Finfuns

Andreas Lochbihler

March 17, 2025

Contents

1 Almost everywhere constant functions **1**
1.1 The *map-default* operation 1
1.2 The finfun type 2
1.3 Kernel functions for type $'a \Rightarrow f\ 'b$ 4
1.4 Code generator setup 4
1.5 Setup for quickcheck 4
1.6 *finfun-update* as instance of *comp-fun-commute* 5
1.7 Default value for FinFuns 5
1.8 Recursion combinator and well-formedness conditions 6
1.9 Weak induction rule and case analysis for FinFuns 7
1.10 Function application 8
1.11 Function composition 9
1.12 Universal quantification 10
1.13 A diagonal operator for FinFuns 11
1.14 Currying for FinFuns 13
1.15 Executable equality for FinFuns 14
1.16 An operator that explicitly removes all redundant updates in the generated representations 14
1.17 The domain of a FinFun as a FinFun 14
1.18 The domain of a FinFun as a sorted list 15
1.18.1 Bundles for concrete syntax 17

2 Predicates modelled as FinFuns **17**

1 Almost everywhere constant functions

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

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

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

1.1 The *map-default* operation

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

lemma *map-default-delete* [*simp*]:
map-default b (*f*(*a* := *None*)) = (*map-default b f*)(*a* := *b*)
⟨*proof*⟩

lemma *map-default-insert*:
map-default b (*f*(*a* $\mapsto$ *b'*)) = (*map-default b f*)(*a* := *b'*)
⟨*proof*⟩

lemma *map-default-empty* [*simp*]: *map-default b Map.empty* = ($\lambda a.\ b$)
⟨*proof*⟩

lemma *map-default-inject*:
fixes *g g'* :: $'a \rightharpoonup 'b$
assumes *infin-eq*: $\neg$ *finite* (*UNIV* :: *'a set*) $\vee$ *b* = *b'*
and *fin*: *finite* (*dom g*) **and** *b*: *b* $\notin$ *ran g*
and *fin'*: *finite* (*dom g'*) **and** *b'*: *b'* $\notin$ *ran g'*
and *eq'*: *map-default b g* = *map-default b' g'*
shows *b* = *b'* *g* = *g'*
⟨*proof*⟩

1.2 The finfun type

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

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

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

setup-lifting *type-definition-finfun*

lemma *fun-upd-finfun*: *y*(*a* := *b*) $\in$ *finfun* $\longleftrightarrow$ *y* $\in$ *finfun*
⟨*proof*⟩

lemma *const-finfun*: ($\lambda x.\ a$) $\in$ *finfun*

⟨proof⟩

lemma *finfun-left-compose*:
  assumes $y \in$ *finfun*
  shows $g \circ y \in$ *finfun*
⟨proof⟩

lemma assumes $y \in$ *finfun*
  shows *fst-finfun*: *fst* $\circ$ $y \in$ *finfun*
  and *snd-finfun*: *snd* $\circ$ $y \in$ *finfun*
⟨proof⟩

lemma *map-of-finfun*: *map-of xs* $\in$ *finfun*
⟨proof⟩

lemma *Diag-finfun*: $(\lambda x.\ (f\ x,\ g\ x)) \in$ *finfun* $\longleftrightarrow f \in$ *finfun* $\wedge\ g \in$ *finfun*
⟨proof⟩

lemma *finfun-right-compose*:
  assumes *g*: $g \in$ *finfun* **and** *inj*: *inj f*
  shows *g o f* $\in$ *finfun*
⟨proof⟩

lemma *finfun-curry*:
  assumes *fin*: $f \in$ *finfun*
  shows *curry f* $\in$ *finfun curry f a* $\in$ *finfun*
⟨proof⟩

bundle *finfun*
begin

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

end

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

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

fixes *x y* :: (′*a* :: *finite*) ⇒ ′*b*
shows *Abs-finfun x* = *Abs-finfun y* ⟷ *x* = *y*
⟨*proof*⟩

lemma *Abs-finfun-inj-finite*:
assumes *fin*: *finite* (*UNIV* :: ′*a set*)
shows *inj* (*Abs-finfun* :: (′*a* ⇒ ′*b*) ⇒ ′*a* ⇒*f* ′*b*)
⟨*proof*⟩

lemma *Abs-finfun-inverse-finite*:
fixes *x* :: ′*a* ⇒ ′*b*
assumes *fin*: *finite* (*UNIV* :: ′*a set*)
shows *finfun-apply* (*Abs-finfun x*) = *x*
including *finfun*
⟨*proof*⟩

lemma *Abs-finfun-inverse-finite-class*:
fixes *x* :: (′*a* :: *finite*) ⇒ ′*b*
shows *finfun-apply* (*Abs-finfun x*) = *x*
⟨*proof*⟩

lemma *finfun-eq-finite-UNIV*: *finite* (*UNIV* :: ′*a set*) ⟹ (*finfun* :: (′*a* ⇒ ′*b*) *set*) = *UNIV*
⟨*proof*⟩

lemma *finfun-finite-UNIV-class*: *finfun* = (*UNIV* :: (′*a* :: *finite* ⇒ ′*b*) *set*)
⟨*proof*⟩

lemma *map-default-in-finfun*:
assumes *fin*: *finite* (*dom f*)
shows *map-default b f* ∈ *finfun*
⟨*proof*⟩

lemma *finfun-cases-map-default*:
obtains *b g* **where** *f* = *Abs-finfun* (*map-default b g*) *finite* (*dom g*) *b* ∉ *ran g*
⟨*proof*⟩

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

lift-definition *finfun-const* :: ′*b* ⇒ ′*a* ⇒*f* ′*b* (‹*K*$/ -› [*0*] *1*)
is λ *b x*. *b* ⟨*proof*⟩

lift-definition *finfun-update* :: ′*a* ⇒*f* ′*b* ⇒ ′*a* ⇒ ′*b* ⇒ ′*a* ⇒*f* ′*b* (‹-′(- $:= -′)› [*1000*,*0*,*0*] *1000*) **is** *fun-upd*
⟨*proof*⟩

lemma *finfun-update-twist*: *a* ≠ *a*′ ⟹ *f*(*a* $:= *b*)(*a*′ $:= *b*′) = *f*(*a*′ $:= *b*′)(*a* $:= *b*)
⟨*proof*⟩

lemma *finfun-update-twice* [*simp*]:
$f(a \$:= b)(a \$:= b') = f(a \$:= b')$
⟨*proof*⟩

lemma *finfun-update-const-same*: $(K\$\ b)(a \$:= b) = (K\$\ b)$
⟨*proof*⟩

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*]:
$(K\$\ b)(a \$:= b') = (if\ b = b'\ then\ (K\$\ b)\ else\ finfun\text{-}update\text{-}code\ (K\$\ b)\ a\ b')$
⟨*proof*⟩

lemma *finfun-update-update-code* [*code*]:
$(finfun\text{-}update\text{-}code\ f\ a\ b)(a' \$:= b') = (if\ a = a'\ then\ f(a \$:= b')\ else\ finfun\text{-}update\text{-}code\ (f(a' \$:= b'))\ a\ b)$
⟨*proof*⟩

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*⟩

lemma *fold-finfun-update-finite-univ*:
assumes *fin*: *finite* $(UNIV :: 'a\ set)$
shows $Finite\text{-}Set.fold\ (\lambda a\ f.\ f(a \$:= b'))\ (K\$\ b)\ (UNIV :: 'a\ set) = (K\$\ b')$
⟨*proof*⟩

1.7 Default value for FinFuns

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

lemma *finfun-default-aux-infinite*:
fixes $f :: 'a \Rightarrow\ 'b$
assumes *infin*: $\neg\ finite\ (UNIV :: 'a\ set)$
and *fin*: $finite\ \{a.\ f\ a \neq b\}$

shows *finfun-default-aux f* = *b*
⟨*proof*⟩

lemma *finite-finfun-default-aux*:
fixes *f* :: *'a* ⇒ *'b*
assumes *fin*: *f* ∈ *finfun*
shows *finite* {*a*. *f a* ≠ *finfun-default-aux f*}
⟨*proof*⟩

lemma *finfun-default-aux-update-const*:
fixes *f* :: *'a* ⇒ *'b*
assumes *fin*: *f* ∈ *finfun*
shows *finfun-default-aux* (*f*(*a* := *b*)) = *finfun-default-aux f*
⟨*proof*⟩

lift-definition *finfun-default* :: *'a* ⇒*f* *'b* ⇒ *'b*
is *finfun-default-aux* ⟨*proof*⟩

lemma *finite-finfun-default*: *finite* {*a*. *finfun-apply f a* ≠ *finfun-default f*}
⟨*proof*⟩

lemma *finfun-default-const*: *finfun-default* ((*K$ b*) :: *'a* ⇒*f* *'b*) = (*if finite* (*UNIV* :: *'a set*) *then undefined else b*)
⟨*proof*⟩

lemma *finfun-default-update-const*:
finfun-default (*f*(*a* $:= *b*)) = *finfun-default f*
⟨*proof*⟩

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

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

1.8 Recursion combinator and well-formedness conditions

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

locale *finfun-rec-wf-aux* =

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

lemma *upd-left-comm*: *comp-fun-commute* (λ*a*. *upd a* (*f a*))
⟨*proof*⟩

lemma *upd-upd-twice*: *upd a b''* (*upd a b'* (*cnst b*)) = *upd a b''* (*cnst b*)
⟨*proof*⟩

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* (λ*a*. *upd a* (*map-default d g a*)) (*cnst d*) (*dom f*)) =
Finite-Set.fold (λ*a*. *upd a* (*map-default d g a*)) (*cnst d*) (*dom f*)
⟨*proof*⟩

lemma *map-default-update-twice*:
assumes *fin*: *finite* (*dom f*)
and *anf*: $a \notin$ *dom f*
and *fg*: $f \subseteq_m g$
shows *upd a d''* (*upd a d'* (*Finite-Set.fold* (λ*a*. *upd a* (*map-default d g a*)) (*cnst d*) (*dom f*))) =
upd a d'' (*Finite-Set.fold* (λ*a*. *upd a* (*map-default d g a*)) (*cnst d*) (*dom f*))
⟨*proof*⟩

lemma *map-default-eq-id* [*simp*]: *map-default d* ((λ*a*. *Some* (*f a*)) |‘ {*a*. $f\ a \neq d$}) = *f*
⟨*proof*⟩

lemma *finite-rec-cong1*:
assumes *f*: *comp-fun-commute f* **and** *g*: *comp-fun-commute g*
and *fin*: *finite A*
and *eq*: $\bigwedge a.\ a \in A \Longrightarrow f\ a = g\ a$
shows *Finite-Set.fold f z A* = *Finite-Set.fold g z A*
⟨*proof*⟩

lemma *finfun-rec-upd* [*simp*]:
finfun-rec cnst upd (*f*(*a'* $:= *b'*)) = *upd a' b'* (*finfun-rec cnst upd f*)
including *finfun*
⟨*proof*⟩

end

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

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

end

1.9 Weak induction rule and case analysis for FinFuns

lemma *finfun-weak-induct* [*consumes 0*, *case-names const update*]:
assumes *const*: ⋀*b*. *P* (*K$ b*)
and *update*: ⋀*f a b*. *P f* ⟹ *P* (*f*(*a* $:= *b*))
shows *P x*
including *finfun*
⟨*proof*⟩

lemma *finfun-exhaust-disj*: (∃ *b*. *x* = *finfun-const b*) ∨ (∃ *f a b*. *x* = *finfun-update f a b*)
⟨*proof*⟩

lemma *finfun-exhaust*:
obtains *b* **where** *x* = (*K$ b*)
| *f a b* **where** *x* = *f*(*a* $:= *b*)
⟨*proof*⟩

lemma *finfun-rec-unique*:
fixes *f* :: *'a* ⇒*f* *'b* ⇒ *'c*
assumes *c*: ⋀*c*. *f* (*K$ c*) = *cnst c*
and *u*: ⋀*g a b*. *f* (*g*(*a* $:= *b*)) = *upd g a b* (*f g*)
and *c'*: ⋀*c*. *f'* (*K$ c*) = *cnst c*
and *u'*: ⋀*g a b*. *f'* (*g*(*a* $:= *b*)) = *upd g a b* (*f' g*)
shows *f* = *f'*
⟨*proof*⟩

1.10 Function application

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

interpretation *finfun-apply-aux*: *finfun-rec-wf-aux* λ*b*. *b* λ*a' b c*. *if* (*a* = *a'*) *then b else c*
⟨*proof*⟩

interpretation *finfun-apply*: *finfun-rec-wf* λ*b*. *b* λ*a' b c*. *if* (*a* = *a'*) *then b else c*
⟨*proof*⟩

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

lemma *finfun-upd-apply*: $f(a\ \${:=}\ b)\ \$\ a' = (\textit{if}\ a = a'\ \textit{then}\ b\ \textit{else}\ f\ \$\ a')$
and *finfun-upd-apply-code* [*code*]: $(\textit{finfun-update-code}\ f\ a\ b)\ \$\ a' = (\textit{if}\ a = a'\ \textit{then}\ b\ \textit{else}\ f\ \$\ a')$
⟨*proof*⟩

lemma *finfun-const-apply* [*simp*, *code*]: $(K\$\ b)\ \$\ a = b$
⟨*proof*⟩

lemma *finfun-upd-apply-same* [*simp*]:
$f(a\ \${:=}\ b)\ \$\ a = b$
⟨*proof*⟩

lemma *finfun-upd-apply-other* [*simp*]:
$a \neq a' \Longrightarrow f(a\ \${:=}\ b)\ \$\ a' = f\ \$\ a'$
⟨*proof*⟩

lemma *finfun-ext*: $(\bigwedge a.\ f\ \$\ a = g\ \$\ a) \Longrightarrow f = g$
⟨*proof*⟩

lemma *expand-finfun-eq*: $(f = g) = ((\$)\ f = (\$)\ g)$
⟨*proof*⟩

lemma *finfun-upd-triv* [*simp*]: $f(x\ \${:=}\ f\ \$\ x) = f$
⟨*proof*⟩

lemma *finfun-const-inject* [*simp*]: $(K\$\ b) = (K\$\ b') \equiv b = b'$
⟨*proof*⟩

lemma *finfun-const-eq-update*:
$((K\$\ b) = f(a\ \${:=}\ b')) = (b = b' \wedge (\forall a'.\ a \neq a' \longrightarrow f\ \$\ a' = b))$
⟨*proof*⟩

1.11 Function composition

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

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

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

interpretation *finfun-comp*: *finfun-rec-wf* (λb. (K$ g b)) (λa b c. c(a $:= g b))
⟨*proof*⟩

lemma *finfun-comp-const* [*simp*, *code*]:
g ∘$ (K$ c) = (K$ g c)
⟨*proof*⟩

lemma *finfun-comp-update* [*simp*]: g ∘$ (f(a $:= b)) = (g ∘$ f)(a $:= g b)
and *finfun-comp-update-code* [*code*]:
g ∘$ (*finfun-update-code* f a b) = *finfun-update-code* (g ∘$ f) a (g b)
⟨*proof*⟩

lemma *finfun-comp-apply* [*simp*]:
($) (g ∘$ f) = g ∘ ($) f
⟨*proof*⟩

lemma *finfun-comp-comp-collapse* [*simp*]: f ∘$ g ∘$ h = (f ∘ g) ∘$ h
⟨*proof*⟩

lemma *finfun-comp-const1* [*simp*]: (λx. c) ∘$ f = (K$ c)
⟨*proof*⟩

lemma *finfun-comp-id1* [*simp*]: (λx. x) ∘$ f = f *id* ∘$ f = f
⟨*proof*⟩

lemma *finfun-comp-conv-comp*: g ∘$ f = *Abs-finfun* (g ∘ ($) f)
including *finfun*
⟨*proof*⟩

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

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

lemma *finfun-comp2-const* [*code*, *simp*]: *finfun-comp2* (K$ c) f = (K$ c)
including *finfun*
⟨*proof*⟩

lemma *finfun-comp2-update*:
assumes *inj*: *inj f*
shows *finfun-comp2* (g(b $:= c)) f = (*if* b ∈ *range f then* (*finfun-comp2 g f*)(*inv f* b $:= c) *else finfun-comp2 g f*)
including *finfun*
⟨*proof*⟩

1.12 Universal quantification

definition *finfun-All-except* :: 'a *list* ⇒ 'a ⇒f *bool* ⇒ *bool*

where [*code del*]: *finfun-All-except A P* ≡ ∀ *a*. *a* ∈ *set A* ∨ *P* $ *a*

lemma *finfun-All-except-const*: *finfun-All-except A* (*K*$ *b*) ⟷ *b* ∨ *set A* = *UNIV*
⟨*proof*⟩

lemma *finfun-All-except-const-finfun-UNIV-code* [*code*]:
finfun-All-except A (*K*$ *b*) = (*b* ∨ *is-list-UNIV A*)
⟨*proof*⟩

lemma *finfun-All-except-update*:
finfun-All-except A f(*a* $:= *b*) = ((*a* ∈ *set A* ∨ *b*) ∧ *finfun-All-except* (*a* # *A*) *f*)
⟨*proof*⟩

lemma *finfun-All-except-update-code* [*code*]:
fixes *a* :: ′*a* :: *card-UNIV*
shows *finfun-All-except A* (*finfun-update-code f a b*) = ((*a* ∈ *set A* ∨ *b*) ∧ *finfun-All-except* (*a* # *A*) *f*)
⟨*proof*⟩

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

lemma *finfun-All-const* [*simp*]: *finfun-All* (*K*$ *b*) = *b*
⟨*proof*⟩

lemma *finfun-All-update*: *finfun-All f*(*a* $:= *b*) = (*b* ∧ *finfun-All-except* [*a*] *f*)
⟨*proof*⟩

lemma *finfun-All-All*: *finfun-All P* = *All* (($) *P*)
⟨*proof*⟩

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

lemma *finfun-Ex-Ex*: *finfun-Ex P* = *Ex* (($) *P*)
⟨*proof*⟩

lemma *finfun-Ex-const* [*simp*]: *finfun-Ex* (*K*$ *b*) = *b*
⟨*proof*⟩

1.13 A diagonal operator for FinFuns

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

interpretation *finfun-Diag-aux*: *finfun-rec-wf-aux* λ*b*. *Pair b* ∘$ *g* λ*a b c*. *c*(*a* $:=

$(b,\ g\ \$\ a))$
⟨*proof*⟩

interpretation *finfun-Diag*: *finfun-rec-wf* $\lambda b.\ Pair\ b \circ\$\ g\ \lambda a\ b\ c.\ c(a\ \$:= (b,\ g\ \$\ a))$
⟨*proof*⟩

lemma *finfun-Diag-const1*: $(\$K\$\ b,\ g\$) = Pair\ b \circ\$\ g$
⟨*proof*⟩

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

lemma *finfun-Diag-const-code* [*code*]:
$(\$K\$\ b,\ K\$\ c\$) = (K\$\ (b,\ c))$
$(\$K\$\ b,\ finfun\text{-}update\text{-}code\ g\ a\ c\$) = finfun\text{-}update\text{-}code\ (\$K\$\ b,\ g\$)\ a\ (b,\ c)$
⟨*proof*⟩

lemma *finfun-Diag-update1*: $(\$f(a\ \$:= b),\ g\$) = (\$f,\ g\$)(a\ \$:= (b,\ g\ \$\ a))$
and *finfun-Diag-update1-code* [*code*]: $(\$finfun\text{-}update\text{-}code\ f\ a\ b,\ g\$) = (\$f,\ g\$)(a\ \$:= (b,\ g\ \$\ a))$
⟨*proof*⟩

lemma *finfun-Diag-const2*: $(\$f,\ K\$\ c\$) = (\lambda b.\ (b,\ c)) \circ\$\ f$
⟨*proof*⟩

lemma *finfun-Diag-update2*: $(\$f,\ g(a\ \$:= c)\$) = (\$f,\ g\$)(a\ \$:= (f\ \$\ a,\ c))$
⟨*proof*⟩

lemma *finfun-Diag-const-const* [*simp*]: $(\$K\$\ b,\ K\$\ c\$) = (K\$\ (b,\ c))$
⟨*proof*⟩

lemma *finfun-Diag-const-update*:
$(\$K\$\ b,\ g(a\ \$:= c)\$) = (\$K\$\ b,\ g\$)(a\ \$:= (b,\ c))$
⟨*proof*⟩

lemma *finfun-Diag-update-const*:
$(\$f(a\ \$:= b),\ K\$\ c\$) = (\$f,\ K\$\ c\$)(a\ \$:= (b,\ c))$
⟨*proof*⟩

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

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

lemma *finfun-Diag-conv-Abs-finfun*:

($\$f,\ g\$) = Abs\text{-}finfun\ ((\lambda x.\ (f\ \$\ x,\ g\ \$\ x)))$
  including *finfun*
⟨*proof*⟩

lemma *finfun-Diag-eq*: $(\$f,\ g\$) = (\$f',\ g'\$) \longleftrightarrow f = f' \wedge g = g'$
⟨*proof*⟩

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

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

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

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

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

lemma *finfun-fst-conv-Abs-finfun*: $finfun\text{-}fst = (\lambda f.\ Abs\text{-}finfun\ (fst \circ (\$)\ f))$
⟨*proof*⟩

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

lemma *finfun-snd-const*: $finfun\text{-}snd\ (K\$\ bc) = (K\$\ snd\ bc)$
⟨*proof*⟩

lemma *finfun-snd-update*: $finfun\text{-}snd\ (f(a\ \$:=\ bc)) = (finfun\text{-}snd\ f)(a\ \$:=\ snd\ bc)$
  and *finfun-snd-update-code* [*code*]: $finfun\text{-}snd\ (finfun\text{-}update\text{-}code\ f\ a\ bc) = (finfun\text{-}snd\ f)(a\ \$:=\ snd\ bc)$
⟨*proof*⟩

lemma *finfun-snd-comp-conv*: $finfun\text{-}snd\ (f \circ\$\ g) = (snd \circ f) \circ\$\ g$
⟨*proof*⟩

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

lemma *finfun-snd-conv-Abs-finfun*: $finfun\text{-}snd = (\lambda f.\ Abs\text{-}finfun\ (snd \circ (\$)\ f))$
⟨*proof*⟩

lemma *finfun-Diag-collapse* [*simp*]: $(\$finfun\text{-}fst\ f,\ finfun\text{-}snd\ f\$) = f$

⟨*proof*⟩

1.14 Currying for FinFuns

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

interpretation *finfun-curry-aux*: *finfun-rec-wf-aux finfun-const* ∘ *finfun-const* λ(*a*, *b*) *c f*. *f*(*a* $:= (*f* $ *a*)(*b* $:= *c*))
⟨*proof*⟩

interpretation *finfun-curry*: *finfun-rec-wf finfun-const* ∘ *finfun-const* λ(*a*, *b*) *c f*. *f*(*a* $:= (*f* $ *a*)(*b* $:= *c*))
⟨*proof*⟩

lemma *finfun-curry-const* [*simp*, *code*]: *finfun-curry* (*K*$ *c*) = (*K*$ *K*$ *c*)
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *finfun-Abs-finfun-curry*: **assumes** *fin*: *f* ∈ *finfun*
shows (λ*a*. *Abs-finfun* (*curry f a*)) ∈ *finfun*
including *finfun*
⟨*proof*⟩

lemma *finfun-curry-conv-curry*:
fixes *f* :: ('*a* × '*b*) ⇒*f* '*c*
shows *finfun-curry f* = *Abs-finfun* (λ*a*. *Abs-finfun* (*curry* (*finfun-apply f*) *a*))
including *finfun*
⟨*proof*⟩

1.15 Executable equality for FinFuns

lemma *eq-finfun-All-ext*: (*f* = *g*) ⟷ *finfun-All* ((λ(*x*, *y*). *x* = *y*) ∘$ ($*f*, *g*$))
⟨*proof*⟩

instantiation *finfun* :: ({*card-UNIV*,*equal*},*equal*) *equal* **begin**
definition *eq-finfun-def* [*code*]: *HOL.equal f g* ⟷ *finfun-All* ((λ(*x*, *y*). *x* = *y*) ∘$ ($*f*, *g*$))
instance ⟨*proof*⟩
end

lemma [*code nbe*]:

HOL.equal (*f* :: - ⇒*f* -) *f* ⟷ *True*
⟨*proof*⟩

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

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

lemma *finfun-clearjunk-const* [*code*]: *finfun-clearjunk* (*K$ b*) = (*K$ b*)
⟨*proof*⟩

lemma *finfun-clearjunk-update* [*code*]:
finfun-clearjunk (*finfun-update-code f a b*) = *f*(*a* $:= *b*)
⟨*proof*⟩

1.17 The domain of a FinFun as a FinFun

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

lemma *finfun-dom-const*:
finfun-dom ((*K$ c*) :: *'a* ⇒*f* *'b*) = (*K$ finite* (*UNIV* :: *'a set*) ∧ *c* ≠ *undefined*)
⟨*proof*⟩

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

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

lemma *finfun-dom-finfunI*: (λ*a*. *f* $ *a* ≠ *finfun-default f*) ∈ *finfun*
⟨*proof*⟩

lemma *finfun-dom-update* [*simp*]:
finfun-dom (*f*(*a* $:= *b*)) = (*finfun-dom f*)(*a* $:= (*b* ≠ *finfun-default f*))
including *finfun* ⟨*proof*⟩

lemma *finfun-dom-update-code* [*code*]:
finfun-dom (*finfun-update-code f a b*) = *finfun-update-code* (*finfun-dom f*) *a* (*b* ≠ *finfun-default f*)
⟨*proof*⟩

lemma *finite-finfun-dom*: *finite* {*x*. *finfun-dom f* $ *x*}
⟨*proof*⟩

1.18 The domain of a FinFun as a sorted list

definition *finfun-to-list* :: (*'a* :: *linorder*) ⇒*f* *'b* ⇒ *'a list*
where
finfun-to-list f = (*THE xs. set xs* = {*x. finfun-dom f* $ *x*} ∧ *sorted xs* ∧ *distinct xs*)

lemma *set-finfun-to-list* [*simp*]: *set* (*finfun-to-list f*) = {*x. finfun-dom f* $ *x*} (**is** *?thesis1*)
and *sorted-finfun-to-list*: *sorted* (*finfun-to-list f*) (**is** *?thesis2*)
and *distinct-finfun-to-list*: *distinct* (*finfun-to-list f*) (**is** *?thesis3*)
⟨*proof*⟩

lemma *finfun-const-False-conv-bot*: ($) (*K*$ *False*) = *bot*
⟨*proof*⟩

lemma *finfun-const-True-conv-top*: ($) (*K*$ *True*) = *top*
⟨*proof*⟩

lemma *finfun-to-list-const*:
finfun-to-list ((*K*$ *c*) :: (*'a* :: {*linorder*} ⇒*f* *'b*)) =
(*if* ¬ *finite* (*UNIV* :: *'a set*) ∨ *c* = *undefined then* [] *else THE xs. set xs* = *UNIV* ∧ *sorted xs* ∧ *distinct xs*)
⟨*proof*⟩

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

lemma *remove1-insort-insert-same*:
x ∉ *set xs* ⟹ *remove1 x* (*insort-insert x xs*) = *xs*
⟨*proof*⟩

lemma *finfun-dom-conv*:
finfun-dom f $ *x* ⟷ *f* $ *x* ≠ *finfun-default f*
⟨*proof*⟩

lemma *finfun-to-list-update*:
finfun-to-list (*f*(*a* $:= *b*)) =
(*if b* = *finfun-default f then List.remove1 a* (*finfun-to-list f*) *else List.insort-insert a* (*finfun-to-list f*))
⟨*proof*⟩

lemma *finfun-to-list-update-code* [*code*]:
finfun-to-list (*finfun-update-code f a b*) =
(*if b* = *finfun-default f then List.remove1 a* (*finfun-to-list f*) *else List.insort-insert a* (*finfun-to-list f*))
⟨*proof*⟩

More type class instantiations

lemma *card-eq-1-iff*: *card A = 1* ⟷ *A* ≠ {} ∧ (∀ *x*∈*A*. ∀ *y*∈*A*. *x* = *y*)
  (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *card-UNIV-finfun*:
  defines *F* == *finfun* :: (′*a* ⇒ ′*b*) *set*
  shows *CARD*(′*a* ⇒*f* ′*b*) = (*if CARD*(′*a*) ≠ *0* ∧ *CARD*(′*b*) ≠ *0* ∨ *CARD*(′*b*) = *1 then CARD*(′*b*) ^ *CARD*(′*a*) *else 0*)
⟨*proof*⟩

lemma *finite-UNIV-finfun*:
  finite (*UNIV* :: (′*a* ⇒*f* ′*b*) *set*) ⟷
  (*finite* (*UNIV* :: ′*a set*) ∧ *finite* (*UNIV* :: ′*b set*) ∨ *CARD*(′*b*) = *1*)
  (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

instantiation *finfun* :: (*finite-UNIV*, *card-UNIV*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(′*a* ⇒*f* ′*b*)
  (*let cb* = *of-phantom* (*card-UNIV* :: ′*b card-UNIV*)
   in cb = *1* ∨ *of-phantom* (*finite-UNIV* :: ′*a finite-UNIV*) ∧ *cb* ≠ *0*)
instance
  ⟨*proof*⟩
end

instantiation *finfun* :: (*card-UNIV*, *card-UNIV*) *card-UNIV* **begin**
definition *card-UNIV* = *Phantom*(′*a* ⇒*f* ′*b*)
  (*let ca* = *of-phantom* (*card-UNIV* :: ′*a card-UNIV*);
       cb = *of-phantom* (*card-UNIV* :: ′*b card-UNIV*)
   in if ca ≠ *0* ∧ *cb* ≠ *0* ∨ *cb* = *1 then cb* ^ *ca else 0*)
instance ⟨*proof*⟩
end

1.18.1 Bundles for concrete syntax

bundle *finfun-syntax*
begin

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

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

notation (*ASCII*)

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

end

unbundle *no finfun-syntax*

end

2 Predicates modelled as FinFuns

theory *FinFunPred*
imports *FinFun*
begin

unbundle *finfun-syntax*

Instantiate FinFun predicates just like predicates

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

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

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

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

instance ⟨*proof*⟩

lemma *le-finfun-code* [*code*]:
$f \leq g \longleftrightarrow$ *finfun-All* $((\lambda(x,\ y).\ x \leq y) \circ\$\ (\$f,\ g\$))$
⟨*proof*⟩

end

instance *finfun* :: (*type*, *preorder*) *preorder*
⟨*proof*⟩

instance *finfun* :: (*type*, *order*) *order*
⟨*proof*⟩

instantiation *finfun* :: (*type*, *order-bot*) *order-bot* **begin**
definition *bot* = *finfun-const bot*
instance ⟨*proof*⟩
end

lemma *bot-finfun-apply* [*simp*]: ($) *bot* = (λ-. *bot*)
⟨*proof*⟩

instantiation *finfun* :: (*type*, *order-top*) *order-top* **begin**
definition *top* = *finfun-const top*
instance ⟨*proof*⟩
end

lemma *top-finfun-apply* [*simp*]: ($) *top* = (λ-. *top*)
⟨*proof*⟩

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

lemma *inf-finfun-apply* [*simp*]: ($) (*inf f g*) = *inf* (($) *f*) (($) *g*)
⟨*proof*⟩

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

lemma *sup-finfun-apply* [*simp*]: ($) (*sup f g*) = *sup* (($) *f*) (($) *g*)
⟨*proof*⟩

instance *finfun* :: (*type*, *semilattice-inf*) *semilattice-inf*
⟨*proof*⟩

instance *finfun* :: (*type*, *semilattice-sup*) *semilattice-sup*
⟨*proof*⟩

instance *finfun* :: (*type*, *lattice*) *lattice* ⟨*proof*⟩

instance *finfun* :: (*type*, *bounded-lattice*) *bounded-lattice*
⟨*proof*⟩

instance *finfun* :: (*type*, *distrib-lattice*) *distrib-lattice*
⟨*proof*⟩

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

lemma *minus-finfun-apply* [*simp*]: ($) (*f* − *g*) = ($) *f* − ($) *g*
⟨*proof*⟩

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

end

lemma *uminus-finfun-apply* [*simp*]: ($) (− *g*) = − ($) *g*
⟨*proof*⟩

instance *finfun* :: (*type*, *boolean-algebra*) *boolean-algebra*
⟨*proof*⟩

Replicate predicate operations for FinFuns

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

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

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

lemma *finfun-single-apply* [*simp*]:
finfun-single x $ *y* ⟷ *x* = *y*
⟨*proof*⟩

lemma [*iff*]:
shows *finfun-single-neq-bot*: *finfun-single x* ≠ *bot*
and *bot-neq-finfun-single*: *bot* ≠ *finfun-single x*
⟨*proof*⟩

lemma *finfun-leI* [*intro*!]: (!!*x*. *A* $ *x* ⟹ *B* $ *x*) ⟹ *A* ≤ *B*
⟨*proof*⟩

lemma *finfun-leD* [*elim*]: ⟦ *A* ≤ *B*; *A* $ *x* ⟧ ⟹ *B* $ *x*
⟨*proof*⟩

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

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

lemma *finfun-Ball-except-const*:
finfun-Ball-except xs (*K*$ *b*) *P* ⟷ ¬ *b* ∨ *set xs* = *UNIV* ∨ *Code.abort* (*STR* ′′*finfun-ball-except*′′) (λ*u*. *finfun-Ball-except xs* (*K*$ *b*) *P*)
⟨*proof*⟩

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

lemma *finfun-Ball-except-update*:

finfun-Ball-except xs (*A*(*a* \$:= *b*)) *P* = ((*a* ∈ *set xs* ∨ (*b* ⟶ *P a*)) ∧ *finfun-Ball-except* (*a* # *xs*) *A P*)
⟨*proof*⟩

lemma *finfun-Ball-except-update-code* [*code*]:
fixes *a* :: ′*a* :: *card-UNIV*
shows *finfun-Ball-except xs* (*finfun-update-code f a b*) *P* = ((*a* ∈ *set xs* ∨ (*b* ⟶ *P a*)) ∧ *finfun-Ball-except* (*a* # *xs*) *f P*)
⟨*proof*⟩

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

lemma *finfun-Ball-code* [*code*]: *finfun-Ball* = *finfun-Ball-except* []
⟨*proof*⟩

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

lemma *finfun-Bex-except-const*:
finfun-Bex-except xs (*K*\$ *b*) *P* ⟷ *b* ∧ *set xs* ≠ *UNIV* ∧ *Code.abort* (*STR* ′′*finfun-Bex-except*′′) (λ*u*. *finfun-Bex-except xs* (*K*\$ *b*) *P*)
⟨*proof*⟩

lemma *finfun-Bex-except-const-finfun-UNIV-code* [*code*]:
finfun-Bex-except xs (*K*\$ *b*) *P* ⟷ *b* ∧ ¬ *is-list-UNIV xs* ∧ *Code.abort* (*STR* ′′*finfun-Bex-except*′′) (λ*u*. *finfun-Bex-except xs* (*K*\$ *b*) *P*)
⟨*proof*⟩

lemma *finfun-Bex-except-update*:
finfun-Bex-except xs (*A*(*a* \$:= *b*)) *P* ⟷ (*a* ∉ *set xs* ∧ *b* ∧ *P a*) ∨ *finfun-Bex-except* (*a* # *xs*) *A P*
⟨*proof*⟩

lemma *finfun-Bex-except-update-code* [*code*]:
fixes *a* :: ′*a* :: *card-UNIV*
shows *finfun-Bex-except xs* (*finfun-update-code f a b*) *P* ⟷ ((*a* ∉ *set xs* ∧ *b* ∧ *P a*) ∨ *finfun-Bex-except* (*a* # *xs*) *f P*)
⟨*proof*⟩

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

lemma *finfun-Bex-code* [*code*]: *finfun-Bex* = *finfun-Bex-except* []
⟨*proof*⟩

Automatically replace predicate operations by finfun predicate operations where possible

lemma *iso-finfun-le* [*code-unfold*]:
($) $A \leq$ ($) $B \longleftrightarrow A \leq B$
⟨*proof*⟩

lemma *iso-finfun-less* [*code-unfold*]:
($) $A <$ ($) $B \longleftrightarrow A < B$
⟨*proof*⟩

lemma *iso-finfun-eq* [*code-unfold*]:
($) $A =$ ($) $B \longleftrightarrow A = B$
⟨*proof*⟩

lemma *iso-finfun-sup* [*code-unfold*]:
sup (($) A) (($) B) = ($) (*sup* A B)
⟨*proof*⟩

lemma *iso-finfun-disj* [*code-unfold*]:
A $ $x \vee B$ $ $x \longleftrightarrow$ *sup* A B $ x
⟨*proof*⟩

lemma *iso-finfun-inf* [*code-unfold*]:
inf (($) A) (($) B) = ($) (*inf* A B)
⟨*proof*⟩

lemma *iso-finfun-conj* [*code-unfold*]:
A $ $x \wedge B$ $ $x \longleftrightarrow$ *inf* A B $ x
⟨*proof*⟩

lemma *iso-finfun-empty-conv* [*code-unfold*]:
(λ-. *False*) = ($) $\{\}_f$
⟨*proof*⟩

lemma *iso-finfun-UNIV-conv* [*code-unfold*]:
(λ-. *True*) = ($) *finfun-UNIV*
⟨*proof*⟩

lemma *iso-finfun-upd* [*code-unfold*]:
fixes A :: *'a* $pred_f$
shows (($) A)($x := b$) = ($) (A(x $:= b))
⟨*proof*⟩

lemma *iso-finfun-uminus* [*code-unfold*]:
fixes A :: *'a* $pred_f$
shows $-$ ($) A = ($) ($- A$)
⟨*proof*⟩

lemma *iso-finfun-minus* [*code-unfold*]:
fixes A :: *'a* $pred_f$
shows ($) $A -$ ($) B = ($) ($A - B$)

⟨*proof*⟩

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

lemma *iso-finfun-Ball-Ball*:
$(\forall x.\ A\ \$\ x \longrightarrow P\ x) \longleftrightarrow$ *finfun-Ball A P*
⟨*proof*⟩

lemma *iso-finfun-Bex-Bex*:
$(\exists x.\ A\ \$\ x \wedge P\ x) \longleftrightarrow$ *finfun-Bex A P*
⟨*proof*⟩

Test code setup

notepad begin
⟨*proof*⟩
end

declare *iso-finfun-Ball-Ball*[*code-unfold*]
notepad begin
⟨*proof*⟩
end
declare *iso-finfun-Ball-Ball*[*code-unfold del*]

end