

Finfun

Andreas Lochbihler

February 23, 2021

Contents

1	Almost everywhere constant functions	1
1.1	The <i>map-default</i> operation	2
1.2	The finfun type	2
1.3	Kernel functions for type $'a \Rightarrow f 'b$	4
1.4	Code generator setup	5
1.5	Setup for quickcheck	5
1.6	<i>finfun-update</i> as instance of <i>comp-fun-commute</i>	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	8
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	14
1.15	Executable equality for FinFuns	14
1.16	An operator that explicitly removes all redundant updates in the generated representations	15
1.17	The domain of a FinFun as a FinFun	15
1.18	The domain of a FinFun as a sorted list	16
1.18.1	Bundles for concrete syntax	17
2	Predicates modelled as FinFuns	18

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 finfun along with a number of operators for them. The code generator is set up such that such functions can be represented as data in the generated code and all operators are executable. For details, see Formalising FinFuns - Generating Code for Functions as Data by A. Lochbihler in TPHOLs 2009.

1.1 The *map-default* operation

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

lemma *map-default-delete* [simp]:
map-default b (f(a := None)) = (*map-default* b f)(a := b)
 <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 \rightarrow 'b
 assumes *infin-eq*: \neg finite (UNIV :: 'a set) \vee b = b'
 and *fin*: finite (dom g) and b: b \notin ran g
 and *fin'*: finite (dom g') and b': b' \notin ran g'
 and *eq'*: *map-default* b g = *map-default* b' g'
 shows b = b' g = g'
 <proof>

1.2 The finfun type

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

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

type-notation *finfun* ((- \Rightarrow 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 $y \in \text{finfun}$

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

<proof>

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

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

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

<proof>

lemma *map-of-finfun*: $\text{map-of } xs \in \text{finfun}$

<proof>

lemma *Diag-finfun*: $(\lambda x. (f x, g x)) \in \text{finfun} \iff f \in \text{finfun} \wedge g \in \text{finfun}$

<proof>

lemma *finfun-right-compose*:

assumes $g: g \in \text{finfun}$ **and** *inj*: $\text{inj } f$

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

<proof>

lemma *finfun-curry*:

assumes *fin*: $f \in \text{finfun}$

shows *curry f* $\in \text{finfun}$ *curry f a* $\in \text{finfun}$

<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* $x = \text{Abs-finfun } y \iff x = y$

<proof>

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

fixes $x\ y :: ('a :: \text{finite}) \Rightarrow 'b$
shows $\text{Abs-funfun } x = \text{Abs-funfun } y \longleftrightarrow x = y$
 $\langle \text{proof} \rangle$

lemma *Abs-funfun-inj-finite*:
assumes $\text{fin}: \text{finite } (\text{UNIV} :: 'a \text{ set})$
shows $\text{inj } (\text{Abs-funfun} :: ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow f 'b)$
 $\langle \text{proof} \rangle$

lemma *Abs-funfun-inverse-finite*:
fixes $x :: 'a \Rightarrow 'b$
assumes $\text{fin}: \text{finite } (\text{UNIV} :: 'a \text{ set})$
shows $\text{funfun-apply } (\text{Abs-funfun } x) = x$
including funfun
 $\langle \text{proof} \rangle$

lemma *Abs-funfun-inverse-finite-class*:
fixes $x :: ('a :: \text{finite}) \Rightarrow 'b$
shows $\text{funfun-apply } (\text{Abs-funfun } x) = x$
 $\langle \text{proof} \rangle$

lemma *funfun-eq-finite-UNIV*: $\text{finite } (\text{UNIV} :: 'a \text{ set}) \Longrightarrow (\text{funfun} :: ('a \Rightarrow 'b) \text{ set}) = \text{UNIV}$
 $\langle \text{proof} \rangle$

lemma *funfun-finite-UNIV-class*: $\text{funfun} = (\text{UNIV} :: ('a :: \text{finite}) \Rightarrow 'b) \text{ set}$
 $\langle \text{proof} \rangle$

lemma *map-default-in-funfun*:
assumes $\text{fin}: \text{finite } (\text{dom } f)$
shows $\text{map-default } b\ f \in \text{funfun}$
 $\langle \text{proof} \rangle$

lemma *funfun-cases-map-default*:
obtains $b\ g$ **where** $f = \text{Abs-funfun } (\text{map-default } b\ g)$ $\text{finite } (\text{dom } g)$ $b \notin \text{ran } g$
 $\langle \text{proof} \rangle$

1.3 Kernel functions for type $'a \Rightarrow f 'b$

lift-definition $\text{funfun-const} :: 'b \Rightarrow 'a \Rightarrow f 'b$ ($K\$/ - [0] 1$)
is $\lambda\ b\ x. b$ $\langle \text{proof} \rangle$

lift-definition $\text{funfun-update} :: 'a \Rightarrow f 'b \Rightarrow 'a \Rightarrow 'b \Rightarrow 'a \Rightarrow f 'b$ ($-'(- \$:= -')$)
 $[1000,0,0] 1000$) **is** fun-upd
 $\langle \text{proof} \rangle$

lemma *funfun-update-twist*: $a \neq a' \Longrightarrow f(a\ \$:= b)(a'\ \$:= b') = f(a'\ \$:= b')(a\ \$:= b)$
 $\langle \text{proof} \rangle$

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

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

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

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

1.7 Default value for FinFuns

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

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

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

lemma *finite-finfun-default-aux*:
fixes $f :: 'a \Rightarrow 'b$
assumes $fin: f \in \text{finfun}$
shows *finite* $\{a. f\ a \neq \text{finfun-default-aux}\ f\}$
 ⟨*proof*⟩

lemma *finfun-default-aux-update-const*:
fixes $f :: 'a \Rightarrow 'b$
assumes $fin: f \in \text{finfun}$
shows $\text{finfun-default-aux}\ (f(a := b)) = \text{finfun-default-aux}\ f$
 ⟨*proof*⟩

lift-definition *finfun-default* $:: 'a \Rightarrow_f 'b \Rightarrow 'b$
is *finfun-default-aux* ⟨*proof*⟩

lemma *finite-finfun-default*: *finite* $\{a. \text{finfun-apply}\ f\ a \neq \text{finfun-default}\ f\}$
 ⟨*proof*⟩

lemma *finfun-default-const*: $\text{finfun-default}\ ((K\$ b) :: 'a \Rightarrow_f 'b) = (\text{if}\ \text{finite}\ (\text{UNIV} :: 'a\ \text{set})\ \text{then}\ \text{undefined}\ \text{else}\ b)$
 ⟨*proof*⟩

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

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

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

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*]:

$\text{finfun-rec}\ \text{cst}\ \text{upd}\ f \equiv$
 $\text{let}\ b = \text{finfun-default}\ f;$
 $g = \text{THE}\ g. f = \text{Abs-finfun}\ (\text{map-default}\ b\ g) \wedge \text{finite}\ (\text{dom}\ g) \wedge b \notin \text{ran}\ g$
 $\text{in}\ \text{Finite-Set.fold}\ (\lambda a. \text{upd}\ a\ (\text{map-default}\ b\ g\ a))\ (\text{cst}\ b)\ (\text{dom}\ g)$

locale *finfun-rec-wf-aux* =

fixes $cnst :: 'b \Rightarrow 'c$
and $upd :: 'a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'c$
assumes $upd\text{-}const\text{-}same: upd\ a\ b\ (cnst\ b) = cnst\ b$
and $upd\text{-}commute: a \neq a' \implies upd\ a\ b\ (upd\ a'\ b'\ c) = upd\ a'\ b'\ (upd\ a\ b\ c)$
and $upd\text{-}idemp: b \neq b' \implies upd\ a\ b''\ (upd\ a\ b'\ (cnst\ b)) = upd\ a\ b''\ (cnst\ b)$
begin

lemma $upd\text{-}left\text{-}comm: comp\text{-}fun\text{-}commute\ (\lambda a. upd\ a\ (f\ a))$
 $\langle proof \rangle$

lemma $upd\text{-}upd\text{-}twice: upd\ a\ b''\ (upd\ a\ b'\ (cnst\ b)) = upd\ a\ b''\ (cnst\ b)$
 $\langle proof \rangle$

lemma $map\text{-}default\text{-}update\text{-}const:$
assumes $fin: finite\ (dom\ f)$
and $anf: a \notin dom\ f$
and $fg: f \subseteq_m g$
shows $upd\ a\ d\ (Finite\text{-}Set.fold\ (\lambda a. upd\ a\ (map\text{-}default\ d\ g\ a))\ (cnst\ d)\ (dom\ f)) =$
 $Finite\text{-}Set.fold\ (\lambda a. upd\ a\ (map\text{-}default\ d\ g\ a))\ (cnst\ d)\ (dom\ f)$
 $\langle proof \rangle$

lemma $map\text{-}default\text{-}update\text{-}twice:$
assumes $fin: finite\ (dom\ f)$
and $anf: a \notin dom\ f$
and $fg: f \subseteq_m g$
shows $upd\ a\ d''\ (upd\ a\ d'\ (Finite\text{-}Set.fold\ (\lambda a. upd\ a\ (map\text{-}default\ d\ g\ a))\ (cnst\ d)\ (dom\ f))) =$
 $upd\ a\ d''\ (Finite\text{-}Set.fold\ (\lambda a. upd\ a\ (map\text{-}default\ d\ g\ a))\ (cnst\ d)\ (dom\ f))$
 $\langle proof \rangle$

lemma $map\text{-}default\text{-}eq\text{-}id\ [simp]: map\text{-}default\ d\ ((\lambda a. Some\ (f\ a)) \mid' \{a. f\ a \neq d\})$
 $= f$
 $\langle proof \rangle$

lemma $finite\text{-}rec\text{-}cong1:$
assumes $f: comp\text{-}fun\text{-}commute\ f$ **and** $g: comp\text{-}fun\text{-}commute\ g$
and $fin: finite\ A$
and $eq: \bigwedge a. a \in A \implies f\ a = g\ a$
shows $Finite\text{-}Set.fold\ f\ z\ A = Finite\text{-}Set.fold\ g\ z\ A$
 $\langle proof \rangle$

lemma $finfun\text{-}rec\text{-}upd\ [simp]:$
 $finfun\text{-}rec\ cnst\ upd\ (f(a' \$:= b')) = upd\ a'\ b'\ (finfun\text{-}rec\ cnst\ upd\ f)$
including $finfun$
 $\langle proof \rangle$

end

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

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

end

1.9 Weak induction rule and case analysis for FinFuns

lemma *finfun-weak-induct* [*consumes 0, case-names const update*]:
assumes *const*: $\bigwedge b. P \ (K \$ b)$
and *update*: $\bigwedge f \ a \ b. P \ f \implies P \ (f(a \ \$:= b))$
shows $P \ x$
including *finfun*
 \langle *proof* \rangle

lemma *finfun-exhaust-disj*: $(\exists b. x = \text{finfun-const } b) \vee (\exists f \ a \ b. x = \text{finfun-update } f \ a \ b)$
 \langle *proof* \rangle

lemma *finfun-exhaust*:
obtains *b* **where** $x = (K \$ b)$
| *f* *a* *b* **where** $x = f(a \ \$:= b)$
 \langle *proof* \rangle

lemma *finfun-rec-unique*:
fixes $f :: 'a \Rightarrow f \ 'b \Rightarrow 'c$
assumes $c: \bigwedge c. f \ (K \$ c) = \text{cnst } c$
and $u: \bigwedge g \ a \ b. f \ (g(a \ \$:= b)) = \text{upd } g \ a \ b \ (f \ g)$
and $c': \bigwedge c. f' \ (K \$ c) = \text{cnst } c$
and $u': \bigwedge g \ a \ b. f' \ (g(a \ \$:= b)) = \text{upd } g \ a \ b \ (f' \ g)$
shows $f = f'$
 \langle *proof* \rangle

1.10 Function application

notation *finfun-apply* (**infixl** \$ 999)

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

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

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

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

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

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

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

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

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

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

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

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

1.11 Function composition

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

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

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

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

lemma *finfun-comp-const* [*simp*, *code*]:
 $g \circ\$ (K\$ c) = (K\$ g c)$
 $\langle proof \rangle$

lemma *finfun-comp-update* [*simp*]: $g \circ\$ (f(a \$:= b)) = (g \circ\$ f)(a \$:= g b)$
and *finfun-comp-update-code* [*code*]:
 $g \circ\$ (\text{finfun-update-code } f a b) = \text{finfun-update-code } (g \circ\$ f) a (g b)$
 $\langle proof \rangle$

lemma *finfun-comp-apply* [*simp*]:
 $(\$) (g \circ\$ f) = g \circ (\$) f$
 $\langle proof \rangle$

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

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

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

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

definition *finfun-comp2* :: $'b \Rightarrow f 'c \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow f 'c$ (**infixr** $\$ \circ 55$)
where [*code del*]: $g \$ \circ f = \text{Abs-finfun } ((\$) g \circ f)$

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

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

lemma *finfun-comp2-update*:
assumes *inj*: *inj* f
shows *finfun-comp2* ($g(b \$:= c)$) $f = (\text{if } b \in \text{range } f \text{ then } (\text{finfun-comp2 } g f)(\text{inv } f b \$:= c) \text{ else } \text{finfun-comp2 } g f)$
including *finfun*
 $\langle proof \rangle$

1.12 Universal quantification

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

where $[code\ del]: finfun-All-except\ A\ P \equiv \forall a. a \in set\ A \vee P\ \$\ a$

lemma $finfun-All-except-const: finfun-All-except\ A\ (K\ \$\ b) \longleftrightarrow b \vee set\ A = UNIV$
 $\langle proof \rangle$

lemma $finfun-All-except-const-finfun-UNIV-code\ [code]:$
 $finfun-All-except\ A\ (K\ \$\ b) = (b \vee is-list-UNIV\ A)$
 $\langle proof \rangle$

lemma $finfun-All-except-update:$
 $finfun-All-except\ A\ f(a\ \$:=\ b) = ((a \in set\ A \vee b) \wedge finfun-All-except\ (a\ \#\ A)\ f)$
 $\langle proof \rangle$

lemma $finfun-All-except-update-code\ [code]:$
fixes $a :: 'a :: card-UNIV$
shows $finfun-All-except\ A\ (finfun-update-code\ f\ a\ b) = ((a \in set\ A \vee b) \wedge finfun-All-except\ (a\ \#\ A)\ f)$
 $\langle proof \rangle$

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

lemma $finfun-All-const\ [simp]: finfun-All\ (K\ \$\ b) = b$
 $\langle proof \rangle$

lemma $finfun-All-update: finfun-All\ f(a\ \$:=\ b) = (b \wedge finfun-All-except\ [a]\ f)$
 $\langle proof \rangle$

lemma $finfun-All-All: finfun-All\ P = All\ ((\$)\ P)$
 $\langle proof \rangle$

definition $finfun-Ex :: 'a \Rightarrow f\ bool \Rightarrow bool$
where $finfun-Ex\ P = Not\ (finfun-All\ (Not\ o\ \$\ P))$

lemma $finfun-Ex-Ex: finfun-Ex\ P = Ex\ ((\$)\ P)$
 $\langle proof \rangle$

lemma $finfun-Ex-const\ [simp]: finfun-Ex\ (K\ \$\ b) = b$
 $\langle proof \rangle$

1.13 A diagonal operator for FinFuns

definition $finfun-Diag :: 'a \Rightarrow f\ 'b \Rightarrow 'a \Rightarrow f\ 'c \Rightarrow 'a \Rightarrow f\ ('b \times 'c) ((1'(\$-/-\$'))$
 $[0, 0]\ 1000)$
where $[code\ del]: (\$f, g\$) = finfun-rec\ (\lambda b. Pair\ b\ o\ \$\ g)\ (\lambda a\ b\ c. c(a\ \$:=\ (b, g\ \$\ a)))\ f$

interpretation $finfun-Diag-aux: finfun-rec-wf-aux\ \lambda b. Pair\ b\ o\ \$\ g\ \lambda a\ b\ c. c(a\ \$:=$

$(b, g \ \$ \ a)$
 $\langle proof \ \rangle$

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

lemma *finfun-Diag-const1*: $(K \$ \ b, g \$) = \text{Pair } b \circ \$ \ g$
 $\langle proof \ \rangle$

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

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

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

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

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

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

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

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

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

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

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

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

lemma *finfun-Diag-eq*: $(\$f, g\$) = (\$f', g'\$) \longleftrightarrow f = f' \wedge g = g'$
 $\langle \text{proof} \rangle$

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

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

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

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

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

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

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

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

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

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

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

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

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

<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 (λa. *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* (λa. *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 ($f :: - \Rightarrow f -$) $f \longleftrightarrow \text{True}$
 ⟨*proof*⟩

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

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

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

lemma *finfun-dom-const*:
finfun-dom ($(K\$ c) :: 'a \Rightarrow f 'b$) = ($K\$ \text{finite } (\text{UNIV} :: 'a \text{set}) \wedge c \neq \text{undefined}$)
 ⟨*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 ($(K\$ c) :: ('a :: \text{card-UNIV}) \Rightarrow f 'b$) =
 (*if* $\text{CARD}('a) = 0$ *then* ($K\$ \text{False}$) *else* *Code.abort* (*STR "finfun-dom called on finite type"*) ($\lambda-. \text{finfun-dom } (K\$ c)$))
 ⟨*proof*⟩

lemma *finfun-dom-finfunI*: $(\lambda a. f \$ a \neq \text{finfun-default } f) \in \text{finfun}$
 ⟨*proof*⟩

lemma *finfun-dom-update* [*simp*]:
finfun-dom ($f(a \$:= b)$) = (*finfun-dom* f)($a \$:= (b \neq \text{finfun-default } f)$)
including *finfun* ⟨*proof*⟩

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

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

1.18 The domain of a FinFun as a sorted list

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

where

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

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

<proof>

lemma *finfun-to-list-const-code [code]*:

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

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

<proof>

lemma *remove1-insort-insert-same*:

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

<proof>

lemma *finfun-dom-conv*:

finfun-dom f \$ x \longleftrightarrow f \$ x \neq 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*: $\text{card } A = 1 \longleftrightarrow A \neq \{\} \wedge (\forall x \in A. \forall y \in A. x = y)$
(**is** ?lhs \longleftrightarrow ?rhs)
<proof>

lemma *card-UNIV-funfun*:

defines $F == \text{funfun} :: ('a \Rightarrow 'b) \text{ set}$
shows $\text{CARD}('a \Rightarrow f 'b) = (\text{if } \text{CARD}('a) \neq 0 \wedge \text{CARD}('b) \neq 0 \vee \text{CARD}('b) = 1 \text{ then } \text{CARD}('b) \wedge \text{CARD}('a) \text{ else } 0)$
<proof>

lemma *finite-UNIV-funfun*:

$\text{finite } (\text{UNIV} :: ('a \Rightarrow f 'b) \text{ set}) \longleftrightarrow$
 $(\text{finite } (\text{UNIV} :: 'a \text{ set}) \wedge \text{finite } (\text{UNIV} :: 'b \text{ set}) \vee \text{CARD}('b) = 1)$
(**is** ?lhs \longleftrightarrow ?rhs)
<proof>

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

definition *finite-UNIV* = *Phantom*('a $\Rightarrow f$ 'b)

(*let* *cb* = *of-phantom* (*card-UNIV* :: 'b *card-UNIV*)

in *cb* = 1 \vee *of-phantom* (*finite-UNIV* :: 'a *finite-UNIV*) \wedge *cb* \neq 0)

instance

<proof>

end

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

definition *card-UNIV* = *Phantom*('a $\Rightarrow f$ 'b)

(*let* *ca* = *of-phantom* (*card-UNIV* :: 'a *card-UNIV*);

cb = *of-phantom* (*card-UNIV* :: 'b *card-UNIV*)

in *if* *ca* \neq 0 \wedge *cb* \neq 0 \vee *cb* = 1 *then* *cb* \wedge *ca* *else* 0)

instance <proof>

end

1.18.1 Bundles for concrete syntax

bundle *funfun-syntax*

begin

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

notation

funfun-const (*K*\$/ - [0] 1) **and**

funfun-update (-'(- \$:= -') [1000, 0, 0] 1000) **and**

funfun-apply (**infixl** \$ 999) **and**

funfun-comp (**infixr** \circ \$ 55) **and**

funfun-comp2 (**infixr** \$ \circ 55) **and**

funfun-Diag ((1'(\$-/ -\$')) [0, 0] 1000)

notation (*ASCII*)

```

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

end

bundle no-finfun-syntax
begin

no-type-notation
    finfun ((- =>f /-) [22, 21] 21)

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

no-notation (ASCII)
    finfun-comp (infixr o$ 55) and
    finfun-comp2 (infixr $o 55)

end

unbundle no-finfun-syntax

end

```

2 Predicates modelled as FinFuns

```

theory FinFunPred
imports FinFun
begin

unbundle finfun-syntax

Instantiate FinFun predicates just like predicates
type-synonym 'a pred_f = 'a =>f bool

instantiation finfun :: (type, ord) ord
begin

definition le-finfun-def [code del]: f ≤ g ⟷ (∀x. f $ x ≤ g $ x)

definition [code del]: (f::'a =>f 'b) < g ⟷ f ≤ g ∧ ¬ g ≤ f

instance ⟨proof⟩

```

```

lemma le-funfun-code [code]:
   $f \leq g \iff \text{finfun-All } ((\lambda(x, y). x \leq y) \circ \$ (\$f, g\$))$ 
  <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-funfun-apply [simp]: ($) bot = ( $\lambda$ -. bot)
  <proof>

instantiation finfun :: (type, order-top) order-top begin
definition top = finfun-const top
instance <proof>
end

lemma top-funfun-apply [simp]: ($) top = ( $\lambda$ -. top)
  <proof>

instantiation finfun :: (type, inf) inf begin
definition [code]:  $\text{inf } f \ g = (\lambda(x, y). \text{inf } x \ y) \circ \$ (\$f, g\$)$ 
instance <proof>
end

lemma inf-funfun-apply [simp]: ($) ( $\text{inf } f \ g$ ) =  $\text{inf } ((\$) \ f) \ ((\$) \ g)$ 
  <proof>

instantiation finfun :: (type, sup) sup begin
definition [code]:  $\text{sup } f \ g = (\lambda(x, y). \text{sup } x \ y) \circ \$ (\$f, g\$)$ 
instance <proof>
end

lemma sup-funfun-apply [simp]: ($) ( $\text{sup } f \ g$ ) =  $\text{sup } ((\$) \ f) \ ((\$) \ g)$ 
  <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 $f - g = \text{case-prod } (-) \circ \$ (f, g)$

instance *<proof>*

end

lemma *minus-finfun-apply* [*simp*]: $(\$) (f - g) = (\$) f - (\$) g$
<proof>

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

definition $- A = \text{uminus } \circ \$ A$

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*} $\equiv \text{bot}$

abbreviation *finfun-UNIV* :: 'a *pred_f*

where *finfun-UNIV* $\equiv \text{top}$

definition *finfun-single* :: 'a \Rightarrow 'a *pred_f*

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

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

finfun-single *x* $\$ y \longleftrightarrow x = y$

<proof>

lemma [*iff*]:

shows *finfun-single-neq-bot*: *finfun-single* *x* $\neq \text{bot}$

and *bot-neq-finfun-single*: *bot* $\neq \text{finfun-single } x$

<proof>

lemma *finfun-leI* [*intro!*]: $(!!x. A \$ x \Longrightarrow B \$ x) \Longrightarrow A \leq B$

<proof>

lemma *finfun-leD* [elim]: $\llbracket A \leq B; A \$ x \rrbracket \implies B \$ x$
<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 = $(\forall a. A \$ a \longrightarrow a \in \text{set } xs \vee P a)$

lemma *finfun-Ball-except-const*:

finfun-Ball-except xs (K\$ b) P $\longleftrightarrow \neg b \vee \text{set } xs = \text{UNIV} \vee \text{Code.abort } (\text{STR } "finfun-ball-except") (\lambda u. finfun-Ball-except xs (K\$ b) P)$
<proof>

lemma *finfun-Ball-except-const-finfun-UNIV-code* [code]:

finfun-Ball-except xs (K\$ b) P $\longleftrightarrow \neg b \vee \text{is-list-UNIV } xs \vee \text{Code.abort } (\text{STR } "finfun-ball-except") (\lambda u. finfun-Ball-except xs (K\$ b) P)$
<proof>

lemma *finfun-Ball-except-update*:

finfun-Ball-except xs (A(a \$:= b)) P = $((a \in \text{set } xs \vee (b \longrightarrow P a)) \wedge finfun-Ball-except (a \# xs) A P)$
<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 = $((a \in \text{set } xs \vee (b \longrightarrow P a)) \wedge finfun-Ball-except (a \# xs) f P)$
<proof>

definition *finfun-Ball* :: 'a *pred_f* \Rightarrow ('a \Rightarrow bool) \Rightarrow 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 \Rightarrow 'a *pred_f* \Rightarrow ('a \Rightarrow bool) \Rightarrow bool

where [code del]: *finfun-Bex-except* xs A P = $(\exists a. A \$ a \wedge a \notin \text{set } xs \wedge P a)$

lemma *finfun-Bex-except-const*:

finfun-Bex-except xs (K\$ b) P $\longleftrightarrow b \wedge \text{set } xs \neq \text{UNIV} \wedge \text{Code.abort } (\text{STR } "finfun-Bex-except") (\lambda u. finfun-Bex-except xs (K\$ b) P)$
<proof>

lemma *finfun-Bex-except-const-finfun-UNIV-code* [code]:

finfun-Bex-except xs (K\$ b) P $\longleftrightarrow b \wedge \neg \text{is-list-UNIV } xs \wedge \text{Code.abort } (\text{STR } "finfun-Bex-except") (\lambda u. finfun-Bex-except xs (K\$ b) P)$

<proof>

lemma *finfun-Bex-except-update*:

finfun-Bex-except xs (A(a \$:= b)) P \longleftrightarrow $(a \notin \text{set } xs \wedge b \wedge P a) \vee \text{finfun-Bex-except}$
 $(a \# xs) A P$

<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* \longleftrightarrow $((a \notin \text{set } xs \wedge b \wedge$
 $P a) \vee \text{finfun-Bex-except } (a \# xs) f P)$

<proof>

definition *finfun-Bex* :: 'a *pred*_f \Rightarrow ('a \Rightarrow bool) \Rightarrow 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]:

$\text{sup } ((\$) A) ((\$) B) = (\$) (\text{sup } A B)$

<proof>

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

$A \$ x \vee B \$ x \longleftrightarrow \text{sup } A B \$ x$

<proof>

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

$\text{inf } ((\$) A) ((\$) B) = (\$) (\text{inf } A B)$

<proof>

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

$A \$ x \wedge B \$ x \longleftrightarrow \text{inf } A B \$ x$

<proof>

lemma *iso-funfun-empty-conv* [*code-unfold*]:
 $(\lambda-. False) = (\$) \{\}_f$
 $\langle proof \rangle$

lemma *iso-funfun-UNIV-conv* [*code-unfold*]:
 $(\lambda-. True) = (\$) finfun-UNIV$
 $\langle proof \rangle$

lemma *iso-funfun-upd* [*code-unfold*]:
fixes $A :: 'a\ pred_f$
shows $(\$) A(x := b) = (\$) (A(x \ $:= b))$
 $\langle proof \rangle$

lemma *iso-funfun-uminus* [*code-unfold*]:
fixes $A :: 'a\ pred_f$
shows $- (\$) A = (\$) (- A)$
 $\langle proof \rangle$

lemma *iso-funfun-minus* [*code-unfold*]:
fixes $A :: 'a\ pred_f$
shows $(\$) A - (\$) B = (\$) (A - B)$
 $\langle proof \rangle$

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

lemma *iso-funfun-Ball-Ball*:
 $(\forall x. A \$ x \longrightarrow P x) \longleftrightarrow finfun-Ball A P$
 $\langle proof \rangle$

lemma *iso-funfun-Bex-Bex*:
 $(\exists x. A \$ x \wedge P x) \longleftrightarrow finfun-Bex A P$
 $\langle proof \rangle$

Test code setup

notepad begin
 $\langle proof \rangle$
end

declare *iso-funfun-Ball-Ball*[*code-unfold*]
notepad begin
 $\langle proof \rangle$
end
declare *iso-funfun-Ball-Ball*[*code-unfold del*]

end