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 *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 $(f(a := None)) = (map\text{-}default\ b\ f)(a := b)$
⟨*proof*⟩

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

lemma *map-default-empty* [*simp*]: *map-default b 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

typedef (*'a*,*'b*) *finfun* = $\{f::'a \Rightarrow 'b.\ \exists b.\ finite\ \{a.\ f\ a \neq b\}\}$
⟨*proof*⟩

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

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

lemma *const-finfun*: $(\lambda x.\ a) \in \mathit{finfun}$
⟨*proof*⟩

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

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

lemma *map-of-finfun*: *map-of xs* ∈ *finfun*
⟨*proof*⟩

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

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

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

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*⟩

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

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

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

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

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

lemma *Abs-finfun-inverse-finite-class*:
fixes *x* :: (*'a* :: *finite*) ⇒ *'b*
shows *Rep-finfun* (*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*

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

definition *finfun-update* :: *'a* ⇒$_f$ *'b* ⇒ *'a* ⇒ *'b* ⇒ *'a* ⇒$_f$ *'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*)

⟨*proof*⟩

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

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

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

1.4 Code generator setup

definition *finfun-update-code* :: *'a* ⇒$_f$ *'b* ⇒ *'a* ⇒ *'b* ⇒ *'a* ⇒$_f$ *'b* (-'(^{fc}/ - := -')
[*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*]:
$(\lambda^f\ b)(^f\ a := b') =$ (*if b* = *b' then* $(\lambda^f\ b)$ *else finfun-update-code* $(\lambda^f\ b)$ *a b'*)
⟨*proof*⟩

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

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* ⇒$_f$ *'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* ⇒$_f$ *'b*) × (*unit* ⇒ *Code-Evaluation.term*) ⇒ (*'a* ⇒$_f$ *'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* ⇒$_f$ *'b* × (*unit* ⇒ *Code-Evaluation.term*)) × *Random.seed* **where**

random-finfun-aux 0 j = Quickcheck.collapse (Random.select-weight
[(1, Quickcheck.random j o→ (λy. 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→ (λx. Quickcheck.random j o→
(λy. random-finfun-aux i j o→ (λf. Pair (valtermify-finfun-update-code x y f))))),
(1, Quickcheck.random j o→ (λy. Pair (valtermify-finfun-const y)))])

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

instance ⟨*proof*⟩

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)))])
⟨*proof*⟩

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* *λa f. f(*^{f} *a :: 'a := b')*
⟨*proof*⟩

lemma *fold-finfun-update-finite-univ*:
assumes *fin*: *finite* (*UNIV :: 'a set*)
shows *fold* (*λa f. f(*^{f} *a := b')*) (*λ*^{f} *b*) (*UNIV :: 'a set*) = (*λ*^{f} *b'*)
⟨*proof*⟩

1.7 Default value for FinFuns

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

lemma *finfun-default-aux-infinite*:
fixes *f* :: '*a* ⇒ '*b*
assumes *infin*: *infinite* (*UNIV :: 'a set*)
and *fin*: *finite* {*a. f a ≠ b*}
shows *finfun-default-aux f = b*
⟨*proof*⟩

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

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

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

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

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

lemma *finfun-default-update-const*:
$finfun\text{-}default\ (f(^f\ a := b)) = finfun\text{-}default\ f$
⟨*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*]:
$finfun\text{-}rec\ cnst\ upd\ f \equiv$
$let\ b = finfun\text{-}default\ f;$
$\quad g = THE\ g.\ f = Abs\text{-}finfun\ (map\text{-}default\ b\ g) \wedge finite\ (dom\ g) \wedge b \notin ran\ g$
$in\ fold\ (\lambda a.\ upd\ a\ (map\text{-}default\ b\ g\ a))\ (cnst\ b)\ (dom\ g)$

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*: $fun\text{-}left\text{-}comm\ (\lambda a.\ upd\ a\ (f\ a))$
⟨*proof*⟩

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

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

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

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

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*⟩

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*⟩

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*) ⟹ *fold* (λ*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*⟩

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*⟩

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

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

lemma *finfun-exhaust*:
obtains *b* **where** *x* = (λ^f *b*)
| *f a b* **where** *x* = *f*(^{f} *a* := *b*)
⟨*proof*⟩

lemma *finfun-rec-unique*:
fixes *f* :: *'a* ⇒$_f$ *'b* ⇒ *'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*⟩

1.10 Function application

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

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* $\lambda b.\ b\ \lambda a'\ b\ c.\ if\ (a = a')\ then\ b\ else\ c$
⟨*proof*⟩

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

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

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

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

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

lemma *finfun-apply-Rep-finfun*:
finfun-apply = *Rep-finfun*
⟨*proof*⟩

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

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

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

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

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

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*]: $g \circ_f f = finfun\text{-}rec\ (\lambda b.\ (\lambda^f\ g\ b))\ (\lambda a\ b\ c.\ c(^f\ a := g\ b))\ f$

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

interpretation *finfun-comp*: *finfun-rec-wf* $(\lambda b.\ (\lambda^f\ g\ b))\ (\lambda a\ b\ c.\ c(^f\ a := g\ b))$
⟨*proof*⟩

lemma *finfun-comp-const* [*simp*, *code*]:
$g \circ_f (\lambda^f\ c) = (\lambda^f\ g\ c)$
⟨*proof*⟩

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*]: $g \circ_f (\textit{finfun-update-code}\ f\ a\ b) = \textit{finfun-update-code}\ (g \circ_f f)\ a\ (g\ b)$
⟨*proof*⟩

lemma *finfun-comp-apply* [*simp*]:
$(g \circ_f f)_f = g \circ f_f$
⟨*proof*⟩

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

lemma *finfun-comp-const1* [*simp*]: $(\lambda x.\ c) \circ_f f = (\lambda^f\ c)$
⟨*proof*⟩

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

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

lemma *finfun-comp-conv-comp*: $g \circ_f f = \textit{Abs-finfun}\ (g \circ \textit{finfun-apply}\ f)$
⟨*proof*⟩

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*]: $\textit{finfun-comp2}\ g\ f = \textit{Abs-finfun}\ (\textit{Rep-finfun}\ g \circ f)$

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

lemma *finfun-comp2-const* [*code*, *simp*]: $\textit{finfun-comp2}\ (\lambda^f\ c)\ f = (\lambda^f\ c)$
⟨*proof*⟩

lemma *finfun-comp2-update*:
assumes *inj*: *inj f*
shows $\textit{finfun-comp2}\ (g(^f\ b := c))\ f = (\textit{if}\ b \in \textit{range}\ f\ \textit{then}\ (\textit{finfun-comp2}\ g\ f)(^f\ \textit{inv}\ f\ b := c)\ \textit{else}\ \textit{finfun-comp2}\ g\ f)$
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *card-UNIV-ge-0-finite-UNIV*:
  card-UNIV x > *0* ⟷ *finite* (*UNIV* :: *'a set*)
⟨*proof*⟩

lemma *card-UNIV-eq-0-infinite-UNIV*:
  card-UNIV x = *0* ⟷ *infinite* (*UNIV* :: *'a set*)
⟨*proof*⟩

definition *is-list-UNIV* :: *'a list* ⇒ *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* ⟷ *set xs* = *UNIV*
⟨*proof*⟩

lemma *card-UNIV-eq-0-is-list-UNIV-False*:
  assumes *cU0*: *card-UNIV x* = *0*
  shows *is-list-UNIV* = (λ*xs*. *False*)
⟨*proof*⟩

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 = (λ*a* :: *nat itself*. *0*)

instance ⟨*proof*⟩

end

1.13.2 *int*

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

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

instance ⟨*proof*⟩

end

1.13.3 *'a list*

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

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

instance ⟨*proof*⟩

end

1.13.4 *unit*

lemma *card-UNIV-unit*: *card* (*UNIV* :: *unit set*) = *1*
⟨*proof*⟩

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

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

instance ⟨*proof*⟩

end

1.13.5 *bool*

lemma *card-UNIV-bool*: *card* (*UNIV* :: *bool set*) = *2*
⟨*proof*⟩

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

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

instance ⟨*proof*⟩

end

1.13.6 *char*

lemma *card-UNIV-char*: *card* (*UNIV* :: *char set*) = *256*
⟨*proof*⟩

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

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

instance ⟨*proof*⟩

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 = (λ*a* :: (*'a* × *'b*) *itself*. *card-UNIV* (*TYPE*(*'a*)) ∗ *card-UNIV* (*TYPE*(*'b*)))

instance ⟨*proof*⟩

end

1.13.8 $'a + 'b$

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

definition *card-UNIV-sum-def*:
card-UNIV-class.card-UNIV = (λ*a* :: (*'a* + *'b*) *itself*. *let ca* = *card-UNIV* (*TYPE*(*'a*)); *cb* = *card-UNIV* (*TYPE*(*'b*))
in if ca ≠ *0* ∧ *cb* ≠ *0 then ca* + *cb else 0*)

instance ⟨*proof*⟩

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 = (λ*a* :: (*'a* ⇒ *'b*) *itself*. *let ca* = *card-UNIV* (*TYPE*(*'a*)); *cb* = *card-UNIV* (*TYPE*(*'b*))
in if ca ≠ *0* ∧ *cb* ≠ *0* ∨ *cb* = *1 then cb* ^ *ca else 0*)

instance ⟨*proof*⟩

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*⟩

end

1.14 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*$_f$ *a*

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

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

lemma *finfun-All-except-update*:
finfun-All-except A f(^{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* (λ^f *b*) = *b*
⟨*proof*⟩

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

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

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
⟨*proof*⟩

lemma *finfun-Ex-const* [*simp*]: *finfun-Ex* (λ^f *b*) = *b*
⟨*proof*⟩

1.15 A diagonal operator for FinFuns

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

interpretation *finfun-Diag*: *finfun-rec-wf* λb. *Pair b* $\circ_f$ *g* $\lambda a\ b\ c$. *c*(^{f} *a* := (*b*, g_f *a*))
⟨*proof*⟩

lemma *finfun-Diag-const1*: (λ^f *b*, *g*)^f = *Pair b* $\circ_f$ *g*
⟨*proof*⟩

Do not use (λ^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*]:
(λ^f *b*, λ^f *c*)^f = (λ^f (*b*, *c*))
(λ^f *b*, *g*(^{fc} *a* := *c*))^f = (λ^f *b*, *g*)^f(^{fc} *a* := (*b*, *c*))
⟨*proof*⟩

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*]: (*finfun-update-code f a b*, *g*)^f = (*f*, *g*)^f(^{f} *a* := (*b*, g_f *a*))
⟨*proof*⟩

lemma *finfun-Diag-const2*: (*f*, λ^f *c*)^f = (λb. (*b*, *c*)) $\circ_f$ *f*
⟨*proof*⟩

lemma *finfun-Diag-update2*: (*f*, *g*(^{f} *a* := *c*))^f = (*f*, *g*)^f(^{f} *a* := (f_f *a*, *c*))
⟨*proof*⟩

lemma *finfun-Diag-const-const* [*simp*]: (λ^f *b*, λ^f *c*)^f = (λ^f (*b*, *c*))
⟨*proof*⟩

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

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

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

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

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*⟩

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

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

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*: $\mathit{finfun\text{-}fst}\ (\lambda^f\ bc) = (\lambda^f\ \mathit{fst}\ bc)$
⟨*proof*⟩

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

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

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

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

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* $(\lambda^f\ bc) = (\lambda^f\ snd\ bc)$
⟨*proof*⟩

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*)
⟨*proof*⟩

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

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

lemma *finfun-snd-conv-Abs-finfun*: *finfun-snd* = ($\lambda f.$ *Abs-finfun* (*snd o Rep-finfun f*))
⟨*proof*⟩

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

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* $\lambda(a, b)\ c\ f.\ f(^f\ a := (f_f\ a)(^f\ b := c))$
⟨*proof*⟩

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*⟩

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

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

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} $b := c$))
⟨*proof*⟩

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*⟩

lemma *finfun-curry-conv-curry*:
fixes $f :: ('a \times 'b) \Rightarrow_f 'c$
shows *finfun-curry* f = *Abs-finfun* ($\lambda a.$ *Abs-finfun* (*curry* (*Rep-finfun* f) a))
⟨*proof*⟩

1.17 Executable equality for FinFuns

lemma *eq-finfun-All-ext*: $(f = g) \longleftrightarrow$ *finfun-All* ($(\lambda(x,\ y).\ x = y) \circ_f\ (f,\ g)^f$)
⟨*proof*⟩

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 ⟨*proof*⟩
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* ($\lambda^{f}\ b$) = ($\lambda^{f}\ b$)
⟨*proof*⟩

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

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*]: $f \leq g \longleftrightarrow (\forall x.\ f_f\ x \leq g_f\ x)$

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

instance ⟨*proof*⟩

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

end

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

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

instance ⟨*proof*⟩

end

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

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

instance ⟨*proof*⟩

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*]: $a \in_f \{\}_f = False$
⟨*proof*⟩

lemma *finfun-subsetI* [*intro*!]: $(!!x.\ x \in_f A ==> x \in_f B) ==> A \subseteq_f B$
⟨*proof*⟩

lemma *finfun-subsetD* [*elim*]: $A \subseteq_f B ==> c \in_f A ==> c \in_f B$
⟨*proof*⟩

lemma *finfun-subset-refl* [*simp*]: $A \subseteq_f A$
⟨*proof*⟩

lemma *finfun-set-ext*: $(!!x.\ (x \in_f A) = (x \in_f B)) \Longrightarrow A = B$
⟨*proof*⟩

lemma *finfun-subset-antisym* [*intro*!]: $A \subseteq_f B \Longrightarrow B \subseteq_f A \Longrightarrow A = B$
⟨*proof*⟩

lemma *finfun-Compl-iff* [*simp*]: $(c \in_f -A) = (\neg\ c \in_f A)$
⟨*proof*⟩

lemma *finfun-Un-iff* [*simp*]: $(c \in_f A \cup_f B) = (c \in_f A \mid c \in_f B)$
⟨*proof*⟩

lemma *finfun-Int-iff* [*simp*]: $(c \in_f A \cap_f B) = (c \in_f A \ \&\ c \in_f B)$
⟨*proof*⟩

lemma *finfun-Diff-iff* [*simp*]: $(c \in_f A - B) = (c \in_f A \ \&\ \neg\ c \in_f B)$
⟨*proof*⟩

lemma *finfun-insert-iff* [*simp*]: $(a \in_f insert_f\ b\ A) = (a = b \mid a \in_f A)$
⟨*proof*⟩

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*]: $loop\text{-}counting\ n\ f = f\ ()$

lemma *loop-counting-code* [*code*]: $loop\text{-}counting\ n = loop\text{-}counting\ (Suc\ n)$
⟨*proof*⟩

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

lemma *loop-code* [*code*]: $loop = loop\text{-}counting\ 0$
⟨*proof*⟩

lemma *mem-finfun-apply-conv*: $x \in f_f \longleftrightarrow f_f\ x$
⟨*proof*⟩

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*]: $finfun\text{-}Ball\text{-}except\ xs\ A\ P = (\forall\, a{\in}A_f.\ a \in set\ xs \vee P\ a)$

lemma *finfun-Ball-except-const*:
$finfun\text{-}Ball\text{-}except\ xs\ (\lambda^f\ b)\ P \longleftrightarrow \neg\ b \vee set\ xs = UNIV \vee loop\ (\lambda u.\ finfun\text{-}Ball\text{-}except\ xs\ (\lambda^f\ b)\ P)$
⟨*proof*⟩

lemma *finfun-Ball-except-const-finfun-UNIV-code* [*code*]:
$finfun\text{-}Ball\text{-}except\ xs\ (\lambda^f\ b)\ P \longleftrightarrow \neg\ b \vee is\text{-}list\text{-}UNIV\ xs \vee loop\ (\lambda u.\ finfun\text{-}Ball\text{-}except\ xs\ (\lambda^f\ b)\ P)$
⟨*proof*⟩

lemma *finfun-Ball-except-update*:
finfun-Ball-except xs $(A(^f\ a := b))\ P = ((a \in set\ xs \vee (b \longrightarrow P\ a)) \wedge$ *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*) $P = ((a \in set\ xs \vee (b \longrightarrow P\ a)) \wedge$ *finfun-Ball-except* $(a\ \#\ xs)\ f\ P)$
⟨*proof*⟩

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* []
⟨*proof*⟩

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* = $(\exists\, a{\in}A_f.\ a \notin set\ xs \wedge P\ a)$

lemma *finfun-Bex-except-const*: *finfun-Bex-except xs* $(\lambda^f\ b)\ P \longleftrightarrow b \wedge set\ xs \neq UNIV \wedge$ *loop* $(\lambda u.$ *finfun-Bex-except xs* $(\lambda^f\ b)\ P)$
⟨*proof*⟩

lemma *finfun-Bex-except-const-finfun-UNIV-code* [*code*]:
finfun-Bex-except xs $(\lambda^f\ b)\ P \longleftrightarrow b \wedge \neg$ *is-list-UNIV xs* $\wedge$ *loop* $(\lambda u.$ *finfun-Bex-except xs* $(\lambda^f\ b)\ P)$
⟨*proof*⟩

lemma *finfun-Bex-except-update*:
finfun-Bex-except xs $(A(^f\ a := b))\ P \longleftrightarrow (a \notin set\ xs \wedge b \wedge P\ a) \vee$ *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*) $P \longleftrightarrow ((a \notin set\ xs \wedge b \wedge P\ a) \vee$ *finfun-Bex-except* $(a\ \#\ xs)\ f\ P)$
⟨*proof*⟩

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* []
⟨*proof*⟩

Automatically replace set operations by finfun set operations where possible

lemma *iso-finfun-mem-mem* [*code-inline*]: $x \in A_f \longleftrightarrow x \in_f A$
⟨*proof*⟩

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

lemma *iso-finfun-subset-subset* [*code-inline*]:
$A_f \subseteq B_f \longleftrightarrow A \subseteq_f B$
⟨*proof*⟩

lemma *iso-finfun-eq* [*code-inline*]:
fixes $A :: 'a \Rightarrow_f bool$
shows $A_f = B_f \longleftrightarrow A = B$
⟨*proof*⟩

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

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

lemma *iso-finfun-empty-conv* [*code-inline*]:
$\{\} = \{\}_{ff}$
⟨*proof*⟩

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

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

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

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*:
$Ball\ (A_f)\ P \longleftrightarrow finfun\text{-}Ball\ A\ P$
⟨*proof*⟩

lemma *iso-finfun-Bex-Bex*:

$Bex\ (A_f)\ P \longleftrightarrow$ *finfun-Bex* $A\ P$
⟨*proof*⟩

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

end