)) = (*finfun-curry* f)(^{f} *fst* a := ((*finfun-curry* f)$_f$ (*fst* a))(^{f} *snd* a := c))
by(*cases* a) *simp* **}**
{ fix c **show** *Abs-finfun* (λa. *Abs-finfun* (*curry* (*Rep-finfun* (λ^f c)) a)) = (λ^f λ^f c)
by(*simp add*: *finfun-curry-def finfun-const-def curry-def*) **}**
{ fix g a b
show *Abs-finfun* (λaa. *Abs-finfun* (*curry* (*Rep-finfun* g(^{f} a := b)) aa)) =
(*Abs-finfun* (λa. *Abs-finfun* (*curry* (*Rep-finfun* g) a)))(^{f}
fst a := ((*Abs-finfun* (λa. *Abs-finfun* (*curry* (*Rep-finfun* g) a)))$_f$ (*fst* a))(^{f} *snd* a := b))
by(*cases* a)(*auto intro*!: *ext arg-cong*[**where** f=*Abs-finfun*] *simp add*: *finfun-curry-def finfun-update-def finfun-apply-Rep-finfun finfun-curry finfun-Abs-finfun-curry*)
}
qed
thus *?thesis* **by**(*auto simp add*: *expand-fun-eq*)
qed

1.17 Executable equality for FinFuns

lemma *eq-finfun-All-ext*: (f = g) $\longleftrightarrow$ *finfun-All* (($\lambda(x, y)$. $x = y$) $\circ_f$ $(f, g)^f$)
by(*simp add*: *expand-finfun-eq expand-fun-eq finfun-All-All o-def*)

instantiation *finfun* :: ({*card-UNIV*,*eq*},*eq*) *eq* **begin**
definition *eq-finfun-def*: *eq-class.eq* f g $\longleftrightarrow$ *finfun-All* (($\lambda(x, y)$. $x = y$) $\circ_f$ $(f, g)^f$)
instance by(*intro-classes*)(*simp add*: *eq-finfun-All-ext eq-finfun-def*)
end

1.18 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*]: *finfun-clearjunk* (λ^f b) = (λ^f b)
by *simp*

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

end

2 Sets modelled as FinFuns

theory *FinFunSet* **imports** *FinFun* **begin**

Instantiate FinFun predicates just like predicates as sets in Set.thy

types $'a\ set_f = 'a \Rightarrow_f bool$

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

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

definition
less-fun-def: $(f{::}'a \Rightarrow_f 'b) < g \longleftrightarrow f \le g \wedge f \neq g$

instance ..

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

end

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

definition
finfun-diff-def: $A - B = split\ (op\ -) \circ_f (A, B)^f$

instance ..

end

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

definition
finfun-Compl-def: $-\ A = uminus \circ_f A$

instance ..

end

Replicate set operations for FinFuns

definition *finfun-empty* :: $'a\ set_f$ $(\{\}_f)$
where $\{\}_f \equiv (\lambda^f\ False)$

definition *finfun-insert* :: $'a \Rightarrow 'a\ set_f \Rightarrow 'a\ set_f$ $(insert_f)$
where $insert_f\ a\ A = A(^f\ a := True)$

definition *finfun-mem* :: $'a \Rightarrow 'a\ set_f \Rightarrow bool$ (-/ $\in_f$ - [50, 51] 50)
where $a \in_f A = A_f\ a$

definition *finfun-UNIV* :: $'a\ set_f$
where $finfun\text{-}UNIV = (\lambda^f\ True)$

definition *finfun-Un* :: $'a\ set_f \Rightarrow 'a\ set_f \Rightarrow 'a\ set_f$ (**infixl** $\cup_f$ 65)
where $A \cup_f B = split\ (op\ \vee) \circ_f (A, B)^f$

definition *finfun-Int* :: $'a \Rightarrow_f bool \Rightarrow 'a \Rightarrow_f bool \Rightarrow 'a \Rightarrow_f bool$ (**infixl** $\cap_f$ 65)
where $A \cap_f B = split\ (op\ \wedge) \circ_f (A, B)^f$

abbreviation *finfun-subset-eq* :: $'a\ set_f \Rightarrow 'a\ set_f \Rightarrow bool$ **where**
finfun-subset-eq $\equiv$ *less-eq*

abbreviation
finfun-subset :: $'a\ set_f \Rightarrow 'a\ set_f \Rightarrow bool$ **where**
finfun-subset $\equiv$ *less*

notation (**output**)
finfun-subset (*op* $<_f$) **and**
finfun-subset ((-/ $<_f$ -) [50, 51] 50) **and**
finfun-subset-eq (*op* $<=_f$) **and**
finfun-subset-eq ((-/ $<=_f$ -) [50, 51] 50)

notation (*xsymbols*)
finfun-subset (*op* $\subset_f$) **and**
finfun-subset ((-/ $\subset_f$ -) [50, 51] 50) **and**
finfun-subset-eq (*op* $\subseteq_f$) **and**
finfun-subset-eq ((-/ $\subseteq_f$ -) [50, 51] 50)

notation (*HTML* **output**)
finfun-subset (*op* $\subset_f$) **and**
finfun-subset ((-/ $\subset_f$ -) [50, 51] 50) **and**
finfun-subset-eq (*op* $\subseteq_f$) **and**
finfun-subset-eq ((-/ $\subseteq_f$ -) [50, 51] 50)

lemma *finfun-mem-empty* [*simp*]: $a \in_f \{\}_f = False$
by(*simp add*: *finfun-mem-def finfun-empty-def*)

lemma *finfun-subsetI* [*intro*!]: (!!*x*. $x \in_f A$ ==> $x \in_f B$) ==> $A \subseteq_f B$
by(*auto simp add*: *finfun-mem-def le-finfun-def le-bool-def*)

lemma *finfun-subsetD* [*elim*]: $A \subseteq_f B$ ==> $c \in_f A$ ==> $c \in_f B$
by(*simp add*: *finfun-mem-def le-finfun-def le-bool-def*)

lemma *finfun-subset-refl* [*simp*]: $A \subseteq_f A$
by *fast*

lemma *finfun-set-ext*: (!!*x*. $(x \in_f A) = (x \in_f B)$) $\Longrightarrow A = B$
by(*simp add*: *expand-finfun-eq finfun-mem-def expand-fun-eq*)

lemma *finfun-subset-antisym* [*intro*!]: $A \subseteq_f B$ ==> $B \subseteq_f A$ ==> $A = B$
by (*iprover intro*: *finfun-set-ext finfun-subsetD*)

lemma *finfun-Compl-iff* [*simp*]: $(c \in_f -A) = (\neg\ c \in_f A)$
by (*simp add*: *finfun-mem-def finfun-Compl-def bool-Compl-def*)

lemma *finfun-Un-iff* [*simp*]: $(c \in_f A \cup_f B) = (c \in_f A \mid c \in_f B)$
by (*simp add*: *finfun-Un-def finfun-mem-def*)

lemma *finfun-Int-iff* [*simp*]: $(c \in_f A \cap_f B) = (c \in_f A\ \&\ c \in_f B)$
by(*simp add*: *finfun-Int-def finfun-mem-def*)

lemma *finfun-Diff-iff* [*simp*]: $(c \in_f A - B) = (c \in_f A\ \&\ \neg\ c \in_f B)$
by (*simp add*: *finfun-mem-def finfun-diff-def bool-diff-def*)

lemma *finfun-insert-iff* [*simp*]: $(a \in_f insert_f\ b\ A) = (a = b \mid a \in_f A)$
by(*simp add*: *finfun-insert-def finfun-mem-def finfun-upd-apply*)

A tail-recursive function that never terminates in the code generator

definition *loop-counting* :: $nat \Rightarrow (unit \Rightarrow 'a) \Rightarrow 'a$
where [*simp*, *code del*]: *loop-counting n f* = *f* ()

lemma *loop-counting-code* [*code*]: *loop-counting n* = *loop-counting* (*Suc n*)
by(*simp add*: *expand-fun-eq*)

definition *loop* :: $(unit \Rightarrow 'a) \Rightarrow 'a$
where [*simp*, *code del*]: *loop f* = *f* ()

lemma *loop-code* [*code*]: *loop* = *loop-counting 0*
by(*simp add*: *expand-fun-eq*)

lemma *mem-finfun-apply-conv*: $x \in f_f \longleftrightarrow f_f\ x$
by(*simp add*: *mem-def*)

Bounded quantification.

Warning: *finfun-Ball* and *finfun-Ex* may fail to terminate, they should not be used for quickcheck

definition *finfun-Ball-except* :: $'a$ *list* $\Rightarrow$ $'a$ set_f $\Rightarrow$ $('a \Rightarrow bool) \Rightarrow bool$
where [*code del*]: *finfun-Ball-except xs A P* = $(\forall a \in A_f.\ a \in set\ xs \lor P\ a)$

lemma *finfun-Ball-except-const*:
finfun-Ball-except xs $(\lambda^f\ b)$ $P \longleftrightarrow \neg\ b \lor set\ xs = UNIV \lor loop\ (\lambda u.\ finfun\text{-}Ball\text{-}except\ xs\ (\lambda^f\ b)\ P)$
by(*auto simp add*: *finfun-Ball-except-def mem-finfun-apply-conv*)

lemma *finfun-Ball-except-const-finfun-UNIV-code* [*code*]:
finfun-Ball-except xs $(\lambda^f\ b)$ $P \longleftrightarrow \neg\ b \lor is\text{-}list\text{-}UNIV\ xs \lor loop\ (\lambda u.\ finfun\text{-}Ball\text{-}except\ xs\ (\lambda^f\ b)\ P)$
by(*auto simp add*: *finfun-Ball-except-def is-list-UNIV-iff mem-finfun-apply-conv*)

lemma *finfun-Ball-except-update*:
finfun-Ball-except xs $(A({}^f\ a := b))$ $P = ((a \in set\ xs \lor (b \longrightarrow P\ a)) \land finfun\text{-}Ball\text{-}except\ (a\ \#\ xs)\ A\ P)$
by(*fastsimp simp add*: *finfun-Ball-except-def mem-finfun-apply-conv finfun-upd-apply dest*: *bspec split*: *split-if-asm*)

lemma *finfun-Ball-except-update-code* [*code*]:
fixes a :: $'a$ :: *card-UNIV*
shows *finfun-Ball-except xs* (*finfun-update-code f a b*) $P = ((a \in set\ xs \lor (b \longrightarrow P\ a)) \land finfun\text{-}Ball\text{-}except\ (a\ \#\ xs)\ f\ P)$
by(*simp add*: *finfun-Ball-except-update*)

definition *finfun-Ball* :: $'a$ set_f $\Rightarrow$ $('a \Rightarrow bool) \Rightarrow bool$
where [*code del*]: *finfun-Ball A P* = *Ball* (A_f) 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* $\Rightarrow$ $'a$ set_f $\Rightarrow$ $('a \Rightarrow bool) \Rightarrow bool$
where [*code del*]: *finfun-Bex-except xs A P* = $(\exists a \in A_f.\ a \notin set\ xs \land P\ a)$

lemma *finfun-Bex-except-const*: *finfun-Bex-except xs* $(\lambda^f\ b)$ $P \longleftrightarrow b \land set\ xs \neq UNIV \land loop\ (\lambda u.\ finfun\text{-}Bex\text{-}except\ xs\ (\lambda^f\ b)\ P)$
by(*auto simp add*: *finfun-Bex-except-def mem-finfun-apply-conv*)

lemma *finfun-Bex-except-const-finfun-UNIV-code* [*code*]:
finfun-Bex-except xs $(\lambda^f\ b)$ $P \longleftrightarrow b \land \neg\ is\text{-}list\text{-}UNIV\ xs \land loop\ (\lambda u.\ finfun\text{-}Bex\text{-}except\ xs\ (\lambda^f\ b)\ P)$
by(*auto simp add*: *finfun-Bex-except-def is-list-UNIV-iff mem-finfun-apply-conv*)

lemma *finfun-Bex-except-update*:
finfun-Bex-except xs $(A({}^f\ a := b))$ $P \longleftrightarrow (a \notin set\ xs \land b \land P\ a) \lor finfun\text{-}Bex\text{-}except\ (a\ \#\ xs)\ A\ P$
by(*fastsimp simp add*: *finfun-Bex-except-def mem-finfun-apply-conv finfun-upd-apply*

dest: *bspec split*: *split-if-asm*)

lemma *finfun-Bex-except-update-code* [*code*]:
fixes a :: $'a$ :: *card-UNIV*
shows *finfun-Bex-except xs* (*finfun-update-code f a b*) $P \longleftrightarrow ((a \notin set\ xs \wedge b \wedge P\ a) \vee$ *finfun-Bex-except* (a # xs) f P)
by(*simp add*: *finfun-Bex-except-update*)

definition *finfun-Bex* :: $'a\ set_f \Rightarrow ('a \Rightarrow bool) \Rightarrow bool$
where [*code del*]: *finfun-Bex* A P = *Bex* (A_f) 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 set operations by finfun set operations where possible

lemma *iso-finfun-mem-mem* [*code-inline*]: $x \in A_f \longleftrightarrow x \in_f A$
by(*auto simp add*: *mem-def finfun-mem-def*)

declare *iso-finfun-mem-mem* [*simp*]

lemma *iso-finfun-subset-subset* [*code-inline*]:
$A_f \subseteq B_f \longleftrightarrow A \subseteq_f B$
by(*auto*)

lemma *iso-finfun-eq* [*code-inline*]:
fixes A :: $'a \Rightarrow_f bool$
shows $A_f = B_f \longleftrightarrow A = B$
by(*simp add*: *expand-finfun-eq*)

lemma *iso-finfun-Un-Un* [*code-inline*]:
$A_f \cup B_f = (A \cup_f B)_f$
by(*auto*)

lemma *iso-finfun-Int-Int* [*code-inline*]:
$A_f \cap B_f = (A \cap_f B)_f$
by(*auto*)

lemma *iso-finfun-empty-conv* [*code-inline*]:
$\{\} = \{\}_{f\,f}$
by(*auto*)

lemma *iso-finfun-insert-insert* [*code-inline*]:
insert a $A_f = (insert_f\ a\ A)_f$
by(*auto*)

lemma *iso-finfun-Compl-Compl* [*code-inline*]:
fixes A :: $'a\ set_f$
shows $-\ A_f = (-\ A)_f$
by(*auto*)

lemma *iso-finfun-diff-diff* [*code-inline*]:
fixes $A :: 'a\ set_f$
shows $A_f - B_f = (A - B)_f$
by(*auto*)

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

lemma *iso-finfun-Ball-Ball*:
$Ball\ (A_f)\ P \longleftrightarrow finfun\text{-}Ball\ A\ P$
by(*simp add*: *finfun-Ball-def*)

lemma *iso-finfun-Bex-Bex*:
$Bex\ (A_f)\ P \longleftrightarrow finfun\text{-}Bex\ A\ P$
by(*simp add*: *finfun-Bex-def*)

declare *iso-finfun-mem-mem* [*simp del*]

end