A Minimal Homotopy Type Theory Object Logic for Isabelle/Pure

Arthur Freitas Ramos David Barros Hulak
Ruy J. G. B. de Queiroz

July 24, 2026

Abstract

This entry presents an independently authored higher-order abstract syntax encoding of intensional Martin-Löf type theory [7, 8] and a core homotopy type theory library [11] in Isabelle/Pure. The primitive object-logic rules are explicit axioms. Derived results cover identity-path algebra, transport, homotopies, contractibility, fibers, bi-invertibility, and the identity, inverse, and composite equivalences, so type equivalence is proved reflexive, symmetric, and transitive. The main characterization proves that a map is bi-invertible if and only if all of its fibers are contractible; contractibility is also shown to transfer along an equivalence in both directions. Function extensionality and univalence are isolated as optional additional axioms, each stated as bi-invertibility of its canonical map.

Contents

1 Scope and independence **3**

2 Representation **4**

3 Derived identity and equivalence layer **5**

4 Extensional principles **6**

5 Trust boundary **6**

6 Theory structure **6**

7 Artifact and reproducibility **7**

8 Expressions, judgments, and universes **8**

9 Dependent products **9**

10 Dependent sums **9**

11 Empty and unit types **10**

12 Natural numbers **11**

13 Non-dependent notation and functions **11**

14 Identity types and path induction **13**

15 Transport and elementary path operations **14**

16 Groupoid laws **16**

17 Functorial action and homotopies **21**

18 Functorial reduction lemmas **33**

19 Whiskering and cancellation of paths **38**

20 The self-action of a homotopy to the identity **42**

21 Contractibility and fibers **51**

22 Bi-invertible maps and equivalences **52**

23 The identity equivalence **55**

24 Composition of equivalences **57**

25 Extracting contraction data **68**

26 Contractible fibers yield a bi-inverse **70**

27 Equivalences preserve contractibility **75**

28 From a bi-inverse to a single quasi-inverse **80**

29 Coherent (half-adjoint) inverse data **84**

30 Contractibility of the fibers of a bi-invertible map **125**

31 Bi-invertible maps are contractible maps **150**

32 The bidirectional characterization **151**

33 Bi-inverses from quasi-inverse data **152**

34 The selected inverse of a bi-invertible map **154**

35 Inverting an equivalence **155**

36 Contractibility transfers along an equivalence **156**

37 Type equivalence is an equivalence relation **156**

38 The canonical map from function paths to homotopies **157**

39 Function extensionality **158**

40 The canonical map from identity to equivalence **160**

41 Univalence **161**

1 Scope and independence

The source was written from the mathematical presentation in the Homotopy Type Theory book [11]. It does not import, copy, rename, or adapt Joshua Chen's Isabelle/HoTT implementation. Chen's earlier Isabelle formalization [3] is cited as related work, not as a source dependency.

The distinct archival value is a minimal, automation-free, explicitly axiomatized HoTT object logic whose complete trusted boundary is visible in ordinary Isabelle theory files. It is intended as a reference foundation and audit target rather than as a replacement for a feature-rich interactive HoTT environment. That role is complementary to the large univalent libraries built on proof assistants with native dependent types, such as the Coq HoTT library [1] and UniMath [12], which optimize for mathematical reach rather than for a small auditable rule set.

Aspect	This entry	Chen's Isabelle/HoTT [3]
Development	Independently authored object logic from published mathematical rules	The original Isabelle/HoTT implementation
Implementation	Ordinary Isabelle/Pure theories; no custom Isabelle/ML automation	Bidirectional type checking and elaboration infrastructure
Trust presentation	Primitive rules and optional extensional axioms are visibly separated	Richer implementation and proof infrastructure for interactive use
Formal scope	Focused foundational library: the full contractible-fiber characterization of equivalences, and closure under identity, inversion, and composition	Broader foundational results and a more convenient working environment
Intended use	Explicit reference foundation and audit surface	Interactive HoTT formalization and elaboration environment

2 Representation

Object expressions inhabit a fresh Pure type. Object binders are represented with higher-order abstract syntax [10], so Pure beta substitution implements object-level substitution. Isabelle/Pure is used as a logical framework in the sense of Paulson [9, 13]; the presentation also follows the general logical-framework discipline of Harper, Honsell, and Plotkin [4]. The object logic has a dedicated typing judgment and a primitive type of universe levels. A zero level, a successor operation, and a strict universe-level relation index cumulative Russell-style universes. The relation is axiomatized by a successor step, transitivity, and irreflexivity. Pure has no falsity, so irreflexivity is stated in ex-falso form: a self-relation $i \prec_l i$ derives an arbitrary Pure proposition. Asymmetry, and hence the absence of short successor cycles, is derived from those axioms. The intended model reads levels as natural numbers, so the strictness axiom is not vacuous. The entry does not identify levels with Isabelle natural numbers or postulate an exhaustion principle for them.

Dependent products and sums use Pure functions for their families. Application and projections are object-language constants, while their beta equations are Pure equalities. The empty type has formation and elimination but no introduction rule. Unit and natural-number eliminators have the expected computation equations. Identity types have reflexivity and based path induction.

3 Derived identity and equivalence layer

Transport is defined by path induction. Path inversion is another direct instance of path induction, and composition transports the first path along the second in a family of paths with fixed source. The development constructs inhabitants of the left and right unit laws, associativity, and both inverse laws, which are the groupoid laws of the identity type in the sense of Hofmann and Streicher [5]. Functorial action on paths is obtained by induction, and homotopies are dependent products of pointwise identity types.

A contractible type packages a center and a path from that center to every point. Fibers are dependent pairs of preimages and image paths, and a contractible map has a contraction in every fiber. Left and right inverse data are kept independent; their product is bi-invertibility. An equivalence is a function equipped with such a witness.

The development proves both directions of the standard characterization. From contractible fibers it selects each fiber center as an inverse image; applying the first projection to the fiber contraction yields the left-inverse homotopy. Conversely, independent left and right inverses are combined into a single quasi-inverse and upgraded to coherent half-adjoint data. The triangle identity then supplies the dependent second-component path needed to contract each fiber. This proof derives functorial laws for `ap`, path whiskering and cancellation, homotopy naturality, and paths of dependent pairs. No new primitive rule is introduced for the characterization.

The identity function is proved bi-invertible using reflexive homotopies. For composition, the selected left inverses and selected right inverses are composed separately. Their new inverse homotopies apply the outer inverse or map to an existing pointwise path and then concatenate with the remaining inverse path. Thus closure of equivalences under composition is derived rather than postulated.

Inversion is derived from the same quasi-inverse construction. Reading the retraction and section homotopies of a bi-invertible f from the other side exhibits f itself as both a left and a right inverse of the selected inverse g, so g is bi-invertible. A general constructor turns any quasi-inverse triple into a bi-invertibility witness, which is the counterpart of the projection lemmas that read the inverse map and inverse homotopy out of an existing witness. Identity, inversion, and composition together make type equivalence reflexive, symmetric, and transitive, each witnessed by an explicit object-language term rather than by a meta-level assertion. Combining inversion with the preservation result also transfers contractibility backwards along an equivalence, so contractibility is invariant in both directions.

4 Extensional principles

For two dependent functions, `happly` sends a path of functions to the pointwise homotopy by path induction. Function extensionality is the additional axiom that this canonical map is bi-invertible. The exported `funext` term is selected from the left inverse supplied by that witness.

For types A, B in a universe, `idtoequiv` sends a universe path to an equivalence by path induction, with the identity equivalence as its reflexivity case. Univalence is the additional axiom that this canonical map is bi-invertible. The function-extensionality and univalence theories both import the intensional equivalence layer directly; neither optional axiom is hidden in the earlier theories.

5 Trust boundary

The typing judgment, universe hierarchy, primitive type formers, eliminators, and their computation equations are axiomatized in Isabelle/Pure. Isabelle checks every derived theorem relative to those axioms. Function extensionality and univalence extend that boundary only in their respective final theories. There is no custom Isabelle/ML proof automation and no oracle. Irreflexivity of the universe-level relation is encoded in Pure's ex-falso form: a hypothetical self-relation derives an arbitrary Pure proposition. That premise is impossible in the intended model, which reads levels as natural numbers, $\prec_l$ as the strict order, and the successor as $+1$; every level axiom holds there, no self-relation does, and the axiom is used only to derive level asymmetry.

The entry does not prove normalization, canonicity, consistency, decidability of type checking, or a semantic model such as the simplicial [6] or cubical [2] models of univalent foundations. It also does not include higher inductive types, cubical computation, a general inductive-definition package, or automatic universe inference.

6 Theory structure

- `HoTT_Base` introduces expressions, typing, and cumulative universes.
- `HoTT_Type_Formers` introduces dependent products and sums, empty and unit types, and natural numbers.
- `HoTT_Identity` derives transport and path algebra from path induction.
- `HoTT_Equivalence` develops contractibility, bi-invertibility, the identity equivalence, and composition of equivalences.

- `HoTT_Equivalence_Characterization` proves the bidirectional contractible-fiber criterion and that equivalences preserve contractibility.
- `HoTT_Equivalence_Inverse` inverts equivalences and collects the reflexivity, symmetry, and transitivity witnesses.
- `HoTT_Function_Extensionality` adds function extensionality as an optional axiom.
- `HoTT_Univalence` adds univalence as an optional axiom.

Both extensional principles live in separate final theories, so their additional assumptions remain visible.

7 Artifact and reproducibility

The artifact targets Isabelle2025-2 and the session `HoTT_Object_Logic`. It contains eight top-level theory files and no Isabelle/ML source. A proof-only build is:

```
.\tools\build.ps1 -Project hott-object-logic-isabelle -NoDocument
```

A clean proof and document build, followed by the AFP-style source audit, is:

```
.\tools\build.ps1 -Project hott-object-logic-isabelle -Clean
.\tools\audit.ps1 -Project hott-object-logic-isabelle
```

The equivalent portable invocation is `isabelle build -c -D <entry directory> HoTT_Object_Logic`.

The final audit records the exact source-line and theorem counts and checks for `sorry`, `oops`, missing AFP files, and generated artifacts. For this revision it reports eight theories, 132 object-logic statements, twelve headline theorems, 7,609 theory-source lines, no forbidden proof tokens, and no findings. Every proof is structured Isar: the entry contains no `apply` script, no `back` or `apply_end`, no diagnostic or counterexample command, no `smt` invocation, no unnamed statement carrying a global attribute, and no use of quick-and-dirty mode. A clean AFP-style proof and PDF build takes well under a minute on a contemporary laptop; build times are machine-dependent.

theory *HoTT-Base*
  imports *Pure*
begin

8 Expressions, judgments, and universes

typedecl *expr*
typedecl *level*

judgment *has-type* :: *expr* ⇒ *expr* ⇒ *prop*
  (**infix** ::$_h$ *50*)

consts
  level-zero :: *level*
  level-succ :: *level* ⇒ *level*
  level-less :: *level* ⇒ *level* ⇒ *prop*
    (**infix** ≺$_l$ *50*)
  universe :: *level* ⇒ *expr*

The level relation is a strict order. Pure has no falsity, so irreflexivity is stated in its ex-falso form: a hypothetical self-relation $i \prec_l i$ derives an arbitrary Pure proposition. Every axiom below holds in the intended model, which reads levels as natural numbers, *level-succ* as the successor, and $\prec_l$ as the strict order; there the irreflexivity premise is impossible, so the axiom is sound and serves only to derive asymmetry.

axiomatization where
  level-step: $i \prec_l$ *level-succ* i **and**
  level-trans: $i \prec_l j \Longrightarrow j \prec_l k \Longrightarrow i \prec_l k$ **and**
  level-irrefl: $i \prec_l i \Longrightarrow$ *PROP W* **and**
  universe-form: *universe* $i ::_h$ *universe* (*level-succ* i) **and**
  universe-cumul:
    $i \prec_l j \Longrightarrow A ::_h$ *universe* $i \Longrightarrow A ::_h$ *universe* j **and**
  type-conversion:
    $a ::_h A \Longrightarrow A \equiv B \Longrightarrow a ::_h B$ **and**
  term-conversion:
    $a \equiv b \Longrightarrow a ::_h A \Longrightarrow b ::_h A$

lemma *level-two-steps*:
  $i \prec_l$ *level-succ* (*level-succ* i)
  by (*rule level-trans*[*OF level-step level-step*])

lemma *level-asym*:
  assumes *ij*: $i \prec_l j$
    and *ji*: $j \prec_l i$
  shows *PROP W*
  by (*rule level-irrefl*[*OF level-trans*[*OF ij ji*]])

lemma *level-succ-not-below*:
  assumes *level-succ* $i \prec_l i$
  shows *PROP W*
  by (*rule level-asym*[*OF level-step assms*])

lemma *level-two-steps-not-below*:

assumes *level-succ* (*level-succ i*) $\prec_l$ *i*
shows *PROP W*
by (*rule level-asym*[*OF level-two-steps assms*])

lemma *universe-cumul-two*:
assumes *A* $::_h$ *universe i*
shows *A* $::_h$ *universe* (*level-succ* (*level-succ i*))
by (*rule universe-cumul*[*OF level-two-steps assms*])

end
theory *HoTT-Type-Formers*
imports *HoTT-Base*
begin

9 Dependent products

consts
pi-type :: *expr* ⇒ (*expr* ⇒ *expr*) ⇒ *expr*
lambda :: *expr* ⇒ (*expr* ⇒ *expr*) ⇒ *expr*
app :: *expr* ⇒ *expr* ⇒ *expr*

axiomatization where
pi-form:
A $::_h$ *universe i* $\Longrightarrow$
($\bigwedge x$. *x* $::_h$ *A* $\Longrightarrow$ *B x* $::_h$ *universe i*) $\Longrightarrow$
pi-type A B $::_h$ *universe i* **and**
pi-intro:
A $::_h$ *universe i* $\Longrightarrow$
($\bigwedge x$. *x* $::_h$ *A* $\Longrightarrow$ *b x* $::_h$ *B x*) $\Longrightarrow$
lambda A b $::_h$ *pi-type A B* **and**
pi-elim:
f $::_h$ *pi-type A B* $\Longrightarrow$ *a* $::_h$ *A* $\Longrightarrow$
app f a $::_h$ *B a* **and**
pi-beta [*simp*]:
app (*lambda A b*) *a* ≡ *b a*

10 Dependent sums

consts
sigma-type :: *expr* ⇒ (*expr* ⇒ *expr*) ⇒ *expr*
pair :: *expr* ⇒ *expr* ⇒ *expr*
first :: *expr* ⇒ *expr*
second :: *expr* ⇒ *expr*
sigma-ind ::
expr ⇒ (*expr* ⇒ *expr*) ⇒ (*expr* ⇒ *expr*) ⇒
(*expr* ⇒ *expr* ⇒ *expr*) ⇒ *expr* ⇒ *expr*

axiomatization where

sigma-form:
$A ::_h$ *universe* $i \Longrightarrow$
$(\bigwedge x.\ x ::_h A \Longrightarrow B\ x ::_h$ *universe* $i) \Longrightarrow$
sigma-type $A\ B ::_h$ *universe* i **and**
sigma-intro:
$a ::_h A \Longrightarrow b ::_h B\ a \Longrightarrow$
pair $a\ b ::_h$ *sigma-type* $A\ B$ **and**
sigma-first:
$p ::_h$ *sigma-type* $A\ B \Longrightarrow$ *first* $p ::_h A$ **and**
sigma-second:
$p ::_h$ *sigma-type* $A\ B \Longrightarrow$ *second* $p ::_h B$ (*first* p) **and**
sigma-elim:
$A ::_h$ *universe* $i \Longrightarrow$
$(\bigwedge x.\ x ::_h A \Longrightarrow B\ x ::_h$ *universe* $i) \Longrightarrow$
$(\bigwedge p.\ p ::_h$ *sigma-type* $A\ B \Longrightarrow C\ p ::_h$ *universe* $j) \Longrightarrow$
$(\bigwedge x\ y.\ x ::_h A \Longrightarrow y ::_h B\ x \Longrightarrow$
$d\ x\ y ::_h C$ (*pair* $x\ y$)) $\Longrightarrow$
$p ::_h$ *sigma-type* $A\ B \Longrightarrow$
sigma-ind $A\ B\ C\ d\ p ::_h C\ p$ **and**
sigma-first-beta [*simp*]:
first (*pair* $a\ b$) $\equiv a$ **and**
sigma-second-beta [*simp*]:
second (*pair* $a\ b$) $\equiv b$ **and**
sigma-ind-beta [*simp*]:
sigma-ind $A\ B\ C\ d$ (*pair* $a\ b$) $\equiv d\ a\ b$

11 Empty and unit types

consts
empty-type :: *expr*
empty-ind :: (*expr* ⇒ *expr*) ⇒ *expr* ⇒ *expr*
unit-type :: *expr*
unit-term :: *expr*
unit-ind :: (*expr* ⇒ *expr*) ⇒ *expr* ⇒ *expr* ⇒ *expr*

axiomatization where
empty-form: *empty-type* $::_h$ *universe* i **and**
empty-elim:
$(\bigwedge z.\ z ::_h$ *empty-type* $\Longrightarrow C\ z ::_h$ *universe* $i) \Longrightarrow$
$z ::_h$ *empty-type* $\Longrightarrow$ *empty-ind* $C\ z ::_h C\ z$ **and**
unit-form: *unit-type* $::_h$ *universe* i **and**
unit-intro: *unit-term* $::_h$ *unit-type* **and**
unit-elim:
$(\bigwedge u.\ u ::_h$ *unit-type* $\Longrightarrow C\ u ::_h$ *universe* $i) \Longrightarrow$
$d ::_h C$ *unit-term* $\Longrightarrow u ::_h$ *unit-type* $\Longrightarrow$
unit-ind $C\ d\ u ::_h C\ u$ **and**
unit-ind-beta [*simp*]:
unit-ind $C\ d$ *unit-term* $\equiv d$

12 Natural numbers

consts
natural-type :: *expr*
natural-zero :: *expr*
natural-succ :: *expr* ⇒ *expr*
natural-ind ::
(*expr* ⇒ *expr*) ⇒ *expr* ⇒
(*expr* ⇒ *expr* ⇒ *expr*) ⇒ *expr* ⇒ *expr*

axiomatization where
natural-form: *natural-type* ::$_h$ *universe i* **and**
natural-zero-intro: *natural-zero* ::$_h$ *natural-type* **and**
natural-succ-intro:
n ::$_h$ *natural-type* ⟹ *natural-succ n* ::$_h$ *natural-type* **and**
natural-elim:
(⋀*n*. *n* ::$_h$ *natural-type* ⟹ *C n* ::$_h$ *universe i*) ⟹
z ::$_h$ *C natural-zero* ⟹
(⋀*n r*. *n* ::$_h$ *natural-type* ⟹ *r* ::$_h$ *C n* ⟹
s n r ::$_h$ *C* (*natural-succ n*)) ⟹
n ::$_h$ *natural-type* ⟹
natural-ind C z s n ::$_h$ *C n* **and**
natural-ind-zero [*simp*]:
natural-ind C z s natural-zero ≡ *z* **and**
natural-ind-succ [*simp*]:
natural-ind C z s (*natural-succ n*) ≡
s n (*natural-ind C z s n*)

13 Non-dependent notation and functions

definition *arrow* :: *expr* ⇒ *expr* ⇒ *expr* **where**
arrow A B ≡ *pi-type A* (λ-. *B*)

definition *product* :: *expr* ⇒ *expr* ⇒ *expr* **where**
product A B ≡ *sigma-type A* (λ-. *B*)

definition *identity-function* :: *expr* ⇒ *expr* **where**
identity-function A ≡ *lambda A* (λ*x*. *x*)

definition *compose* ::
expr ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr*
where
compose A B C g f ≡ *lambda A* (λ*x*. *app g* (*app f x*))

lemma *arrow-form*:
assumes *A*: *A* ::$_h$ *universe i*
and *B*: *B* ::$_h$ *universe i*
shows *arrow A B* ::$_h$ *universe i*
unfolding *arrow-def*

proof (*rule pi-form*)
show $A ::_h$ *universe i* **by** (*rule A*)
fix *x*
assume $x ::_h A$
show $B ::_h$ *universe i* **by** (*rule B*)
qed

lemma *arrow-intro*:
assumes *A*: $A ::_h$ *universe i*
and *b*: $\bigwedge x.\ x ::_h A \Longrightarrow b\ x ::_h B$
shows *lambda A b* $::_h$ *arrow A B*
unfolding *arrow-def*
by (*rule pi-intro*[*OF A b*])

lemma *arrow-elim*:
assumes *f*: $f ::_h$ *arrow A B*
and *a*: $a ::_h A$
shows *app f a* $::_h B$
using *f* **unfolding** *arrow-def*
by (*rule pi-elim*[*OF - a*])

lemma *product-form*:
assumes *A*: $A ::_h$ *universe i*
and *B*: $B ::_h$ *universe i*
shows *product A B* $::_h$ *universe i*
unfolding *product-def*
proof (*rule sigma-form*)
show $A ::_h$ *universe i* **by** (*rule A*)
fix *x*
assume $x ::_h A$
show $B ::_h$ *universe i* **by** (*rule B*)
qed

lemma *product-intro*:
assumes *a*: $a ::_h A$
and *b*: $b ::_h B$
shows *pair a b* $::_h$ *product A B*
unfolding *product-def*
by (*rule sigma-intro*[*OF a b*])

lemma *product-first*:
assumes $p ::_h$ *product A B*
shows *first p* $::_h A$
using *assms* **unfolding** *product-def* **by** (*rule sigma-first*)

lemma *product-second*:
assumes $p ::_h$ *product A B*
shows *second p* $::_h B$
using *assms* **unfolding** *product-def* **by** (*rule sigma-second*)

lemma *identity-function-type*:
assumes *A*: $A ::_h$ *universe i*
shows *identity-function A* $::_h$ *arrow A A*
unfolding *identity-function-def*
by (*rule arrow-intro*[*OF A*], *assumption*)

lemma *compose-type*:
assumes *A*: $A ::_h$ *universe i*
and *f*: $f ::_h$ *arrow A B*
and *g*: $g ::_h$ *arrow B C*
shows *compose A B C g f* $::_h$ *arrow A C*
unfolding *compose-def*
proof (*rule arrow-intro*[*OF A*])
fix *x*
assume *x*: $x ::_h A$
have *fx*: *app f x* $::_h$ *B*
by (*rule arrow-elim*[*OF f x*])
show *app g* (*app f x*) $::_h$ *C*
by (*rule arrow-elim*[*OF g fx*])
qed

lemma *apply-identity* [*simp*]:
app (*identity-function A*) $x \equiv x$
unfolding *identity-function-def* **by** *simp*

lemma *apply-compose* [*simp*]:
app (*compose A B C g f*) $x \equiv$ *app g* (*app f x*)
unfolding *compose-def* **by** *simp*

end
theory *HoTT-Identity*
imports *HoTT-Type-Formers*
begin

14 Identity types and path induction

consts
identity-type :: *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr*
reflexivity :: *expr* ⇒ *expr* ⇒ *expr*
identity-ind ::
expr ⇒ *expr* ⇒ (*expr* ⇒ *expr* ⇒ *expr*) ⇒
expr ⇒ *expr* ⇒ *expr* ⇒ *expr*

axiomatization where
identity-form:
$A ::_h$ *universe i* $\Longrightarrow$ $x ::_h A \Longrightarrow$ $y ::_h A \Longrightarrow$
identity-type A x y $::_h$ *universe i* **and**
identity-intro:

$A ::_h$ *universe* $i \Longrightarrow x ::_h A \Longrightarrow$
reflexivity $A\ x ::_h$ *identity-type* $A\ x\ x$ **and**
identity-elim:
$A ::_h$ *universe* $i \Longrightarrow x ::_h A \Longrightarrow$
$(\bigwedge y\ p.\ y ::_h A \Longrightarrow p ::_h$ *identity-type* $A\ x\ y \Longrightarrow$
$C\ y\ p ::_h$ *universe* $j) \Longrightarrow$
$d ::_h C\ x$ (*reflexivity* $A\ x$) $\Longrightarrow$
$y ::_h A \Longrightarrow p ::_h$ *identity-type* $A\ x\ y \Longrightarrow$
identity-ind $A\ x\ C\ d\ y\ p ::_h C\ y\ p$ **and**
identity-beta [*simp*]:
identity-ind $A\ x\ C\ d\ x$ (*reflexivity* $A\ x$) $\equiv d$

15 Transport and elementary path operations

definition *transport* ::
expr ⇒ (*expr* ⇒ *expr*) ⇒ *expr* ⇒ *expr* ⇒
expr ⇒ *expr* ⇒ *expr*
where
transport $A\ P\ x\ y\ p\ u \equiv$
identity-ind $A\ x$ (λy -. $P\ y$) $u\ y\ p$

lemma *transport-type*:
assumes A: $A ::_h$ *universe* i
and P: $\bigwedge w.\ w ::_h A \Longrightarrow P\ w ::_h$ *universe* j
and x: $x ::_h A$
and y: $y ::_h A$
and p: $p ::_h$ *identity-type* $A\ x\ y$
and u: $u ::_h P\ x$
shows *transport* $A\ P\ x\ y\ p\ u ::_h P\ y$
unfolding *transport-def*
proof (*rule identity-elim*[*OF* $A\ x$])
fix $w\ q$
assume $w ::_h A$
then show $P\ w ::_h$ *universe* j **by** (*rule* P)
next
show $u ::_h P\ x$ **by** (*rule* u)
next
show $y ::_h A$ **by** (*rule* y)
next
show $p ::_h$ *identity-type* $A\ x\ y$ **by** (*rule* p)
qed

lemma *transport-refl* [*simp*]:
transport $A\ P\ x\ x$ (*reflexivity* $A\ x$) $u \equiv u$
unfolding *transport-def* **by** *simp*

definition *path-inverse* ::
expr ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr*
where

```
  path-inverse A x y p ≡
    identity-ind A x
      (λy -. identity-type A y x)
      (reflexivity A x) y p

lemma path-inverse-type:
  assumes A: A ::_h universe i
    and x: x ::_h A
    and y: y ::_h A
    and p: p ::_h identity-type A x y
  shows path-inverse A x y p ::_h identity-type A y x
  unfolding path-inverse-def
proof (rule identity-elim[OF A x])
  fix w q
  assume w: w ::_h A
  assume q ::_h identity-type A x w
  show identity-type A w x ::_h universe i
    by (rule identity-form[OF A w x])
next
  show reflexivity A x ::_h identity-type A x x
    by (rule identity-intro[OF A x])
next
  show y ::_h A by (rule y)
next
  show p ::_h identity-type A x y by (rule p)
qed

lemma path-inverse-refl [simp]:
  path-inverse A x x (reflexivity A x) ≡ reflexivity A x
  unfolding path-inverse-def by simp

definition path-concat ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒
    expr ⇒ expr ⇒ expr
where
  path-concat A x y z p q ≡
    transport A (λw. identity-type A x w) y z q p

lemma path-concat-type:
  assumes A: A ::_h universe i
    and x: x ::_h A
    and y: y ::_h A
    and z: z ::_h A
    and p: p ::_h identity-type A x y
    and q: q ::_h identity-type A y z
  shows path-concat A x y z p q ::_h identity-type A x z
  unfolding path-concat-def
proof (rule transport-type[OF A])
  fix w
```

assume w: $w ::_h A$
show *identity-type* $A\ x\ w ::_h$ *universe* i
by (*rule identity-form*[*OF* $A\ x\ w$])
next
show $y ::_h A$ **by** (*rule* y)
next
show $z ::_h A$ **by** (*rule* z)
next
show $q ::_h$ *identity-type* $A\ y\ z$ **by** (*rule* q)
next
show $p ::_h$ *identity-type* $A\ x\ y$ **by** (*rule* p)
qed

lemma *path-concat-refl-right* [*simp*]:
path-concat $A\ x\ y\ y\ p$ (*reflexivity* $A\ y$) $\equiv p$
unfolding *path-concat-def* **by** *simp*

16 Groupoid laws

definition *path-right-unit* ::
$expr \Rightarrow expr \Rightarrow expr \Rightarrow expr \Rightarrow expr$
where
path-right-unit $A\ x\ y\ p \equiv$
reflexivity (*identity-type* $A\ x\ y$) p

lemma *path-right-unit-type*:
assumes A: $A ::_h$ *universe* i
and x: $x ::_h A$
and y: $y ::_h A$
and p: $p ::_h$ *identity-type* $A\ x\ y$
shows *path-right-unit* $A\ x\ y\ p ::_h$
identity-type (*identity-type* $A\ x\ y$)
(*path-concat* $A\ x\ y\ y\ p$ (*reflexivity* $A\ y$)) p
proof −
have *path-space*: *identity-type* $A\ x\ y ::_h$ *universe* i
by (*rule identity-form*[*OF* $A\ x\ y$])
have *reflexivity* (*identity-type* $A\ x\ y$) $p ::_h$
identity-type (*identity-type* $A\ x\ y$) $p\ p$
by (*rule identity-intro*[*OF path-space* p])
then show *path-right-unit* $A\ x\ y\ p ::_h$
identity-type (*identity-type* $A\ x\ y$)
(*path-concat* $A\ x\ y\ y\ p$ (*reflexivity* $A\ y$)) p
unfolding *path-right-unit-def*
by *simp*
qed

definition *path-left-unit* ::
$expr \Rightarrow expr \Rightarrow expr \Rightarrow expr \Rightarrow expr$
where

```
  path-left-unit A x y p ≡
    identity-ind A x
      (λy p. identity-type (identity-type A x y)
        (path-concat A x x y (reflexivity A x) p) p)
      (reflexivity (identity-type A x x) (reflexivity A x)) y p

lemma path-left-unit-type:
  assumes A: A ::h universe i
    and x: x ::h A
    and y: y ::h A
    and p: p ::h identity-type A x y
  shows path-left-unit A x y p ::h
    identity-type (identity-type A x y)
      (path-concat A x x y (reflexivity A x) p) p
  unfolding path-left-unit-def
proof (rule identity-elim[OF A x])
  fix w q
  assume w: w ::h A
  assume q: q ::h identity-type A x w
  have path-space: identity-type A x w ::h universe i
    by (rule identity-form[OF A x w])
  have composed: path-concat A x x w (reflexivity A x) q
      ::h identity-type A x w
    by (rule path-concat-type[OF A x x w identity-intro[OF A x] q])
  show identity-type (identity-type A x w)
      (path-concat A x x w (reflexivity A x) q) q ::h universe i
    by (rule identity-form[OF path-space composed q])
next
  have path-space: identity-type A x x ::h universe i
    by (rule identity-form[OF A x x])
  have base: reflexivity (identity-type A x x) (reflexivity A x)
      ::h identity-type (identity-type A x x)
        (reflexivity A x) (reflexivity A x)
    by (rule identity-intro[OF path-space identity-intro[OF A x]])
  show reflexivity (identity-type A x x) (reflexivity A x)
      ::h identity-type (identity-type A x x)
        (path-concat A x x x (reflexivity A x) (reflexivity A x))
        (reflexivity A x)
    using base by simp
next
  show y ::h A by (rule y)
next
  show p ::h identity-type A x y by (rule p)
qed

definition path-assoc ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr ⇒
    expr ⇒ expr ⇒ expr ⇒ expr
where
```

```
  path-assoc A w x y z p q r ≡
    identity-ind A y
      (λz r. identity-type (identity-type A w z)
        (path-concat A w y z (path-concat A w x y p q) r)
        (path-concat A w x z p (path-concat A x y z q r)))
      (reflexivity (identity-type A w y) (path-concat A w x y p q))
      z r

lemma path-assoc-type:
  assumes A: A ::h universe i
    and w: w ::h A
    and x: x ::h A
    and y: y ::h A
    and z: z ::h A
    and p: p ::h identity-type A w x
    and q: q ::h identity-type A x y
    and r: r ::h identity-type A y z
  shows path-assoc A w x y z p q r ::h
    identity-type (identity-type A w z)
      (path-concat A w y z (path-concat A w x y p q) r)
      (path-concat A w x z p (path-concat A x y z q r))
  unfolding path-assoc-def
proof (rule identity-elim[OF A y])
  fix v s
  assume v: v ::h A
  assume s: s ::h identity-type A y v
  have pq: path-concat A w x y p q ::h identity-type A w y
    by (rule path-concat-type[OF A w x y p q])
  have lhs: path-concat A w y v (path-concat A w x y p q) s
      ::h identity-type A w v
    by (rule path-concat-type[OF A w y v pq s])
  have qr: path-concat A x y v q s ::h identity-type A x v
    by (rule path-concat-type[OF A x y v q s])
  have rhs: path-concat A w x v p (path-concat A x y v q s)
      ::h identity-type A w v
    by (rule path-concat-type[OF A w x v p qr])
  have path-space: identity-type A w v ::h universe i
    by (rule identity-form[OF A w v])
  show identity-type (identity-type A w v)
      (path-concat A w y v (path-concat A w x y p q) s)
      (path-concat A w x v p (path-concat A x y v q s))
      ::h universe i
    by (rule identity-form[OF path-space lhs rhs])
next
  have pq: path-concat A w x y p q ::h identity-type A w y
    by (rule path-concat-type[OF A w x y p q])
  have path-space: identity-type A w y ::h universe i
    by (rule identity-form[OF A w y])
  have base: reflexivity (identity-type A w y) (path-concat A w x y p q)
```

```
    ::_h identity-type (identity-type A w y)
      (path-concat A w x y p q) (path-concat A w x y p q)
   by (rule identity-intro[OF path-space pq])
 show reflexivity (identity-type A w y) (path-concat A w x y p q)
    ::_h identity-type (identity-type A w y)
      (path-concat A w y y (path-concat A w x y p q) (reflexivity A y))
      (path-concat A w x y p
        (path-concat A x y y q (reflexivity A y)))
   using base by simp
next
 show z ::_h A by (rule z)
next
 show r ::_h identity-type A y z by (rule r)
qed

definition path-inverse-right ::
 expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr
where
 path-inverse-right A x y p ≡
   identity-ind A x
     (λy p. identity-type (identity-type A x x)
       (path-concat A x y x p (path-inverse A x y p))
       (reflexivity A x))
     (reflexivity (identity-type A x x) (reflexivity A x)) y p

lemma path-inverse-right-type:
 assumes A: A ::_h universe i
   and x: x ::_h A
   and y: y ::_h A
   and p: p ::_h identity-type A x y
 shows path-inverse-right A x y p ::_h
   identity-type (identity-type A x x)
     (path-concat A x y x p (path-inverse A x y p))
     (reflexivity A x)
 unfolding path-inverse-right-def
proof (rule identity-elim[OF A x])
 fix w q
 assume w: w ::_h A
 assume q: q ::_h identity-type A x w
 have inv: path-inverse A x w q ::_h identity-type A w x
   by (rule path-inverse-type[OF A x w q])
 have composed: path-concat A x w x q (path-inverse A x w q)
    ::_h identity-type A x x
   by (rule path-concat-type[OF A x w x q inv])
 have path-space: identity-type A x x ::_h universe i
   by (rule identity-form[OF A x x])
 show identity-type (identity-type A x x)
    (path-concat A x w x q (path-inverse A x w q))
    (reflexivity A x) ::_h universe i
```

```
    by (rule identity-form[OF path-space composed identity-intro[OF A x]])
next
  have path-space: identity-type A x x ::_h universe i
    by (rule identity-form[OF A x x])
  have base: reflexivity (identity-type A x x) (reflexivity A x)
      ::_h identity-type (identity-type A x x)
        (reflexivity A x) (reflexivity A x)
    by (rule identity-intro[OF path-space identity-intro[OF A x]])
  show reflexivity (identity-type A x x) (reflexivity A x)
      ::_h identity-type (identity-type A x x)
        (path-concat A x x x (reflexivity A x)
          (path-inverse A x x (reflexivity A x)))
        (reflexivity A x)
    using base by simp
next
  show y ::_h A by (rule y)
next
  show p ::_h identity-type A x y by (rule p)
qed

definition path-inverse-left ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr
where
  path-inverse-left A x y p ≡
    identity-ind A x
      (λy p. identity-type (identity-type A y y)
        (path-concat A y x y (path-inverse A x y p) p)
        (reflexivity A y))
      (reflexivity (identity-type A x x) (reflexivity A x)) y p

lemma path-inverse-left-type:
  assumes A: A ::_h universe i
    and x: x ::_h A
    and y: y ::_h A
    and p: p ::_h identity-type A x y
  shows path-inverse-left A x y p ::_h
    identity-type (identity-type A y y)
      (path-concat A y x y (path-inverse A x y p) p)
      (reflexivity A y)
  unfolding path-inverse-left-def
proof (rule identity-elim[OF A x])
  fix w q
  assume w: w ::_h A
  assume q: q ::_h identity-type A x w
  have inv: path-inverse A x w q ::_h identity-type A w x
    by (rule path-inverse-type[OF A x w q])
  have composed: path-concat A w x w (path-inverse A x w q) q
      ::_h identity-type A w w
    by (rule path-concat-type[OF A w x w inv q])
```

```
  have path-space: identity-type A w w ::_h universe i
    by (rule identity-form[OF A w w])
  show identity-type (identity-type A w w)
      (path-concat A w x w (path-inverse A x w q) q)
      (reflexivity A w) ::_h universe i
    by (rule identity-form[OF path-space composed identity-intro[OF A w]])
next
  have path-space: identity-type A x x ::_h universe i
    by (rule identity-form[OF A x x])
  have base: reflexivity (identity-type A x x) (reflexivity A x)
      ::_h identity-type (identity-type A x x)
        (reflexivity A x) (reflexivity A x)
    by (rule identity-intro[OF path-space identity-intro[OF A x]])
  show reflexivity (identity-type A x x) (reflexivity A x)
      ::_h identity-type (identity-type A x x)
        (path-concat A x x x
          (path-inverse A x x (reflexivity A x)) (reflexivity A x))
        (reflexivity A x)
    using base by simp
next
  show y ::_h A by (rule y)
next
  show p ::_h identity-type A x y by (rule p)
qed
```

17 Functorial action and homotopies

```
definition ap ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr ⇒
    expr ⇒ expr
where
  ap A B f x y p ≡
    identity-ind A x
      (λy -. identity-type B (app f x) (app f y))
      (reflexivity B (app f x)) y p

lemma ap-type:
  assumes A: A ::_h universe i
    and B: B ::_h universe i
    and f: f ::_h arrow A B
    and x: x ::_h A
    and y: y ::_h A
    and p: p ::_h identity-type A x y
  shows ap A B f x y p ::_h
    identity-type B (app f x) (app f y)
  unfolding ap-def
proof (rule identity-elim[OF A x])
  fix w q
  assume w: w ::_h A
```

```
  assume q ::h identity-type A x w
  have fx: app f x ::h B by (rule arrow-elim[OF f x])
  have fw: app f w ::h B by (rule arrow-elim[OF f w])
  show identity-type B (app f x) (app f w) ::h universe i
    by (rule identity-form[OF B fx fw])
next
  have fx: app f x ::h B by (rule arrow-elim[OF f x])
  show reflexivity B (app f x) ::h
      identity-type B (app f x) (app f x)
    by (rule identity-intro[OF B fx])
next
  show y ::h A by (rule y)
next
  show p ::h identity-type A x y by (rule p)
qed

lemma ap-refl [simp]:
  ap A B f x x (reflexivity A x) ≡ reflexivity B (app f x)
  unfolding ap-def by simp

definition ap-concat ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr ⇒
    expr ⇒ expr ⇒ expr ⇒ expr
where
  ap-concat A B f x y z p q ≡
    identity-ind A y
      (λz q. identity-type
        (identity-type B (app f x) (app f z))
        (ap A B f x z (path-concat A x y z p q))
        (path-concat B (app f x) (app f y) (app f z)
          (ap A B f x y p) (ap A B f y z q)))
      (reflexivity
        (identity-type B (app f x) (app f y))
        (ap A B f x y p))
      z q

lemma ap-concat-type:
  assumes A: A ::h universe i
    and B: B ::h universe i
    and f: f ::h arrow A B
    and x: x ::h A
    and y: y ::h A
    and z: z ::h A
    and p: p ::h identity-type A x y
    and q: q ::h identity-type A y z
  shows ap-concat A B f x y z p q ::h
    identity-type (identity-type B (app f x) (app f z))
      (ap A B f x z (path-concat A x y z p q))
      (path-concat B (app f x) (app f y) (app f z)
```

```
      (ap A B f x y p) (ap A B f y z q))
  unfolding ap-concat-def
proof (rule identity-elim[OF A y])
  fix w r
  assume w: w ::_h A
  assume r: r ::_h identity-type A y w
  have fx: app f x ::_h B by (rule arrow-elim[OF f x])
  have fy: app f y ::_h B by (rule arrow-elim[OF f y])
  have fw: app f w ::_h B by (rule arrow-elim[OF f w])
  have pq: path-concat A x y w p r ::_h identity-type A x w
    by (rule path-concat-type[OF A x y w p r])
  have ap-pq: ap A B f x w (path-concat A x y w p r) ::_h
      identity-type B (app f x) (app f w)
    by (rule ap-type[OF A B f x w pq])
  have ap-p: ap A B f x y p ::_h
      identity-type B (app f x) (app f y)
    by (rule ap-type[OF A B f x y p])
  have ap-r: ap A B f y w r ::_h
      identity-type B (app f y) (app f w)
    by (rule ap-type[OF A B f y w r])
  have composite: path-concat B (app f x) (app f y) (app f w)
      (ap A B f x y p) (ap A B f y w r) ::_h
      identity-type B (app f x) (app f w)
    by (rule path-concat-type[OF B fx fy fw ap-p ap-r])
  have path-space: identity-type B (app f x) (app f w)
      ::_h universe i
    by (rule identity-form[OF B fx fw])
  show identity-type (identity-type B (app f x) (app f w))
      (ap A B f x w (path-concat A x y w p r))
      (path-concat B (app f x) (app f y) (app f w)
        (ap A B f x y p) (ap A B f y w r)) ::_h universe i
    by (rule identity-form[OF path-space ap-pq composite])
next
  have fx: app f x ::_h B by (rule arrow-elim[OF f x])
  have fy: app f y ::_h B by (rule arrow-elim[OF f y])
  have ap-p: ap A B f x y p ::_h
      identity-type B (app f x) (app f y)
    by (rule ap-type[OF A B f x y p])
  have path-space: identity-type B (app f x) (app f y)
      ::_h universe i
    by (rule identity-form[OF B fx fy])
  show reflexivity (identity-type B (app f x) (app f y))
      (ap A B f x y p) ::_h
      identity-type (identity-type B (app f x) (app f y))
        (ap A B f x y (path-concat A x y y p (reflexivity A y)))
        (path-concat B (app f x) (app f y) (app f y)
          (ap A B f x y p)
          (ap A B f y y (reflexivity A y)))
    using identity-intro[OF path-space ap-p] by simp
```

```
next
  show z ::h A by (rule z)
next
  show q ::h identity-type A y z by (rule q)
qed

definition ap-inverse ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr ⇒
    expr ⇒ expr
where
  ap-inverse A B f x y p ≡
    identity-ind A x
      (λy p. identity-type
        (identity-type B (app f y) (app f x))
        (ap A B f y x (path-inverse A x y p))
        (path-inverse B (app f x) (app f y) (ap A B f x y p)))
      (reflexivity
        (identity-type B (app f x) (app f x))
        (reflexivity B (app f x)))
      y p

lemma ap-inverse-type:
  assumes A: A ::h universe i
    and B: B ::h universe i
    and f: f ::h arrow A B
    and x: x ::h A
    and y: y ::h A
    and p: p ::h identity-type A x y
  shows ap-inverse A B f x y p ::h
    identity-type (identity-type B (app f y) (app f x))
      (ap A B f y x (path-inverse A x y p))
      (path-inverse B (app f x) (app f y) (ap A B f x y p))
  unfolding ap-inverse-def
proof (rule identity-elim[OF A x])
  fix w q
  assume w: w ::h A
  assume q: q ::h identity-type A x w
  have fx: app f x ::h B by (rule arrow-elim[OF f x])
  have fw: app f w ::h B by (rule arrow-elim[OF f w])
  have inv-q: path-inverse A x w q ::h identity-type A w x
    by (rule path-inverse-type[OF A x w q])
  have lhs: ap A B f w x (path-inverse A x w q) ::h
      identity-type B (app f w) (app f x)
    by (rule ap-type[OF A B f w x inv-q])
  have ap-q: ap A B f x w q ::h
      identity-type B (app f x) (app f w)
    by (rule ap-type[OF A B f x w q])
  have rhs: path-inverse B (app f x) (app f w) (ap A B f x w q)
      ::h identity-type B (app f w) (app f x)
```

```
    by (rule path-inverse-type[OF B fx fw ap-q])
  have path-space: identity-type B (app f w) (app f x)
      ::h universe i
    by (rule identity-form[OF B fw fx])
  show identity-type (identity-type B (app f w) (app f x))
      (ap A B f w x (path-inverse A x w q))
      (path-inverse B (app f x) (app f w) (ap A B f x w q))
      ::h universe i
    by (rule identity-form[OF path-space lhs rhs])
next
  have fx: app f x ::h B by (rule arrow-elim[OF f x])
  have loop-space: identity-type B (app f x) (app f x)
      ::h universe i
    by (rule identity-form[OF B fx fx])
  have refl-path: reflexivity B (app f x) ::h
      identity-type B (app f x) (app f x)
    by (rule identity-intro[OF B fx])
  show reflexivity (identity-type B (app f x) (app f x))
      (reflexivity B (app f x)) ::h
      identity-type (identity-type B (app f x) (app f x))
        (ap A B f x x (path-inverse A x x (reflexivity A x)))
        (path-inverse B (app f x) (app f x)
          (ap A B f x x (reflexivity A x)))
    using identity-intro[OF loop-space refl-path] by simp
next
  show y ::h A by (rule y)
next
  show p ::h identity-type A x y by (rule p)
qed

definition pair-second-map ::
  expr ⇒ (expr ⇒ expr) ⇒ expr ⇒ expr
where
  pair-second-map A B x ≡ lambda (B x) (λu. pair x u)

lemma pair-second-map-type:
  assumes A: A ::h universe i
    and B: ⋀x. x ::h A ⟹ B x ::h universe i
    and x: x ::h A
  shows pair-second-map A B x ::h arrow (B x) (sigma-type A B)
  unfolding pair-second-map-def
proof (rule arrow-intro[OF B[OF x]])
  fix u
  assume u: u ::h B x
  show pair x u ::h sigma-type A B
    by (rule sigma-intro[where B=B, OF x u])
qed

definition sigma-path ::
```

```
  expr ⇒ (expr ⇒ expr) ⇒ expr ⇒ expr ⇒
    expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr
where
  sigma-path A B x y u v p q ≡
    app
      (app
        (identity-ind A x
          (λy p. pi-type (B y) (λv.
            arrow
              (identity-type (B y) (transport A B x y p u) v)
              (identity-type (sigma-type A B) (pair x u) (pair y v))))
          (lambda (B x) (λv.
            lambda (identity-type (B x) u v) (λr.
              ap (B x) (sigma-type A B) (pair-second-map A B x)
                u v r)))
          y p)
        v)
      q

lemma sigma-path-type:
  assumes A: A ::h universe i
    and B: ⋀x. x ::h A ⟹ B x ::h universe i
    and x: x ::h A
    and y: y ::h A
    and u: u ::h B x
    and v: v ::h B y
    and p: p ::h identity-type A x y
    and q: q ::h
      identity-type (B y) (transport A B x y p u) v
  shows sigma-path A B x y u v p q ::h
    identity-type (sigma-type A B) (pair x u) (pair y v)
proof −
  have Sigma: sigma-type A B ::h universe i
    by (rule sigma-form[OF A B])
  have xu: pair x u ::h sigma-type A B
    by (rule sigma-intro[where B=B, OF x u])
  have motive:
    ⋀w r. w ::h A ⟹
      r ::h identity-type A x w ⟹
      pi-type (B w) (λt.
        arrow
          (identity-type (B w) (transport A B x w r u) t)
          (identity-type (sigma-type A B) (pair x u) (pair w t)))
        ::h universe i
  proof −
    fix w r
    assume w: w ::h A
    assume r: r ::h identity-type A x w
    have transported: transport A B x w r u ::h B w
```

```
    by (rule transport-type[where P=B, OF A B x w r u])
  show pi-type (B w) (λt.
      arrow
        (identity-type (B w) (transport A B x w r u) t)
        (identity-type (sigma-type A B) (pair x u) (pair w t)))
      ::h universe i
  proof (rule pi-form[OF B[OF w]])
    fix t
    assume t: t ::h B w
    have wt: pair w t ::h sigma-type A B
      by (rule sigma-intro[where B=B, OF w t])
    have path-domain:
      identity-type (B w) (transport A B x w r u) t
        ::h universe i
      by (rule identity-form[OF B[OF w] transported t])
    have path-codomain:
      identity-type (sigma-type A B) (pair x u) (pair w t)
        ::h universe i
      by (rule identity-form[OF Sigma xu wt])
    show arrow
        (identity-type (B w) (transport A B x w r u) t)
        (identity-type (sigma-type A B) (pair x u) (pair w t))
        ::h universe i
      by (rule arrow-form[OF path-domain path-codomain])
  qed
qed
have pair-map: pair-second-map A B x ::h
    arrow (B x) (sigma-type A B)
  by (rule pair-second-map-type[OF A B x])
have base-raw: lambda (B x) (λt.
    lambda (identity-type (B x) u t) (λr.
      ap (B x) (sigma-type A B) (pair-second-map A B x) u t r))
    ::h
    pi-type (B x) (λt.
      arrow (identity-type (B x) u t)
        (identity-type (sigma-type A B) (pair x u) (pair x t)))
proof (rule pi-intro[OF B[OF x]])
  fix t
  assume t: t ::h B x
  have path-domain: identity-type (B x) u t ::h universe i
    by (rule identity-form[OF B[OF x] u t])
  show lambda (identity-type (B x) u t) (λr.
      ap (B x) (sigma-type A B) (pair-second-map A B x) u t r)
      ::h arrow (identity-type (B x) u t)
        (identity-type (sigma-type A B) (pair x u) (pair x t))
  proof (rule arrow-intro[OF path-domain])
    fix r
    assume r: r ::h identity-type (B x) u t
    show ap (B x) (sigma-type A B) (pair-second-map A B x) u t r
```

```
      ::h identity-type (sigma-type A B) (pair x u) (pair x t)
    using ap-type[OF B[OF x] Sigma pair-map u t r]
    unfolding pair-second-map-def by simp
  qed
qed
have base: lambda (B x) (λt.
    lambda (identity-type (B x) u t) (λr.
      ap (B x) (sigma-type A B) (pair-second-map A B x) u t r))
    ::h
    pi-type (B x) (λt.
      arrow
        (identity-type (B x)
          (transport A B x x (reflexivity A x) u) t)
        (identity-type (sigma-type A B) (pair x u) (pair x t)))
  using base-raw by simp
have inducted: identity-ind A x
    (λw r. pi-type (B w) (λt.
      arrow
        (identity-type (B w) (transport A B x w r u) t)
        (identity-type (sigma-type A B) (pair x u) (pair w t))))
    (lambda (B x) (λt.
      lambda (identity-type (B x) u t) (λr.
        ap (B x) (sigma-type A B) (pair-second-map A B x) u t r)))
    y p ::h
    pi-type (B y) (λt.
      arrow
        (identity-type (B y) (transport A B x y p u) t)
        (identity-type (sigma-type A B) (pair x u) (pair y t)))
proof (rule identity-elim[OF A x])
  fix w r
  assume w: w ::h A
  assume r: r ::h identity-type A x w
  show pi-type (B w) (λt.
      arrow
        (identity-type (B w) (transport A B x w r u) t)
        (identity-type (sigma-type A B) (pair x u) (pair w t)))
      ::h universe i
    by (rule motive[OF w r])
next
  show lambda (B x) (λt.
      lambda (identity-type (B x) u t) (λr.
        ap (B x) (sigma-type A B) (pair-second-map A B x) u t r))
      ::h
      pi-type (B x) (λt.
        arrow
          (identity-type (B x)
            (transport A B x x (reflexivity A x) u) t)
          (identity-type (sigma-type A B) (pair x u) (pair x t)))
    by (rule base)
```

```
  next
    show y ::h A by (rule y)
  next
    show p ::h identity-type A x y by (rule p)
  qed
  have at-v: app
      (identity-ind A x
        (λw r. pi-type (B w) (λt.
          arrow
            (identity-type (B w) (transport A B x w r u) t)
            (identity-type (sigma-type A B) (pair x u) (pair w t))))
        (lambda (B x) (λt.
          lambda (identity-type (B x) u t) (λr.
            ap (B x) (sigma-type A B) (pair-second-map A B x) u t r)))
        y p)
      v ::h
      arrow
        (identity-type (B y) (transport A B x y p u) v)
        (identity-type (sigma-type A B) (pair x u) (pair y v))
    by (rule pi-elim[OF inducted v])
  show sigma-path A B x y u v p q ::h
      identity-type (sigma-type A B) (pair x u) (pair y v)
    unfolding sigma-path-def
    by (rule arrow-elim[OF at-v q])
qed

definition homotopy ::
  expr ⇒ (expr ⇒ expr) ⇒ expr ⇒ expr ⇒ expr
where
  homotopy A B f g ≡
    pi-type A (λx. identity-type (B x) (app f x) (app g x))

lemma homotopy-form:
  assumes A: A ::h universe i
    and B: ⋀x. x ::h A ⟹ B x ::h universe i
    and f: f ::h pi-type A B
    and g: g ::h pi-type A B
  shows homotopy A B f g ::h universe i
  unfolding homotopy-def
proof (rule pi-form[OF A])
  fix x
  assume x: x ::h A
  have fx: app f x ::h B x by (rule pi-elim[OF f x])
  have gx: app g x ::h B x by (rule pi-elim[OF g x])
  show identity-type (B x) (app f x) (app g x) ::h universe i
    by (rule identity-form[OF B[OF x] fx gx])
qed

definition homotopy-refl ::
```

```
  expr ⇒ (expr ⇒ expr) ⇒ expr ⇒ expr
where
  homotopy-refl A B f ≡
    lambda A (λx. reflexivity (B x) (app f x))

lemma homotopy-refl-type:
  assumes A: A ::_h universe i
    and B: ⋀x. x ::_h A ⟹ B x ::_h universe i
    and f: f ::_h pi-type A B
  shows homotopy-refl A B f ::_h homotopy A B f f
  unfolding homotopy-refl-def homotopy-def
proof (rule pi-intro[OF A])
  fix x
  assume x: x ::_h A
  have fx: app f x ::_h B x by (rule pi-elim[OF f x])
  show reflexivity (B x) (app f x) ::_h
      identity-type (B x) (app f x) (app f x)
    by (rule identity-intro[OF B[OF x] fx])
qed

definition homotopy-concat ::
  expr ⇒ (expr ⇒ expr) ⇒ expr ⇒ expr ⇒
    expr ⇒ expr ⇒ expr ⇒ expr
where
  homotopy-concat A B f g h H K ≡
    lambda A (λx.
      path-concat (B x) (app f x) (app g x) (app h x)
        (app H x) (app K x))

lemma homotopy-concat-type:
  assumes A: A ::_h universe i
    and B: ⋀x. x ::_h A ⟹ B x ::_h universe i
    and f: f ::_h pi-type A B
    and g: g ::_h pi-type A B
    and h: h ::_h pi-type A B
    and H: H ::_h homotopy A B f g
    and K: K ::_h homotopy A B g h
  shows homotopy-concat A B f g h H K ::_h homotopy A B f h
  unfolding homotopy-concat-def homotopy-def
proof (rule pi-intro[OF A])
  fix x
  assume x: x ::_h A
  have fx: app f x ::_h B x by (rule pi-elim[OF f x])
  have gx: app g x ::_h B x by (rule pi-elim[OF g x])
  have hx: app h x ::_h B x by (rule pi-elim[OF h x])
  have Hx: app H x ::_h identity-type (B x) (app f x) (app g x)
    using H unfolding homotopy-def by (rule pi-elim[OF - x])
  have Kx: app K x ::_h identity-type (B x) (app g x) (app h x)
    using K unfolding homotopy-def by (rule pi-elim[OF - x])
```

```
  show path-concat (B x) (app f x) (app g x) (app h x)
      (app H x) (app K x) ::h
      identity-type (B x) (app f x) (app h x)
    by (rule path-concat-type[OF B[OF x] fx gx hx Hx Kx])
qed

definition homotopy-inverse ::
  expr ⇒ (expr ⇒ expr) ⇒ expr ⇒ expr ⇒
    expr ⇒ expr
where
  homotopy-inverse A B f g H ≡
    lambda A (λx.
      path-inverse (B x) (app f x) (app g x) (app H x))

lemma homotopy-inverse-type:
  assumes A: A ::h universe i
    and B: ⋀x. x ::h A ⟹ B x ::h universe i
    and f: f ::h pi-type A B
    and g: g ::h pi-type A B
    and H: H ::h homotopy A B f g
  shows homotopy-inverse A B f g H ::h homotopy A B g f
  unfolding homotopy-inverse-def homotopy-def
proof (rule pi-intro[OF A])
  fix x
  assume x: x ::h A
  have fx: app f x ::h B x by (rule pi-elim[OF f x])
  have gx: app g x ::h B x by (rule pi-elim[OF g x])
  have Hx: app H x ::h
      identity-type (B x) (app f x) (app g x)
    using H unfolding homotopy-def by (rule pi-elim[OF - x])
  show path-inverse (B x) (app f x) (app g x) (app H x)
      ::h identity-type (B x) (app g x) (app f x)
    by (rule path-inverse-type[OF B[OF x] fx gx Hx])
qed

definition homotopy-naturality ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr ⇒
    expr ⇒ expr ⇒ expr ⇒ expr
where
  homotopy-naturality A B f g H x y p ≡
    identity-ind A x
      (λy p. identity-type
        (identity-type B (app f x) (app g y))
        (path-concat B (app f x) (app f y) (app g y)
          (ap A B f x y p) (app H y))
        (path-concat B (app f x) (app g x) (app g y)
          (app H x) (ap A B g x y p)))
      (path-left-unit B (app f x) (app g x) (app H x))
      y p
```

lemma *homotopy-naturality-type*:
assumes A: $A ::_h$ *universe* i
and B: $B ::_h$ *universe* i
and f: $f ::_h$ *arrow* A B
and g: $g ::_h$ *arrow* A B
and H: $H ::_h$ *homotopy* A (λ-. B) f g
and x: $x ::_h A$
and y: $y ::_h A$
and p: $p ::_h$ *identity-type* A x y
shows *homotopy-naturality* A B f g H x y $p ::_h$
identity-type (*identity-type* B (*app* f x) (*app* g y))
(*path-concat* B (*app* f x) (*app* f y) (*app* g y)
(*ap* A B f x y p) (*app* H y))
(*path-concat* B (*app* f x) (*app* g x) (*app* g y)
(*app* H x) (*ap* A B g x y p))
unfolding *homotopy-naturality-def*
proof (*rule identity-elim*[*OF* A x])
fix w q
assume w: $w ::_h A$
assume q: $q ::_h$ *identity-type* A x w
have fx: *app* f $x ::_h B$ **by** (*rule arrow-elim*[*OF* f x])
have fw: *app* f $w ::_h B$ **by** (*rule arrow-elim*[*OF* f w])
have gx: *app* g $x ::_h B$ **by** (*rule arrow-elim*[*OF* g x])
have gw: *app* g $w ::_h B$ **by** (*rule arrow-elim*[*OF* g w])
have Hx: *app* H $x ::_h$ *identity-type* B (*app* f x) (*app* g x)
using H **unfolding** *homotopy-def* **by** (*rule pi-elim*[*OF* - x])
have Hw: *app* H $w ::_h$ *identity-type* B (*app* f w) (*app* g w)
using H **unfolding** *homotopy-def* **by** (*rule pi-elim*[*OF* - w])
have *ap-f*: *ap* A B f x w $q ::_h$
identity-type B (*app* f x) (*app* f w)
by (*rule ap-type*[*OF* A B f x w q])
have *ap-g*: *ap* A B g x w $q ::_h$
identity-type B (*app* g x) (*app* g w)
by (*rule ap-type*[*OF* A B g x w q])
have *lhs*: *path-concat* B (*app* f x) (*app* f w) (*app* g w)
(*ap* A B f x w q) (*app* H w) $::_h$
identity-type B (*app* f x) (*app* g w)
by (*rule path-concat-type*[*OF* B *fx fw gw ap-f Hw*])
have *rhs*: *path-concat* B (*app* f x) (*app* g x) (*app* g w)
(*app* H x) (*ap* A B g x w q) $::_h$
identity-type B (*app* f x) (*app* g w)
by (*rule path-concat-type*[*OF* B *fx gx gw Hx ap-g*])
have *path-space*: *identity-type* B (*app* f x) (*app* g w)
$::_h$ *universe* i
by (*rule identity-form*[*OF* B *fx gw*])
show *identity-type* (*identity-type* B (*app* f x) (*app* g w))
(*path-concat* B (*app* f x) (*app* f w) (*app* g w)
(*ap* A B f x w q) (*app* H w))

```
      (path-concat B (app f x) (app g x) (app g w)
        (app H x) (ap A B g x w q)) ::h universe i
    by (rule identity-form[OF path-space lhs rhs])
next
  have fx: app f x ::h B by (rule arrow-elim[OF f x])
  have gx: app g x ::h B by (rule arrow-elim[OF g x])
  have Hx: app H x ::h identity-type B (app f x) (app g x)
    using H unfolding homotopy-def by (rule pi-elim[OF - x])
  show path-left-unit B (app f x) (app g x) (app H x)
      ::h identity-type
        (identity-type B (app f x) (app g x))
        (path-concat B (app f x) (app f x) (app g x)
          (ap A B f x x (reflexivity A x)) (app H x))
        (path-concat B (app f x) (app g x) (app g x)
          (app H x) (ap A B g x x (reflexivity A x)))
    using path-left-unit-type[OF B fx gx Hx] by simp
next
  show y ::h A by (rule y)
next
  show p ::h identity-type A x y by (rule p)
qed
```

18 Functorial reduction lemmas

```
definition ap-identity ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr
where
  ap-identity A x y p ≡
    identity-ind A x
      (λy p. identity-type (identity-type A x y)
        (ap A A (identity-function A) x y p) p)
      (reflexivity (identity-type A x x) (reflexivity A x))
      y p

lemma ap-identity-type:
  assumes A: A ::h universe i
    and x: x ::h A
    and y: y ::h A
    and p: p ::h identity-type A x y
  shows ap-identity A x y p ::h
    identity-type (identity-type A x y)
      (ap A A (identity-function A) x y p) p
  unfolding ap-identity-def
proof (rule identity-elim[OF A x])
  fix w q
  assume w: w ::h A
  assume q: q ::h identity-type A x w
  have id: identity-function A ::h arrow A A
    by (rule identity-function-type[OF A])
```

```
  have apq: ap A A (identity-function A) x w q ::h
      identity-type A x w
    using ap-type[OF A A id x w q] by simp
  have path-space: identity-type A x w ::h universe i
    by (rule identity-form[OF A x w])
  show identity-type (identity-type A x w)
      (ap A A (identity-function A) x w q) q ::h universe i
    by (rule identity-form[OF path-space apq q])
next
  have path-space: identity-type A x x ::h universe i
    by (rule identity-form[OF A x x])
  have base: reflexivity (identity-type A x x) (reflexivity A x) ::h
      identity-type (identity-type A x x)
        (reflexivity A x) (reflexivity A x)
    by (rule identity-intro[OF path-space identity-intro[OF A x]])
  show reflexivity (identity-type A x x) (reflexivity A x) ::h
      identity-type (identity-type A x x)
        (ap A A (identity-function A) x x (reflexivity A x))
        (reflexivity A x)
    using base by simp
next
  show y ::h A by (rule y)
next
  show p ::h identity-type A x y by (rule p)
qed

definition ap-compose ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr ⇒
    expr ⇒ expr ⇒ expr ⇒ expr
where
  ap-compose A B C g f x y p ≡
    identity-ind A x
      (λy p. identity-type
        (identity-type C (app g (app f x)) (app g (app f y)))
        (ap A C (compose A B C g f) x y p)
        (ap B C g (app f x) (app f y) (ap A B f x y p)))
      (reflexivity
        (identity-type C (app g (app f x)) (app g (app f x)))
        (reflexivity C (app g (app f x))))
      y p

lemma ap-compose-type:
  assumes A: A ::h universe i
    and B: B ::h universe i
    and C: C ::h universe i
    and f: f ::h arrow A B
    and g: g ::h arrow B C
    and x: x ::h A
    and y: y ::h A
```

```
    and p: p ::_h identity-type A x y
  shows ap-compose A B C g f x y p ::_h
    identity-type
      (identity-type C (app g (app f x)) (app g (app f y)))
      (ap A C (compose A B C g f) x y p)
      (ap B C g (app f x) (app f y) (ap A B f x y p))
  unfolding ap-compose-def
proof (rule identity-elim[OF A x])
  fix w q
  assume w: w ::_h A
  assume q: q ::_h identity-type A x w
  have fx: app f x ::_h B by (rule arrow-elim[OF f x])
  have fw: app f w ::_h B by (rule arrow-elim[OF f w])
  have gfx: app g (app f x) ::_h C by (rule arrow-elim[OF g fx])
  have gfw: app g (app f w) ::_h C by (rule arrow-elim[OF g fw])
  have gf: compose A B C g f ::_h arrow A C
    by (rule compose-type[OF A f g])
  have ap-gf: ap A C (compose A B C g f) x w q ::_h
      identity-type C (app g (app f x)) (app g (app f w))
    using ap-type[OF A C gf x w q] by simp
  have apf: ap A B f x w q ::_h
      identity-type B (app f x) (app f w)
    by (rule ap-type[OF A B f x w q])
  have ap-g: ap B C g (app f x) (app f w) (ap A B f x w q) ::_h
      identity-type C (app g (app f x)) (app g (app f w))
    by (rule ap-type[OF B C g fx fw apf])
  have path-space:
      identity-type C (app g (app f x)) (app g (app f w))
        ::_h universe i
    by (rule identity-form[OF C gfx gfw])
  show identity-type
      (identity-type C (app g (app f x)) (app g (app f w)))
      (ap A C (compose A B C g f) x w q)
      (ap B C g (app f x) (app f w) (ap A B f x w q))
      ::_h universe i
    by (rule identity-form[OF path-space ap-gf ap-g])
next
  have fx: app f x ::_h B by (rule arrow-elim[OF f x])
  have gfx: app g (app f x) ::_h C by (rule arrow-elim[OF g fx])
  have path-space:
      identity-type C (app g (app f x)) (app g (app f x))
        ::_h universe i
    by (rule identity-form[OF C gfx gfx])
  have base: reflexivity
      (identity-type C (app g (app f x)) (app g (app f x)))
      (reflexivity C (app g (app f x))) ::_h
      identity-type
        (identity-type C (app g (app f x)) (app g (app f x)))
        (reflexivity C (app g (app f x)))
```

```
      (reflexivity C (app g (app f x)))
    by (rule identity-intro[OF path-space identity-intro[OF C gfx]])
  show reflexivity
    (identity-type C (app g (app f x)) (app g (app f x)))
    (reflexivity C (app g (app f x))) ::h
    identity-type
      (identity-type C (app g (app f x)) (app g (app f x)))
      (ap A C (compose A B C g f) x x (reflexivity A x))
      (ap B C g (app f x) (app f x)
        (ap A B f x x (reflexivity A x)))
    using base by simp
next
  show y ::h A by (rule y)
next
  show p ::h identity-type A x y by (rule p)
qed

definition transport-identity-left ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr ⇒
    expr ⇒ expr ⇒ expr ⇒ expr
where
  transport-identity-left A B f c x y p w ≡
    identity-ind A x
      (λy p. identity-type (identity-type B (app f y) c)
        (transport A (λa. identity-type B (app f a) c) x y p w)
        (path-concat B (app f y) (app f x) c
          (path-inverse B (app f x) (app f y) (ap A B f x y p)) w))
      (path-inverse (identity-type B (app f x) c)
        (path-concat B (app f x) (app f x) c
          (reflexivity B (app f x)) w)
        w
        (path-left-unit B (app f x) c w))
      y p

lemma transport-identity-left-type:
  assumes A: A ::h universe i
    and B: B ::h universe i
    and f: f ::h arrow A B
    and c: c ::h B
    and x: x ::h A
    and y: y ::h A
    and p: p ::h identity-type A x y
    and w: w ::h identity-type B (app f x) c
  shows transport-identity-left A B f c x y p w ::h
    identity-type (identity-type B (app f y) c)
      (transport A (λa. identity-type B (app f a) c) x y p w)
      (path-concat B (app f y) (app f x) c
        (path-inverse B (app f x) (app f y) (ap A B f x y p)) w)
  unfolding transport-identity-left-def
```

proof (*rule identity-elim*[*OF A x*])
  fix *v r*
  assume *v*: $v ::_h A$
  assume *r*: $r ::_h$ *identity-type A x v*
  have *fx*: *app f x* $::_h B$ **by** (*rule arrow-elim*[*OF f x*])
  have *fv*: *app f v* $::_h B$ **by** (*rule arrow-elim*[*OF f v*])
  have *transported*:
    transport A (*λa. identity-type B* (*app f a*) *c*) *x v r w* $::_h$
      identity-type B (*app f v*) *c*
  proof (*rule transport-type*[*OF A*])
    fix *a*
    assume *a*: $a ::_h A$
    have *fa*: *app f a* $::_h B$ **by** (*rule arrow-elim*[*OF f a*])
    show *identity-type B* (*app f a*) *c* $::_h$ *universe i*
      by (*rule identity-form*[*OF B fa c*])
  next
    show $x ::_h A$ **by** (*rule x*)
  next
    show $v ::_h A$ **by** (*rule v*)
  next
    show $r ::_h$ *identity-type A x v* **by** (*rule r*)
  next
    show $w ::_h$ *identity-type B* (*app f x*) *c* **by** (*rule w*)
  qed
  have *apr*: *ap A B f x v r* $::_h$ *identity-type B* (*app f x*) (*app f v*)
    by (*rule ap-type*[*OF A B f x v r*])
  have *inv*: *path-inverse B* (*app f x*) (*app f v*) (*ap A B f x v r*) $::_h$
      identity-type B (*app f v*) (*app f x*)
    by (*rule path-inverse-type*[*OF B fx fv apr*])
  have *composed*: *path-concat B* (*app f v*) (*app f x*) *c*
      (*path-inverse B* (*app f x*) (*app f v*) (*ap A B f x v r*)) *w* $::_h$
      identity-type B (*app f v*) *c*
    by (*rule path-concat-type*[*OF B fv fx c inv w*])
  have *path-space*: *identity-type B* (*app f v*) *c* $::_h$ *universe i*
    by (*rule identity-form*[*OF B fv c*])
  show *identity-type* (*identity-type B* (*app f v*) *c*)
      (*transport A* (*λa. identity-type B* (*app f a*) *c*) *x v r w*)
      (*path-concat B* (*app f v*) (*app f x*) *c*
        (*path-inverse B* (*app f x*) (*app f v*) (*ap A B f x v r*)) *w*)
      $::_h$ *universe i*
    by (*rule identity-form*[*OF path-space transported composed*])
next
  have *fx*: *app f x* $::_h B$ **by** (*rule arrow-elim*[*OF f x*])
  have *path-space*: *identity-type B* (*app f x*) *c* $::_h$ *universe i*
    by (*rule identity-form*[*OF B fx c*])
  have *composed*: *path-concat B* (*app f x*) (*app f x*) *c*
      (*reflexivity B* (*app f x*)) *w* $::_h$
      identity-type B (*app f x*) *c*
    by (*rule path-concat-type*[*OF B fx fx c identity-intro*[*OF B fx*] *w*])

```
  have base: path-inverse (identity-type B (app f x) c)
      (path-concat B (app f x) (app f x) c
        (reflexivity B (app f x)) w)
      w
      (path-left-unit B (app f x) c w) ::_h
      identity-type (identity-type B (app f x) c)
        w
        (path-concat B (app f x) (app f x) c
          (reflexivity B (app f x)) w)
    by (rule path-inverse-type[OF path-space composed w
          path-left-unit-type[OF B fx c w]])
  show path-inverse (identity-type B (app f x) c)
      (path-concat B (app f x) (app f x) c
        (reflexivity B (app f x)) w)
      w
      (path-left-unit B (app f x) c w) ::_h
      identity-type (identity-type B (app f x) c)
        (transport A (λa. identity-type B (app f a) c) x x
          (reflexivity A x) w)
        (path-concat B (app f x) (app f x) c
          (path-inverse B (app f x) (app f x)
            (ap A B f x x (reflexivity A x))) w)
    using base by simp
next
  show y ::_h A by (rule y)
next
  show p ::_h identity-type A x y by (rule p)
qed
```

19 Whiskering and cancellation of paths

```
definition path-inverse-inverse ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr
where
  path-inverse-inverse A x y p ≡
    identity-ind A x
      (λy p. identity-type (identity-type A x y)
        (path-inverse A y x (path-inverse A x y p)) p)
      (reflexivity (identity-type A x x) (reflexivity A x))
      y p

lemma path-inverse-inverse-type:
  assumes A: A ::_h universe i
    and x: x ::_h A
    and y: y ::_h A
    and p: p ::_h identity-type A x y
  shows path-inverse-inverse A x y p ::_h
    identity-type (identity-type A x y)
      (path-inverse A y x (path-inverse A x y p)) p
```

```
  unfolding path-inverse-inverse-def
proof (rule identity-elim[OF A x])
  fix w q
  assume w: w ::_h A
  assume q: q ::_h identity-type A x w
  have inv: path-inverse A x w q ::_h identity-type A w x
    by (rule path-inverse-type[OF A x w q])
  have double: path-inverse A w x (path-inverse A x w q) ::_h
      identity-type A x w
    by (rule path-inverse-type[OF A w x inv])
  have path-space: identity-type A x w ::_h universe i
    by (rule identity-form[OF A x w])
  show identity-type (identity-type A x w)
      (path-inverse A w x (path-inverse A x w q)) q ::_h universe i
    by (rule identity-form[OF path-space double q])
next
  have path-space: identity-type A x x ::_h universe i
    by (rule identity-form[OF A x x])
  have base: reflexivity (identity-type A x x) (reflexivity A x) ::_h
      identity-type (identity-type A x x)
        (reflexivity A x) (reflexivity A x)
    by (rule identity-intro[OF path-space identity-intro[OF A x]])
  show reflexivity (identity-type A x x) (reflexivity A x) ::_h
      identity-type (identity-type A x x)
        (path-inverse A x x (path-inverse A x x (reflexivity A x)))
        (reflexivity A x)
    using base by simp
next
  show y ::_h A by (rule y)
next
  show p ::_h identity-type A x y by (rule p)
qed

definition path-whisker-left ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr ⇒
    expr ⇒ expr ⇒ expr ⇒ expr
where
  path-whisker-left A x y z r q q′ m ≡
    identity-ind (identity-type A y z) q
      (λq′ m. identity-type (identity-type A x z)
        (path-concat A x y z r q) (path-concat A x y z r q′))
      (reflexivity (identity-type A x z) (path-concat A x y z r q))
      q′ m

lemma path-whisker-left-type:
  assumes A: A ::_h universe i
    and x: x ::_h A
    and y: y ::_h A
    and z: z ::_h A
```

```
    and r: r ::_h identity-type A x y
    and q: q ::_h identity-type A y z
    and q': q' ::_h identity-type A y z
    and m: m ::_h identity-type (identity-type A y z) q q'
  shows path-whisker-left A x y z r q q' m ::_h
    identity-type (identity-type A x z)
      (path-concat A x y z r q) (path-concat A x y z r q')
  unfolding path-whisker-left-def
proof (rule identity-elim[OF identity-form[OF A y z] q])
  fix w mm
  assume w: w ::_h identity-type A y z
  assume mm: mm ::_h identity-type (identity-type A y z) q w
  have rq: path-concat A x y z r q ::_h identity-type A x z
    by (rule path-concat-type[OF A x y z r q])
  have rw: path-concat A x y z r w ::_h identity-type A x z
    by (rule path-concat-type[OF A x y z r w])
  have path-space: identity-type A x z ::_h universe i
    by (rule identity-form[OF A x z])
  show identity-type (identity-type A x z)
      (path-concat A x y z r q) (path-concat A x y z r w) ::_h universe i
    by (rule identity-form[OF path-space rq rw])
next
  have rq: path-concat A x y z r q ::_h identity-type A x z
    by (rule path-concat-type[OF A x y z r q])
  have path-space: identity-type A x z ::_h universe i
    by (rule identity-form[OF A x z])
  show reflexivity (identity-type A x z) (path-concat A x y z r q) ::_h
      identity-type (identity-type A x z)
        (path-concat A x y z r q) (path-concat A x y z r q)
    by (rule identity-intro[OF path-space rq])
next
  show q' ::_h identity-type A y z by (rule q')
next
  show m ::_h identity-type (identity-type A y z) q q' by (rule m)
qed

definition path-right-cancel ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr ⇒
    expr ⇒ expr ⇒ expr ⇒ expr
where
  path-right-cancel A x y z alpha beta r s ≡
    app
      (identity-ind A y
        (λz r. arrow
          (identity-type (identity-type A x z)
            (path-concat A x y z alpha r)
            (path-concat A x y z beta r))
          (identity-type (identity-type A x y) alpha beta))
        (identity-function
```

```
      (identity-type (identity-type A x y) alpha beta))
    z r)
  s

lemma path-right-cancel-type:
  assumes A: A ::_h universe i
    and x: x ::_h A
    and y: y ::_h A
    and z: z ::_h A
    and alpha: alpha ::_h identity-type A x y
    and beta: beta ::_h identity-type A x y
    and r: r ::_h identity-type A y z
    and s: s ::_h identity-type (identity-type A x z)
          (path-concat A x y z alpha r) (path-concat A x y z beta r)
  shows path-right-cancel A x y z alpha beta r s ::_h
    identity-type (identity-type A x y) alpha beta
proof −
  have path-space: identity-type A x y ::_h universe i
    by (rule identity-form[OF A x y])
  have goal-space:
      identity-type (identity-type A x y) alpha beta ::_h universe i
    by (rule identity-form[OF path-space alpha beta])
  have base: identity-function
      (identity-type (identity-type A x y) alpha beta) ::_h
      arrow (identity-type (identity-type A x y) alpha beta)
          (identity-type (identity-type A x y) alpha beta)
    by (rule identity-function-type[OF goal-space])
  have inducted: identity-ind A y
      (λz r. arrow
        (identity-type (identity-type A x z)
          (path-concat A x y z alpha r)
          (path-concat A x y z beta r))
        (identity-type (identity-type A x y) alpha beta))
      (identity-function
        (identity-type (identity-type A x y) alpha beta))
      z r ::_h
      arrow
        (identity-type (identity-type A x z)
          (path-concat A x y z alpha r)
          (path-concat A x y z beta r))
        (identity-type (identity-type A x y) alpha beta)
  proof (rule identity-elim[OF A y])
    fix w rr
    assume w: w ::_h A
    assume rr: rr ::_h identity-type A y w
    have ar: path-concat A x y w alpha rr ::_h identity-type A x w
      by (rule path-concat-type[OF A x y w alpha rr])
    have br: path-concat A x y w beta rr ::_h identity-type A x w
      by (rule path-concat-type[OF A x y w beta rr])
```

```
    have pspace: identity-type A x w ::_h universe i
      by (rule identity-form[OF A x w])
    have dom: identity-type (identity-type A x w)
        (path-concat A x y w alpha rr)
        (path-concat A x y w beta rr) ::_h universe i
      by (rule identity-form[OF pspace ar br])
    show arrow
        (identity-type (identity-type A x w)
          (path-concat A x y w alpha rr)
          (path-concat A x y w beta rr))
        (identity-type (identity-type A x y) alpha beta) ::_h universe i
      by (rule arrow-form[OF dom goal-space])
  next
    show identity-function
        (identity-type (identity-type A x y) alpha beta) ::_h
        arrow
          (identity-type (identity-type A x y)
            (path-concat A x y y alpha (reflexivity A y))
            (path-concat A x y y beta (reflexivity A y)))
          (identity-type (identity-type A x y) alpha beta)
      using base by simp
  next
    show z ::_h A by (rule z)
  next
    show r ::_h identity-type A y z by (rule r)
  qed
  show path-right-cancel A x y z alpha beta r s ::_h
      identity-type (identity-type A x y) alpha beta
    unfolding path-right-cancel-def
    by (rule arrow-elim[OF inducted s])
qed
```

20 The self-action of a homotopy to the identity

```
definition homotopy-self ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr
where
  homotopy-self A u H a ≡
    path-right-cancel A (app u (app u a)) (app u a) a
      (ap A A u (app u a) a (app H a))
      (app H (app u a))
      (app H a)
      (path-concat (identity-type A (app u (app u a)) a)
        (path-concat A (app u (app u a)) (app u a) a
          (ap A A u (app u a) a (app H a)) (app H a))
        (path-concat A (app u (app u a)) (app u a) a
          (app H (app u a))
          (ap A A (identity-function A) (app u a) a (app H a)))
        (path-concat A (app u (app u a)) (app u a) a
```

```
    (app H (app u a)) (app H a))
  (homotopy-naturality A A u (identity-function A) H
    (app u a) a (app H a))
  (path-whisker-left A (app u (app u a)) (app u a) a
    (app H (app u a))
    (ap A A (identity-function A) (app u a) a (app H a))
    (app H a)
    (ap-identity A (app u a) a (app H a))))

lemma homotopy-self-type:
  assumes A: A ::_h universe i
    and u: u ::_h arrow A A
    and H: H ::_h homotopy A (λ-. A) u (identity-function A)
    and a: a ::_h A
  shows homotopy-self A u H a ::_h
    identity-type (identity-type A (app u (app u a)) (app u a))
      (ap A A u (app u a) a (app H a))
      (app H (app u a))
proof −
  have id: identity-function A ::_h arrow A A
    by (rule identity-function-type[OF A])
  have ua: app u a ::_h A by (rule arrow-elim[OF u a])
  have uua: app u (app u a) ::_h A by (rule arrow-elim[OF u ua])
  have Hpi: H ::_h
      pi-type A (λx. identity-type A (app u x)
        (app (identity-function A) x))
    using H unfolding homotopy-def .
  have Ha: app H a ::_h identity-type A (app u a) a
    using pi-elim[OF Hpi a] by simp
  have Hua: app H (app u a) ::_h
      identity-type A (app u (app u a)) (app u a)
    using pi-elim[OF Hpi ua] by simp
  have apuHa: ap A A u (app u a) a (app H a) ::_h
      identity-type A (app u (app u a)) (app u a)
    by (rule ap-type[OF A A u ua a Ha])
  have apidHa: ap A A (identity-function A) (app u a) a (app H a) ::_h
      identity-type A (app u a) a
    using ap-type[OF A A id ua a Ha] by simp
  have NAT: homotopy-naturality A A u (identity-function A) H
      (app u a) a (app H a) ::_h
    identity-type (identity-type A (app u (app u a)) a)
      (path-concat A (app u (app u a)) (app u a) a
        (ap A A u (app u a) a (app H a)) (app H a))
      (path-concat A (app u (app u a)) (app u a) a
        (app H (app u a))
        (ap A A (identity-function A) (app u a) a (app H a)))
    using homotopy-naturality-type[OF A A u id H ua a Ha] by simp
  have AI: ap-identity A (app u a) a (app H a) ::_h
    identity-type (identity-type A (app u a) a)
```

(*ap A A* (*identity-function A*) (*app u a*) *a* (*app H a*))
(*app H a*)
by (*rule ap-identity-type*[*OF A ua a Ha*])
have *WH*: *path-whisker-left A* (*app u* (*app u a*)) (*app u a*) *a*
(*app H* (*app u a*))
(*ap A A* (*identity-function A*) (*app u a*) *a* (*app H a*))
(*app H a*)
(*ap-identity A* (*app u a*) *a* (*app H a*)) ::$_h$
identity-type (*identity-type A* (*app u* (*app u a*)) *a*)
(*path-concat A* (*app u* (*app u a*)) (*app u a*) *a*
(*app H* (*app u a*))
(*ap A A* (*identity-function A*) (*app u a*) *a* (*app H a*)))
(*path-concat A* (*app u* (*app u a*)) (*app u a*) *a*
(*app H* (*app u a*)) (*app H a*))
by (*rule path-whisker-left-type*[*OF A uua ua a Hua apidHa Ha AI*])
have *ps*: *identity-type A* (*app u* (*app u a*)) *a* ::$_h$ *universe i*
by (*rule identity-form*[*OF A uua a*])
have *P1*: *path-concat A* (*app u* (*app u a*)) (*app u a*) *a*
(*ap A A u* (*app u a*) *a* (*app H a*)) (*app H a*) ::$_h$
identity-type A (*app u* (*app u a*)) *a*
by (*rule path-concat-type*[*OF A uua ua a apuHa Ha*])
have *P2*: *path-concat A* (*app u* (*app u a*)) (*app u a*) *a*
(*app H* (*app u a*))
(*ap A A* (*identity-function A*) (*app u a*) *a* (*app H a*)) ::$_h$
identity-type A (*app u* (*app u a*)) *a*
by (*rule path-concat-type*[*OF A uua ua a Hua apidHa*])
have *P3*: *path-concat A* (*app u* (*app u a*)) (*app u a*) *a*
(*app H* (*app u a*)) (*app H a*) ::$_h$
identity-type A (*app u* (*app u a*)) *a*
by (*rule path-concat-type*[*OF A uua ua a Hua Ha*])
have *NAT2*: *path-concat* (*identity-type A* (*app u* (*app u a*)) *a*)
(*path-concat A* (*app u* (*app u a*)) (*app u a*) *a*
(*ap A A u* (*app u a*) *a* (*app H a*)) (*app H a*))
(*path-concat A* (*app u* (*app u a*)) (*app u a*) *a*
(*app H* (*app u a*))
(*ap A A* (*identity-function A*) (*app u a*) *a* (*app H a*)))
(*path-concat A* (*app u* (*app u a*)) (*app u a*) *a*
(*app H* (*app u a*)) (*app H a*))
(*homotopy-naturality A A u* (*identity-function A*) *H*
(*app u a*) *a* (*app H a*))
(*path-whisker-left A* (*app u* (*app u a*)) (*app u a*) *a*
(*app H* (*app u a*))
(*ap A A* (*identity-function A*) (*app u a*) *a* (*app H a*))
(*app H a*)
(*ap-identity A* (*app u a*) *a* (*app H a*))) ::$_h$
identity-type (*identity-type A* (*app u* (*app u a*)) *a*)
(*path-concat A* (*app u* (*app u a*)) (*app u a*) *a*
(*ap A A u* (*app u a*) *a* (*app H a*)) (*app H a*))
(*path-concat A* (*app u* (*app u a*)) (*app u a*) *a*

```
        (app H (app u a)) (app H a))
    by (rule path-concat-type[OF ps P1 P2 P3 NAT WH])
  show homotopy-self A u H a ::h
      identity-type (identity-type A (app u (app u a)) (app u a))
        (ap A A u (app u a) a (app H a))
        (app H (app u a))
    unfolding homotopy-self-def
    by (rule path-right-cancel-type[OF A uua ua a apuHa Hua Ha NAT2])
qed

definition ap-congruence ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr ⇒
    expr ⇒ expr ⇒ expr ⇒ expr
where
  ap-congruence A B f x y q q′ m ≡
    ap (identity-type A x y) (identity-type B (app f x) (app f y))
      (lambda (identity-type A x y) (λt. ap A B f x y t))
      q q′ m

lemma ap-congruence-type:
  assumes A: A ::h universe i
    and B: B ::h universe i
    and f: f ::h arrow A B
    and x: x ::h A
    and y: y ::h A
    and q: q ::h identity-type A x y
    and q′: q′ ::h identity-type A x y
    and m: m ::h identity-type (identity-type A x y) q q′
  shows ap-congruence A B f x y q q′ m ::h
    identity-type (identity-type B (app f x) (app f y))
      (ap A B f x y q) (ap A B f x y q′)
proof −
  have fx: app f x ::h B by (rule arrow-elim[OF f x])
  have fy: app f y ::h B by (rule arrow-elim[OF f y])
  have pxA: identity-type A x y ::h universe i
    by (rule identity-form[OF A x y])
  have pxB: identity-type B (app f x) (app f y) ::h universe i
    by (rule identity-form[OF B fx fy])
  have lam: lambda (identity-type A x y) (λt. ap A B f x y t) ::h
      arrow (identity-type A x y)
        (identity-type B (app f x) (app f y))
  proof (rule arrow-intro[OF pxA])
    fix t
    assume t: t ::h identity-type A x y
    show ap A B f x y t ::h identity-type B (app f x) (app f y)
      by (rule ap-type[OF A B f x y t])
  qed
  have ap (identity-type A x y) (identity-type B (app f x) (app f y))
      (lambda (identity-type A x y) (λt. ap A B f x y t))
```

```
    q q' m ::_h
    identity-type (identity-type B (app f x) (app f y))
      (app (lambda (identity-type A x y) (λt. ap A B f x y t)) q)
      (app (lambda (identity-type A x y) (λt. ap A B f x y t)) q')
  by (rule ap-type[OF pxA pxB lam q q' m])
then show ap-congruence A B f x y q q' m ::_h
    identity-type (identity-type B (app f x) (app f y))
      (ap A B f x y q) (ap A B f x y q')
  unfolding ap-congruence-def by simp
qed

definition path-whisker-right ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr ⇒
    expr ⇒ expr ⇒ expr ⇒ expr
where
  path-whisker-right A x y z p p' r m ≡
    identity-ind (identity-type A x y) p
      (λp' m. identity-type (identity-type A x z)
        (path-concat A x y z p r) (path-concat A x y z p' r))
      (reflexivity (identity-type A x z) (path-concat A x y z p r))
      p' m

lemma path-whisker-right-type:
  assumes A: A ::_h universe i
    and x: x ::_h A
    and y: y ::_h A
    and z: z ::_h A
    and p: p ::_h identity-type A x y
    and p': p' ::_h identity-type A x y
    and r: r ::_h identity-type A y z
    and m: m ::_h identity-type (identity-type A x y) p p'
  shows path-whisker-right A x y z p p' r m ::_h
    identity-type (identity-type A x z)
      (path-concat A x y z p r) (path-concat A x y z p' r)
  unfolding path-whisker-right-def
proof (rule identity-elim[OF identity-form[OF A x y] p])
  fix w mm
  assume w: w ::_h identity-type A x y
  assume mm: mm ::_h identity-type (identity-type A x y) p w
  have pr: path-concat A x y z p r ::_h identity-type A x z
    by (rule path-concat-type[OF A x y z p r])
  have wr: path-concat A x y z w r ::_h identity-type A x z
    by (rule path-concat-type[OF A x y z w r])
  have path-space: identity-type A x z ::_h universe i
    by (rule identity-form[OF A x z])
  show identity-type (identity-type A x z)
      (path-concat A x y z p r) (path-concat A x y z w r) ::_h universe i
    by (rule identity-form[OF path-space pr wr])
next
```

```
  have pr: path-concat A x y z p r ::h identity-type A x z
    by (rule path-concat-type[OF A x y z p r])
  have path-space: identity-type A x z ::h universe i
    by (rule identity-form[OF A x z])
  show reflexivity (identity-type A x z) (path-concat A x y z p r) ::h
      identity-type (identity-type A x z)
        (path-concat A x y z p r) (path-concat A x y z p r)
    by (rule identity-intro[OF path-space pr])
next
  show p' ::h identity-type A x y by (rule p')
next
  show m ::h identity-type (identity-type A x y) p p' by (rule m)
qed

definition path-inverse-concat-left ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr
where
  path-inverse-concat-left A x y z q s ≡
    path-concat (identity-type A y z)
      (path-concat A y x z (path-inverse A x y q)
        (path-concat A x y z q s))
      (path-concat A y y z
        (path-concat A y x y (path-inverse A x y q) q) s)
      s
      (path-inverse (identity-type A y z)
        (path-concat A y y z
          (path-concat A y x y (path-inverse A x y q) q) s)
        (path-concat A y x z (path-inverse A x y q)
          (path-concat A x y z q s))
        (path-assoc A y x y z (path-inverse A x y q) q s))
      (path-concat (identity-type A y z)
        (path-concat A y y z
          (path-concat A y x y (path-inverse A x y q) q) s)
        (path-concat A y y z (reflexivity A y) s)
        s
        (path-whisker-right A y y z
          (path-concat A y x y (path-inverse A x y q) q)
          (reflexivity A y) s
          (path-inverse-left A x y q))
        (path-left-unit A y z s))

lemma path-inverse-concat-left-type:
  assumes A: A ::h universe i
    and x: x ::h A
    and y: y ::h A
    and z: z ::h A
    and q: q ::h identity-type A x y
    and s: s ::h identity-type A y z
  shows path-inverse-concat-left A x y z q s ::h
```

```
    identity-type (identity-type A y z)
      (path-concat A y x z (path-inverse A x y q)
        (path-concat A x y z q s)) s
proof −
  have qinv: path-inverse A x y q ::_h identity-type A y x
    by (rule path-inverse-type[OF A x y q])
  have qs: path-concat A x y z q s ::_h identity-type A x z
    by (rule path-concat-type[OF A x y z q s])
  have qinv-qs: path-concat A y x z (path-inverse A x y q)
      (path-concat A x y z q s) ::_h identity-type A y z
    by (rule path-concat-type[OF A y x z qinv qs])
  have qinv-q: path-concat A y x y (path-inverse A x y q) q ::_h
      identity-type A y y
    by (rule path-concat-type[OF A y x y qinv q])
  have qinv-q-s: path-concat A y y z
      (path-concat A y x y (path-inverse A x y q) q) s ::_h
      identity-type A y z
    by (rule path-concat-type[OF A y y z qinv-q s])
  have refl-s: path-concat A y y z (reflexivity A y) s ::_h
      identity-type A y z
    by (rule path-concat-type[OF A y y z identity-intro[OF A y] s])
  have path-space: identity-type A y z ::_h universe i
    by (rule identity-form[OF A y z])
  have assoc: path-assoc A y x y z (path-inverse A x y q) q s ::_h
      identity-type (identity-type A y z)
        (path-concat A y y z
          (path-concat A y x y (path-inverse A x y q) q) s)
        (path-concat A y x z (path-inverse A x y q)
          (path-concat A x y z q s))
    by (rule path-assoc-type[OF A y x y z qinv q s])
  have assoc-inv: path-inverse (identity-type A y z)
      (path-concat A y y z
        (path-concat A y x y (path-inverse A x y q) q) s)
      (path-concat A y x z (path-inverse A x y q)
        (path-concat A x y z q s))
      (path-assoc A y x y z (path-inverse A x y q) q s) ::_h
      identity-type (identity-type A y z)
        (path-concat A y x z (path-inverse A x y q)
          (path-concat A x y z q s))
        (path-concat A y y z
          (path-concat A y x y (path-inverse A x y q) q) s)
    by (rule path-inverse-type[OF path-space qinv-q-s qinv-qs assoc])
  have inv-left: path-inverse-left A x y q ::_h
      identity-type (identity-type A y y)
        (path-concat A y x y (path-inverse A x y q) q)
        (reflexivity A y)
    by (rule path-inverse-left-type[OF A x y q])
  have whisk: path-whisker-right A y y z
      (path-concat A y x y (path-inverse A x y q) q)
```

```
    (reflexivity A y) s
    (path-inverse-left A x y q) ::_h
    identity-type (identity-type A y z)
      (path-concat A y y z
        (path-concat A y x y (path-inverse A x y q) q) s)
      (path-concat A y y z (reflexivity A y) s)
  by (rule path-whisker-right-type[OF A y y z qinv-q
       identity-intro[OF A y] s inv-left])
have left-unit: path-left-unit A y z s ::_h
    identity-type (identity-type A y z)
      (path-concat A y y z (reflexivity A y) s) s
  by (rule path-left-unit-type[OF A y z s])
have tail: path-concat (identity-type A y z)
    (path-concat A y y z
      (path-concat A y x y (path-inverse A x y q) q) s)
    (path-concat A y y z (reflexivity A y) s)
    s
    (path-whisker-right A y y z
      (path-concat A y x y (path-inverse A x y q) q)
      (reflexivity A y) s
      (path-inverse-left A x y q))
    (path-left-unit A y z s) ::_h
    identity-type (identity-type A y z)
      (path-concat A y y z
        (path-concat A y x y (path-inverse A x y q) q) s) s
  by (rule path-concat-type[OF path-space qinv-q-s refl-s s whisk left-unit])
show path-inverse-concat-left A x y z q s ::_h
    identity-type (identity-type A y z)
      (path-concat A y x z (path-inverse A x y q)
        (path-concat A x y z q s)) s
  unfolding path-inverse-concat-left-def
  by (rule path-concat-type[OF path-space qinv-qs qinv-q-s s
       assoc-inv tail])
qed

definition path-concat-inverse-right ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr
where
  path-concat-inverse-right A x y z s q ≡
    path-concat (identity-type A x z)
      (path-concat A x y z
        (path-concat A x z y s (path-inverse A y z q)) q)
      (path-concat A x z z s
        (path-concat A z y z (path-inverse A y z q) q))
      s
      (path-assoc A x z y z s (path-inverse A y z q) q)
      (path-whisker-left A x z z s
        (path-concat A z y z (path-inverse A y z q) q)
        (reflexivity A z)
```

```
      (path-inverse-left A y z q))

lemma path-concat-inverse-right-type:
  assumes A: A ::h universe i
    and x: x ::h A
    and y: y ::h A
    and z: z ::h A
    and s: s ::h identity-type A x z
    and q: q ::h identity-type A y z
  shows path-concat-inverse-right A x y z s q ::h
    identity-type (identity-type A x z)
      (path-concat A x y z
        (path-concat A x z y s (path-inverse A y z q)) q) s
proof −
  have qinv: path-inverse A y z q ::h identity-type A z y
    by (rule path-inverse-type[OF A y z q])
  have sqinv: path-concat A x z y s (path-inverse A y z q) ::h
      identity-type A x y
    by (rule path-concat-type[OF A x z y s qinv])
  have sqinv-q: path-concat A x y z
      (path-concat A x z y s (path-inverse A y z q)) q ::h
      identity-type A x z
    by (rule path-concat-type[OF A x y z sqinv q])
  have qinv-q: path-concat A z y z (path-inverse A y z q) q ::h
      identity-type A z z
    by (rule path-concat-type[OF A z y z qinv q])
  have s-qinvq: path-concat A x z z s
      (path-concat A z y z (path-inverse A y z q) q) ::h
      identity-type A x z
    by (rule path-concat-type[OF A x z z s qinv-q])
  have path-space: identity-type A x z ::h universe i
    by (rule identity-form[OF A x z])
  have assoc: path-assoc A x z y z s (path-inverse A y z q) q ::h
      identity-type (identity-type A x z)
        (path-concat A x y z
          (path-concat A x z y s (path-inverse A y z q)) q)
        (path-concat A x z z s
          (path-concat A z y z (path-inverse A y z q) q))
    by (rule path-assoc-type[OF A x z y z s qinv q])
  have inv-left: path-inverse-left A y z q ::h
      identity-type (identity-type A z z)
        (path-concat A z y z (path-inverse A y z q) q)
        (reflexivity A z)
    by (rule path-inverse-left-type[OF A y z q])
  have whisk: path-whisker-left A x z z s
      (path-concat A z y z (path-inverse A y z q) q)
      (reflexivity A z)
      (path-inverse-left A y z q) ::h
      identity-type (identity-type A x z)
```

```
      (path-concat A x z z s
        (path-concat A z y z (path-inverse A y z q) q)) s
    using path-whisker-left-type[OF A x z z s qinv-q
      identity-intro[OF A z] inv-left] by simp
  show path-concat-inverse-right A x y z s q ::h
      identity-type (identity-type A x z)
        (path-concat A x y z
          (path-concat A x z y s (path-inverse A y z q)) q) s
    unfolding path-concat-inverse-right-def
    by (rule path-concat-type[OF path-space sqinv-q s-qinvq s assoc whisk])
qed

end
theory HoTT-Equivalence
  imports HoTT-Identity
begin
```

21 Contractibility and fibers

```
definition contractible :: expr ⇒ expr where
  contractible A ≡
    sigma-type A
      (λcenter. pi-type A
        (λx. identity-type A center x))

lemma contractible-form:
  assumes A: A ::h universe i
  shows contractible A ::h universe i
  unfolding contractible-def
proof (rule sigma-form[OF A])
  fix center
  assume center: center ::h A
  show pi-type A (λx. identity-type A center x) ::h universe i
  proof (rule pi-form[OF A])
    fix x
    assume x: x ::h A
    show identity-type A center x ::h universe i
      by (rule identity-form[OF A center x])
  qed
qed

definition fiber ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr
where
  fiber A B f b ≡
    sigma-type A (λa. identity-type B (app f a) b)

lemma fiber-form:
  assumes A: A ::h universe i
```

```
    and B: B ::_h universe i
    and f: f ::_h arrow A B
    and b: b ::_h B
  shows fiber A B f b ::_h universe i
  unfolding fiber-def
proof (rule sigma-form[OF A])
  fix a
  assume a: a ::_h A
  have fa: app f a ::_h B by (rule arrow-elim[OF f a])
  show identity-type B (app f a) b ::_h universe i
    by (rule identity-form[OF B fa b])
qed

definition contractible-map ::
  expr ⇒ expr ⇒ expr ⇒ expr
where
  contractible-map A B f ≡
    pi-type B (λb. contractible (fiber A B f b))

lemma contractible-map-form:
  assumes A: A ::_h universe i
    and B: B ::_h universe i
    and f: f ::_h arrow A B
  shows contractible-map A B f ::_h universe i
  unfolding contractible-map-def
proof (rule pi-form[OF B])
  fix b
  assume b: b ::_h B
  have fib: fiber A B f b ::_h universe i
    by (rule fiber-form[OF A B f b])
  show contractible (fiber A B f b) ::_h universe i
    by (rule contractible-form[OF fib])
qed
```

22 Bi-invertible maps and equivalences

```
definition left-inverse ::
  expr ⇒ expr ⇒ expr ⇒ expr
where
  left-inverse A B f ≡
    sigma-type (arrow B A)
      (λg. homotopy A (λ-. A)
        (compose A B A g f) (identity-function A))

definition right-inverse ::
  expr ⇒ expr ⇒ expr ⇒ expr
where
  right-inverse A B f ≡
    sigma-type (arrow B A)
```

```
    (λg. homotopy B (λ-. B)
      (compose B A B f g) (identity-function B))

definition biinvertible ::
  expr ⇒ expr ⇒ expr ⇒ expr
where
  biinvertible A B f ≡
    product (left-inverse A B f) (right-inverse A B f)

definition equivalence :: expr ⇒ expr ⇒ expr where
  equivalence A B ≡
    sigma-type (arrow A B) (λf. biinvertible A B f)

lemma nondependent-homotopy-form:
  assumes A: A ::h universe i
    and B: B ::h universe i
    and f: f ::h arrow A B
    and g: g ::h arrow A B
  shows homotopy A (λ-. B) f g ::h universe i
proof (rule homotopy-form[OF A])
  fix x
  assume x ::h A
  show B ::h universe i by (rule B)
next
  show f ::h pi-type A (λ-. B)
    using f unfolding arrow-def .
next
  show g ::h pi-type A (λ-. B)
    using g unfolding arrow-def .
qed

lemma nondependent-homotopy-refl-type:
  assumes A: A ::h universe i
    and B: B ::h universe i
    and f: f ::h arrow A B
  shows homotopy-refl A (λ-. B) f ::h
    homotopy A (λ-. B) f f
proof (rule homotopy-refl-type[OF A])
  fix x
  assume x ::h A
  show B ::h universe i by (rule B)
next
  show f ::h pi-type A (λ-. B)
    using f unfolding arrow-def .
qed

lemma left-inverse-form:
  assumes A: A ::h universe i
    and B: B ::h universe i
```

```
    and f: f ::_h arrow A B
  shows left-inverse A B f ::_h universe i
  unfolding left-inverse-def
proof (rule sigma-form)
  show arrow B A ::_h universe i
    by (rule arrow-form[OF B A])
next
  fix g
  assume g: g ::_h arrow B A
  have gf: compose A B A g f ::_h arrow A A
    by (rule compose-type[OF A f g])
  have id: identity-function A ::_h arrow A A
    by (rule identity-function-type[OF A])
  show homotopy A (λ-. A) (compose A B A g f)
      (identity-function A) ::_h universe i
    by (rule nondependent-homotopy-form[OF A A gf id])
qed

lemma right-inverse-form:
  assumes A: A ::_h universe i
    and B: B ::_h universe i
    and f: f ::_h arrow A B
  shows right-inverse A B f ::_h universe i
  unfolding right-inverse-def
proof (rule sigma-form)
  show arrow B A ::_h universe i
    by (rule arrow-form[OF B A])
next
  fix g
  assume g: g ::_h arrow B A
  have fg: compose B A B f g ::_h arrow B B
    by (rule compose-type[OF B g f])
  have id: identity-function B ::_h arrow B B
    by (rule identity-function-type[OF B])
  show homotopy B (λ-. B) (compose B A B f g)
      (identity-function B) ::_h universe i
    by (rule nondependent-homotopy-form[OF B B fg id])
qed

lemma biinvertible-form:
  assumes A: A ::_h universe i
    and B: B ::_h universe i
    and f: f ::_h arrow A B
  shows biinvertible A B f ::_h universe i
  unfolding biinvertible-def
  by (rule product-form[OF left-inverse-form[OF A B f]
       right-inverse-form[OF A B f]])

lemma equivalence-form:
```

```
  assumes A: A ::_h universe i
    and B: B ::_h universe i
  shows equivalence A B ::_h universe i
  unfolding equivalence-def
proof (rule sigma-form)
  show arrow A B ::_h universe i
    by (rule arrow-form[OF A B])
next
  fix f
  assume f: f ::_h arrow A B
  show biinvertible A B f ::_h universe i
    by (rule biinvertible-form[OF A B f])
qed

lemma equivalence-intro:
  assumes f: f ::_h arrow A B
    and w: w ::_h biinvertible A B f
  shows pair f w ::_h equivalence A B
  unfolding equivalence-def
proof (rule sigma-intro)
  show f ::_h arrow A B by (rule f)
next
  show w ::_h biinvertible A B f by (rule w)
qed
```

23 The identity equivalence

```
lemma compose-identity-self [simp]:
  compose A A A (identity-function A) (identity-function A) ≡
    identity-function A
  unfolding compose-def identity-function-def by simp

definition identity-biinvertible :: expr ⇒ expr where
  identity-biinvertible A ≡
    pair
      (pair (identity-function A)
        (homotopy-refl A (λ-. A) (identity-function A)))
      (pair (identity-function A)
        (homotopy-refl A (λ-. A) (identity-function A)))

lemma identity-biinvertible-type:
  assumes A: A ::_h universe i
  shows identity-biinvertible A ::_h
    biinvertible A A (identity-function A)
proof −
  have id: identity-function A ::_h arrow A A
    by (rule identity-function-type[OF A])
  have H: homotopy-refl A (λ-. A) (identity-function A)
      ::_h homotopy A (λ-. A)
```

```
      (identity-function A) (identity-function A)
    by (rule nondependent-homotopy-refl-type[OF A A id])
  have Hcomp: homotopy-refl A (λ-. A) (identity-function A)
      ::h homotopy A (λ-. A)
        (compose A A A (identity-function A) (identity-function A))
        (identity-function A)
    using H by simp
  have left: pair (identity-function A)
      (homotopy-refl A (λ-. A) (identity-function A))
      ::h left-inverse A A (identity-function A)
    unfolding left-inverse-def
  proof (rule sigma-intro)
    show identity-function A ::h arrow A A by (rule id)
  next
    show homotopy-refl A (λ-. A) (identity-function A) ::h
        homotopy A (λ-. A)
          (compose A A A (identity-function A) (identity-function A))
          (identity-function A)
      by (rule Hcomp)
  qed
  have right: pair (identity-function A)
      (homotopy-refl A (λ-. A) (identity-function A))
      ::h right-inverse A A (identity-function A)
    unfolding right-inverse-def
  proof (rule sigma-intro)
    show identity-function A ::h arrow A A by (rule id)
  next
    show homotopy-refl A (λ-. A) (identity-function A) ::h
        homotopy A (λ-. A)
          (compose A A A (identity-function A) (identity-function A))
          (identity-function A)
      by (rule Hcomp)
  qed
  show identity-biinvertible A ::h
      biinvertible A A (identity-function A)
    unfolding identity-biinvertible-def biinvertible-def
    by (rule product-intro[OF left right])
qed

definition identity-equivalence :: expr ⇒ expr where
  identity-equivalence A ≡
    pair (identity-function A) (identity-biinvertible A)

theorem identity-equivalence-type:
  assumes A: A ::h universe i
  shows identity-equivalence A ::h equivalence A A
  unfolding identity-equivalence-def equivalence-def
proof (rule sigma-intro)
  show identity-function A ::h arrow A A
```

by (*rule identity-function-type*[*OF A*])
next
show *identity-biinvertible A* ::$_h$
biinvertible A A (*identity-function A*)
by (*rule identity-biinvertible-type*[*OF A*])
qed

24 Composition of equivalences

definition *left-inverse-function* ::
expr ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr*
where
left-inverse-function A B f w ≡ *first* (*first w*)

definition *left-inverse-homotopy* ::
expr ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr*
where
left-inverse-homotopy A B f w ≡ *second* (*first w*)

definition *right-inverse-function* ::
expr ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr*
where
right-inverse-function A B f w ≡ *first* (*second w*)

definition *right-inverse-homotopy* ::
expr ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr*
where
right-inverse-homotopy A B f w ≡ *second* (*second w*)

lemma *left-inverse-component-type*:
assumes *w*: *w* ::$_h$ *biinvertible A B f*
shows *first w* ::$_h$ *left-inverse A B f*
using *w* **unfolding** *biinvertible-def* **by** (*rule product-first*)

lemma *right-inverse-component-type*:
assumes *w*: *w* ::$_h$ *biinvertible A B f*
shows *second w* ::$_h$ *right-inverse A B f*
using *w* **unfolding** *biinvertible-def* **by** (*rule product-second*)

lemma *left-inverse-function-type*:
assumes *w*: *w* ::$_h$ *biinvertible A B f*
shows *left-inverse-function A B f w* ::$_h$ *arrow B A*
proof −
have *L*: *first w* ::$_h$ *left-inverse A B f*
by (*rule left-inverse-component-type*[*OF w*])
show *left-inverse-function A B f w* ::$_h$ *arrow B A*
unfolding *left-inverse-function-def*
using *L* **unfolding** *left-inverse-def* **by** (*rule sigma-first*)
qed

lemma *left-inverse-homotopy-type*:
assumes *w*: $w ::_h$ *biinvertible A B f*
shows *left-inverse-homotopy A B f w* $::_h$
homotopy A (λ-. *A*)
(*compose A B A* (*left-inverse-function A B f w*) *f*)
(*identity-function A*)
proof −
have *L*: *first w* $::_h$ *left-inverse A B f*
by (*rule left-inverse-component-type*[*OF w*])
show *left-inverse-homotopy A B f w* $::_h$
homotopy A (λ-. *A*)
(*compose A B A* (*left-inverse-function A B f w*) *f*)
(*identity-function A*)
unfolding *left-inverse-homotopy-def left-inverse-function-def*
using *L* **unfolding** *left-inverse-def* **by** (*rule sigma-second*)
qed

lemma *right-inverse-function-type*:
assumes *w*: $w ::_h$ *biinvertible A B f*
shows *right-inverse-function A B f w* $::_h$ *arrow B A*
proof −
have *R*: *second w* $::_h$ *right-inverse A B f*
by (*rule right-inverse-component-type*[*OF w*])
show *right-inverse-function A B f w* $::_h$ *arrow B A*
unfolding *right-inverse-function-def*
using *R* **unfolding** *right-inverse-def* **by** (*rule sigma-first*)
qed

lemma *right-inverse-homotopy-type*:
assumes *w*: $w ::_h$ *biinvertible A B f*
shows *right-inverse-homotopy A B f w* $::_h$
homotopy B (λ-. *B*)
(*compose B A B f* (*right-inverse-function A B f w*))
(*identity-function B*)
proof −
have *R*: *second w* $::_h$ *right-inverse A B f*
by (*rule right-inverse-component-type*[*OF w*])
show *right-inverse-homotopy A B f w* $::_h$
homotopy B (λ-. *B*)
(*compose B A B f* (*right-inverse-function A B f w*))
(*identity-function B*)
unfolding *right-inverse-homotopy-def right-inverse-function-def*
using *R* **unfolding** *right-inverse-def* **by** (*rule sigma-second*)
qed

definition *composite-left-inverse-function* ::
expr ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒
expr ⇒ *expr* ⇒ *expr*

where
composite-left-inverse-function A B C f g wf wg ≡
compose C B A
(*left-inverse-function A B f wf*)
(*left-inverse-function B C g wg*)

lemma *composite-left-inverse-function-type*:
assumes *C*: *C* ::$_h$ *universe i*
and *wf*: *wf* ::$_h$ *biinvertible A B f*
and *wg*: *wg* ::$_h$ *biinvertible B C g*
shows *composite-left-inverse-function A B C f g wf wg*
::$_h$ *arrow C A*
unfolding *composite-left-inverse-function-def*
by (*rule compose-type*[*OF C*
left-inverse-function-type[*OF wg*]
left-inverse-function-type[*OF wf*]])

definition *composite-left-homotopy* ::
expr ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒
expr ⇒ *expr* ⇒ *expr*
where
composite-left-homotopy A B C f g wf wg ≡
lambda A (λ*x*.
path-concat A
(*app* (*left-inverse-function A B f wf*)
(*app* (*left-inverse-function B C g wg*) (*app g* (*app f x*))))
(*app* (*left-inverse-function A B f wf*) (*app f x*))
x
(*ap B A* (*left-inverse-function A B f wf*)
(*app* (*left-inverse-function B C g wg*) (*app g* (*app f x*)))
(*app f x*)
(*app* (*left-inverse-homotopy B C g wg*) (*app f x*)))
(*app* (*left-inverse-homotopy A B f wf*) *x*))

lemma *composite-left-homotopy-type*:
assumes *A*: *A* ::$_h$ *universe i*
and *B*: *B* ::$_h$ *universe i*
and *C*: *C* ::$_h$ *universe i*
and *f*: *f* ::$_h$ *arrow A B*
and *g*: *g* ::$_h$ *arrow B C*
and *wf*: *wf* ::$_h$ *biinvertible A B f*
and *wg*: *wg* ::$_h$ *biinvertible B C g*
shows *composite-left-homotopy A B C f g wf wg* ::$_h$
homotopy A (λ-. *A*)
(*compose A C A*
(*composite-left-inverse-function A B C f g wf wg*)
(*compose A B C g f*))
(*identity-function A*)
unfolding *composite-left-homotopy-def homotopy-def*

```
    composite-left-inverse-function-def
proof (rule pi-intro[OF A])
  fix x
  assume x: x ::h A
  have fx: app f x ::h B
    by (rule arrow-elim[OF f x])
  have gfx: app g (app f x) ::h C
    by (rule arrow-elim[OF g fx])
  have lf: left-inverse-function A B f wf ::h arrow B A
    by (rule left-inverse-function-type[OF wf])
  have lg: left-inverse-function B C g wg ::h arrow C B
    by (rule left-inverse-function-type[OF wg])
  have lg-gfx: app (left-inverse-function B C g wg) (app g (app f x))
      ::h B
    by (rule arrow-elim[OF lg gfx])
  have lf-lg-gfx: app (left-inverse-function A B f wf)
      (app (left-inverse-function B C g wg) (app g (app f x)))
      ::h A
    by (rule arrow-elim[OF lf lg-gfx])
  have lf-fx: app (left-inverse-function A B f wf) (app f x)
      ::h A
    by (rule arrow-elim[OF lf fx])
  have Lg: left-inverse-homotopy B C g wg ::h
      homotopy B (λ-. B)
        (compose B C B (left-inverse-function B C g wg) g)
        (identity-function B)
    by (rule left-inverse-homotopy-type[OF wg])
  have Lg-pi: left-inverse-homotopy B C g wg ::h
      pi-type B
        (λu. identity-type B
          (app (compose B C B (left-inverse-function B C g wg) g) u)
          (app (identity-function B) u))
    using Lg unfolding homotopy-def .
  have Lg-x-raw: app (left-inverse-homotopy B C g wg) (app f x)
      ::h identity-type B
        (app (compose B C B (left-inverse-function B C g wg) g) (app f x))
        (app (identity-function B) (app f x))
    by (rule pi-elim[OF Lg-pi fx])
  have Lg-x: app (left-inverse-homotopy B C g wg) (app f x)
      ::h identity-type B
        (app (left-inverse-function B C g wg) (app g (app f x)))
        (app f x)
    using Lg-x-raw by simp
  have step-one: ap B A (left-inverse-function A B f wf)
      (app (left-inverse-function B C g wg) (app g (app f x)))
      (app f x)
      (app (left-inverse-homotopy B C g wg) (app f x))
      ::h identity-type A
        (app (left-inverse-function A B f wf)
```

```
      (app (left-inverse-function B C g wg) (app g (app f x))))
    (app (left-inverse-function A B f wf) (app f x))
  by (rule ap-type[OF B A lf lg-gfx fx Lg-x])
have Lf: left-inverse-homotopy A B f wf ::h
    homotopy A (λ-. A)
      (compose A B A (left-inverse-function A B f wf) f)
      (identity-function A)
  by (rule left-inverse-homotopy-type[OF wf])
have Lf-pi: left-inverse-homotopy A B f wf ::h
    pi-type A
      (λu. identity-type A
        (app (compose A B A (left-inverse-function A B f wf) f) u)
        (app (identity-function A) u))
  using Lf unfolding homotopy-def .
have Lf-x-raw: app (left-inverse-homotopy A B f wf) x
    ::h identity-type A
      (app (compose A B A (left-inverse-function A B f wf) f) x)
      (app (identity-function A) x)
  by (rule pi-elim[OF Lf-pi x])
have Lf-x: app (left-inverse-homotopy A B f wf) x
    ::h identity-type A
      (app (left-inverse-function A B f wf) (app f x)) x
  using Lf-x-raw by simp
have composed: path-concat A
    (app (left-inverse-function A B f wf)
      (app (left-inverse-function B C g wg) (app g (app f x))))
    (app (left-inverse-function A B f wf) (app f x))
    x
    (ap B A (left-inverse-function A B f wf)
      (app (left-inverse-function B C g wg) (app g (app f x)))
      (app f x)
      (app (left-inverse-homotopy B C g wg) (app f x)))
    (app (left-inverse-homotopy A B f wf) x)
    ::h identity-type A
      (app (left-inverse-function A B f wf)
        (app (left-inverse-function B C g wg) (app g (app f x)))) x
  by (rule path-concat-type[OF A lf-lg-gfx lf-fx x step-one Lf-x])
show path-concat A
    (app (left-inverse-function A B f wf)
      (app (left-inverse-function B C g wg) (app g (app f x))))
    (app (left-inverse-function A B f wf) (app f x)) x
    (ap B A (left-inverse-function A B f wf)
      (app (left-inverse-function B C g wg) (app g (app f x)))
      (app f x)
      (app (left-inverse-homotopy B C g wg) (app f x)))
    (app (left-inverse-homotopy A B f wf) x)
    ::h identity-type A
      (app
        (compose A C A
```

```
          (compose C B A
            (left-inverse-function A B f wf)
            (left-inverse-function B C g wg))
          (compose A B C g f)) x)
        (app (identity-function A) x)
    using composed by simp
qed

definition composite-left-inverse ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr ⇒
    expr ⇒ expr ⇒ expr
where
  composite-left-inverse A B C f g wf wg ≡
    pair
      (composite-left-inverse-function A B C f g wf wg)
      (composite-left-homotopy A B C f g wf wg)

lemma composite-left-inverse-type:
  assumes A: A ::h universe i
    and B: B ::h universe i
    and C: C ::h universe i
    and f: f ::h arrow A B
    and g: g ::h arrow B C
    and wf: wf ::h biinvertible A B f
    and wg: wg ::h biinvertible B C g
  shows composite-left-inverse A B C f g wf wg ::h
    left-inverse A C (compose A B C g f)
  unfolding composite-left-inverse-def left-inverse-def
proof (rule sigma-intro)
  show composite-left-inverse-function A B C f g wf wg ::h
      arrow C A
    by (rule composite-left-inverse-function-type[OF C wf wg])
next
  show composite-left-homotopy A B C f g wf wg ::h
      homotopy A (λ-. A)
        (compose A C A
          (composite-left-inverse-function A B C f g wf wg)
          (compose A B C g f))
        (identity-function A)
    by (rule composite-left-homotopy-type[OF A B C f g wf wg])
qed

definition composite-right-inverse-function ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr ⇒
    expr ⇒ expr ⇒ expr
where
  composite-right-inverse-function A B C f g wf wg ≡
    compose C B A
      (right-inverse-function A B f wf)
```

```
    (right-inverse-function B C g wg)

lemma composite-right-inverse-function-type:
  assumes C: C ::_h universe i
    and wf: wf ::_h biinvertible A B f
    and wg: wg ::_h biinvertible B C g
  shows composite-right-inverse-function A B C f g wf wg
    ::_h arrow C A
  unfolding composite-right-inverse-function-def
  by (rule compose-type[OF C
        right-inverse-function-type[OF wg]
        right-inverse-function-type[OF wf]])

definition composite-right-homotopy ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr ⇒
    expr ⇒ expr ⇒ expr
where
  composite-right-homotopy A B C f g wf wg ≡
    lambda C (λz.
      path-concat C
        (app g (app f
          (app (right-inverse-function A B f wf)
            (app (right-inverse-function B C g wg) z))))
        (app g (app (right-inverse-function B C g wg) z))
        z
        (ap B C g
          (app f
            (app (right-inverse-function A B f wf)
              (app (right-inverse-function B C g wg) z)))
          (app (right-inverse-function B C g wg) z)
          (app (right-inverse-homotopy A B f wf)
            (app (right-inverse-function B C g wg) z)))
        (app (right-inverse-homotopy B C g wg) z))

lemma composite-right-homotopy-type:
  assumes A: A ::_h universe i
    and B: B ::_h universe i
    and C: C ::_h universe i
    and f: f ::_h arrow A B
    and g: g ::_h arrow B C
    and wf: wf ::_h biinvertible A B f
    and wg: wg ::_h biinvertible B C g
  shows composite-right-homotopy A B C f g wf wg ::_h
    homotopy C (λ-. C)
      (compose C A C (compose A B C g f)
        (composite-right-inverse-function A B C f g wf wg))
      (identity-function C)
  unfolding composite-right-homotopy-def homotopy-def
    composite-right-inverse-function-def
```

proof (*rule pi-intro*[*OF C*])
fix *z*
assume *z*: $z ::_h C$
have *rf*: *right-inverse-function A B f wf* $::_h$ *arrow B A*
by (*rule right-inverse-function-type*[*OF wf*])
have *rg*: *right-inverse-function B C g wg* $::_h$ *arrow C B*
by (*rule right-inverse-function-type*[*OF wg*])
have *rgz*: *app* (*right-inverse-function B C g wg*) *z* $::_h$ *B*
by (*rule arrow-elim*[*OF rg z*])
have *rf-rgz*: *app* (*right-inverse-function A B f wf*)
(*app* (*right-inverse-function B C g wg*) *z*) $::_h$ *A*
by (*rule arrow-elim*[*OF rf rgz*])
have *f-rf-rgz*: *app f*
(*app* (*right-inverse-function A B f wf*)
(*app* (*right-inverse-function B C g wg*) *z*)) $::_h$ *B*
by (*rule arrow-elim*[*OF f rf-rgz*])
have *g-f-rf-rgz*: *app g* (*app f*
(*app* (*right-inverse-function A B f wf*)
(*app* (*right-inverse-function B C g wg*) *z*))) $::_h$ *C*
by (*rule arrow-elim*[*OF g f-rf-rgz*])
have *g-rgz*: *app g* (*app* (*right-inverse-function B C g wg*) *z*)
$::_h$ *C*
by (*rule arrow-elim*[*OF g rgz*])
have *Rf*: *right-inverse-homotopy A B f wf* $::_h$
homotopy B (λ*-. B*)
(*compose B A B f* (*right-inverse-function A B f wf*))
(*identity-function B*)
by (*rule right-inverse-homotopy-type*[*OF wf*])
have *Rf-pi*: *right-inverse-homotopy A B f wf* $::_h$
pi-type B
(λ*u. identity-type B*
(*app* (*compose B A B f* (*right-inverse-function A B f wf*)) *u*)
(*app* (*identity-function B*) *u*))
using *Rf* **unfolding** *homotopy-def* **.**
have *Rf-z-raw*: *app* (*right-inverse-homotopy A B f wf*)
(*app* (*right-inverse-function B C g wg*) *z*)
$::_h$ *identity-type B*
(*app* (*compose B A B f* (*right-inverse-function A B f wf*))
(*app* (*right-inverse-function B C g wg*) *z*))
(*app* (*identity-function B*)
(*app* (*right-inverse-function B C g wg*) *z*))
by (*rule pi-elim*[*OF Rf-pi rgz*])
have *Rf-z*: *app* (*right-inverse-homotopy A B f wf*)
(*app* (*right-inverse-function B C g wg*) *z*)
$::_h$ *identity-type B*
(*app f* (*app* (*right-inverse-function A B f wf*)
(*app* (*right-inverse-function B C g wg*) *z*)))
(*app* (*right-inverse-function B C g wg*) *z*)
using *Rf-z-raw* **by** *simp*

have *step-one*: *ap B C g*
  (*app f* (*app* (*right-inverse-function A B f wf*)
    (*app* (*right-inverse-function B C g wg*) *z*)))
  (*app* (*right-inverse-function B C g wg*) *z*)
  (*app* (*right-inverse-homotopy A B f wf*)
    (*app* (*right-inverse-function B C g wg*) *z*))
  ::$_h$ *identity-type C*
    (*app g* (*app f* (*app* (*right-inverse-function A B f wf*)
      (*app* (*right-inverse-function B C g wg*) *z*))))
    (*app g* (*app* (*right-inverse-function B C g wg*) *z*))
 by (*rule ap-type*[*OF B C g f-rf-rgz rgz Rf-z*])
have *Rg*: *right-inverse-homotopy B C g wg* ::$_h$
  homotopy C (*λ-. C*)
    (*compose C B C g* (*right-inverse-function B C g wg*))
    (*identity-function C*)
 by (*rule right-inverse-homotopy-type*[*OF wg*])
have *Rg-pi*: *right-inverse-homotopy B C g wg* ::$_h$
  pi-type C
    (*λu. identity-type C*
      (*app* (*compose C B C g* (*right-inverse-function B C g wg*)) *u*)
      (*app* (*identity-function C*) *u*))
 using *Rg* **unfolding** *homotopy-def* **.**
have *Rg-z-raw*: *app* (*right-inverse-homotopy B C g wg*) *z*
  ::$_h$ *identity-type C*
    (*app* (*compose C B C g* (*right-inverse-function B C g wg*)) *z*)
    (*app* (*identity-function C*) *z*)
 by (*rule pi-elim*[*OF Rg-pi z*])
have *Rg-z*: *app* (*right-inverse-homotopy B C g wg*) *z*
  ::$_h$ *identity-type C*
    (*app g* (*app* (*right-inverse-function B C g wg*) *z*)) *z*
 using *Rg-z-raw* **by** *simp*
have *composed*: *path-concat C*
  (*app g* (*app f* (*app* (*right-inverse-function A B f wf*)
    (*app* (*right-inverse-function B C g wg*) *z*))))
  (*app g* (*app* (*right-inverse-function B C g wg*) *z*)) *z*
  (*ap B C g*
    (*app f* (*app* (*right-inverse-function A B f wf*)
      (*app* (*right-inverse-function B C g wg*) *z*)))
    (*app* (*right-inverse-function B C g wg*) *z*)
    (*app* (*right-inverse-homotopy A B f wf*)
      (*app* (*right-inverse-function B C g wg*) *z*)))
  (*app* (*right-inverse-homotopy B C g wg*) *z*)
  ::$_h$ *identity-type C*
    (*app g* (*app f* (*app* (*right-inverse-function A B f wf*)
      (*app* (*right-inverse-function B C g wg*) *z*)))) *z*
 by (*rule path-concat-type*[*OF C g-f-rf-rgz g-rgz z step-one Rg-z*])
show *path-concat C*
  (*app g* (*app f* (*app* (*right-inverse-function A B f wf*)
    (*app* (*right-inverse-function B C g wg*) *z*))))

```
    (app g (app (right-inverse-function B C g wg) z)) z
    (ap B C g
      (app f (app (right-inverse-function A B f wf)
        (app (right-inverse-function B C g wg) z)))
      (app (right-inverse-function B C g wg) z)
      (app (right-inverse-homotopy A B f wf)
        (app (right-inverse-function B C g wg) z)))
    (app (right-inverse-homotopy B C g wg) z)
    ::h identity-type C
      (app
        (compose C A C (compose A B C g f)
          (compose C B A
            (right-inverse-function A B f wf)
            (right-inverse-function B C g wg))) z)
      (app (identity-function C) z)
  using composed by simp
qed

definition composite-right-inverse ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr ⇒
    expr ⇒ expr ⇒ expr
where
  composite-right-inverse A B C f g wf wg ≡
    pair
      (composite-right-inverse-function A B C f g wf wg)
      (composite-right-homotopy A B C f g wf wg)

lemma composite-right-inverse-type:
  assumes A: A ::h universe i
    and B: B ::h universe i
    and C: C ::h universe i
    and f: f ::h arrow A B
    and g: g ::h arrow B C
    and wf: wf ::h biinvertible A B f
    and wg: wg ::h biinvertible B C g
  shows composite-right-inverse A B C f g wf wg ::h
    right-inverse A C (compose A B C g f)
  unfolding composite-right-inverse-def right-inverse-def
proof (rule sigma-intro)
  show composite-right-inverse-function A B C f g wf wg ::h
      arrow C A
    by (rule composite-right-inverse-function-type[OF C wf wg])
next
  show composite-right-homotopy A B C f g wf wg ::h
      homotopy C (λ-. C)
        (compose C A C (compose A B C g f)
          (composite-right-inverse-function A B C f g wf wg))
        (identity-function C)
    by (rule composite-right-homotopy-type[OF A B C f g wf wg])
```

qed

definition *compose-biinvertible* ::
  expr ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒
    expr ⇒ *expr* ⇒ *expr*
where
  compose-biinvertible A B C f g wf wg ≡
    pair
      (*composite-left-inverse A B C f g wf wg*)
      (*composite-right-inverse A B C f g wf wg*)

lemma *compose-biinvertible-type*:
  assumes *A*: *A* ::$_h$ *universe i*
    and *B*: *B* ::$_h$ *universe i*
    and *C*: *C* ::$_h$ *universe i*
    and *f*: *f* ::$_h$ *arrow A B*
    and *g*: *g* ::$_h$ *arrow B C*
    and *wf*: *wf* ::$_h$ *biinvertible A B f*
    and *wg*: *wg* ::$_h$ *biinvertible B C g*
  shows *compose-biinvertible A B C f g wf wg* ::$_h$
    biinvertible A C (*compose A B C g f*)
  unfolding *compose-biinvertible-def biinvertible-def*
  by (*rule product-intro*[*OF*
      composite-left-inverse-type[*OF A B C f g wf wg*]
      composite-right-inverse-type[*OF A B C f g wf wg*]])

definition *compose-equivalence* ::
  expr ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr*
where
  compose-equivalence A B C e d ≡
    pair
      (*compose A B C* (*first d*) (*first e*))
      (*compose-biinvertible A B C*
        (*first e*) (*first d*) (*second e*) (*second d*))

theorem *compose-equivalence-type*:
  assumes *A*: *A* ::$_h$ *universe i*
    and *B*: *B* ::$_h$ *universe i*
    and *C*: *C* ::$_h$ *universe i*
    and *e*: *e* ::$_h$ *equivalence A B*
    and *d*: *d* ::$_h$ *equivalence B C*
  shows *compose-equivalence A B C e d* ::$_h$ *equivalence A C*
proof −
  have *ef*: *first e* ::$_h$ *arrow A B*
    using *e* **unfolding** *equivalence-def* **by** (*rule sigma-first*)
  have *ew*: *second e* ::$_h$ *biinvertible A B* (*first e*)
    using *e* **unfolding** *equivalence-def* **by** (*rule sigma-second*)
  have *df*: *first d* ::$_h$ *arrow B C*
    using *d* **unfolding** *equivalence-def* **by** (*rule sigma-first*)

have *dw*: *second d* $::_h$ *biinvertible B C* (*first d*)
using *d* **unfolding** *equivalence-def* **by** (*rule sigma-second*)
have *composite*: *compose A B C* (*first d*) (*first e*)
$::_h$ *arrow A C*
by (*rule compose-type*[*OF A ef df*])
have *witness*: *compose-biinvertible A B C*
(*first e*) (*first d*) (*second e*) (*second d*)
$::_h$ *biinvertible A C*
(*compose A B C* (*first d*) (*first e*))
by (*rule compose-biinvertible-type*[*OF A B C ef df ew dw*])
show *compose-equivalence A B C e d* $::_h$ *equivalence A C*
unfolding *compose-equivalence-def equivalence-def*
by (*intro sigma-intro composite witness*)
qed

end
theory *HoTT-Equivalence-Characterization*
imports *HoTT-Equivalence*
begin

25 Extracting contraction data

definition *contraction-center* :: *expr* ⇒ *expr* **where**
contraction-center c ≡ *first c*

definition *contraction-paths* :: *expr* ⇒ *expr* **where**
contraction-paths c ≡ *second c*

lemma *contraction-center-type*:
assumes *c*: *c* $::_h$ *contractible A*
shows *contraction-center c* $::_h$ *A*
unfolding *contraction-center-def*
using *c* **unfolding** *contractible-def* **by** (*rule sigma-first*)

lemma *contraction-paths-type*:
assumes *c*: *c* $::_h$ *contractible A*
shows *contraction-paths c* $::_h$
pi-type A (λ*x*. *identity-type A* (*contraction-center c*) *x*)
unfolding *contraction-paths-def contraction-center-def*
using *c* **unfolding** *contractible-def* **by** (*rule sigma-second*)

definition *fiber-center* ::
expr ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr*
where
fiber-center A B f c b ≡ *contraction-center* (*app c b*)

definition *fiber-contraction* ::
expr ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr*
where

fiber-contraction A B f c b ≡ *contraction-paths* (*app c b*)

lemma *contractible-map-component-type*:
assumes *c*: *c* ::$_h$ *contractible-map A B f*
and *b*: *b* ::$_h$ *B*
shows *app c b* ::$_h$ *contractible* (*fiber A B f b*)
using *c* **unfolding** *contractible-map-def*
by (*rule pi-elim*[*OF - b*])

lemma *fiber-center-type*:
assumes *c*: *c* ::$_h$ *contractible-map A B f*
and *b*: *b* ::$_h$ *B*
shows *fiber-center A B f c b* ::$_h$ *fiber A B f b*
unfolding *fiber-center-def*
by (*rule contraction-center-type*[*OF contractible-map-component-type*[*OF c b*]])

lemma *fiber-contraction-type*:
assumes *c*: *c* ::$_h$ *contractible-map A B f*
and *b*: *b* ::$_h$ *B*
shows *fiber-contraction A B f c b* ::$_h$
pi-type (*fiber A B f b*)
(λ*z*. *identity-type* (*fiber A B f b*)
(*fiber-center A B f c b*) *z*)
unfolding *fiber-contraction-def fiber-center-def*
by (*rule contraction-paths-type*[*OF contractible-map-component-type*[*OF c b*]])

definition *fiber-first-map* ::
expr ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr*
where
fiber-first-map A B f b ≡
lambda (*fiber A B f b*) (λ*z*. *first z*)

lemma *fiber-first-map-type*:
assumes *A*: *A* ::$_h$ *universe i*
and *B*: *B* ::$_h$ *universe i*
and *f*: *f* ::$_h$ *arrow A B*
and *b*: *b* ::$_h$ *B*
shows *fiber-first-map A B f b* ::$_h$ *arrow* (*fiber A B f b*) *A*
proof −
have *fib*: *fiber A B f b* ::$_h$ *universe i*
by (*rule fiber-form*[*OF A B f b*])
show *fiber-first-map A B f b* ::$_h$
arrow (*fiber A B f b*) *A*
unfolding *fiber-first-map-def*
proof (*rule arrow-intro*[*OF fib*])
fix *z*
assume *z*: *z* ::$_h$ *fiber A B f b*
show *first z* ::$_h$ *A*
using *z* **unfolding** *fiber-def* **by** (*rule sigma-first*)

qed
qed

26 Contractible fibers yield a bi-inverse

definition *contractible-inverse* ::
expr ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr*
where
contractible-inverse A B f c ≡
lambda B (λ*b*. *first* (*fiber-center A B f c b*))

lemma *contractible-inverse-type*:
assumes *B*: *B* ::$_h$ *universe i*
and *c*: *c* ::$_h$ *contractible-map A B f*
shows *contractible-inverse A B f c* ::$_h$ *arrow B A*
unfolding *contractible-inverse-def*
proof (*rule arrow-intro*[*OF B*])
fix *b*
assume *b*: *b* ::$_h$ *B*
have *center*: *fiber-center A B f c b* ::$_h$ *fiber A B f b*
by (*rule fiber-center-type*[*OF c b*])
show *first* (*fiber-center A B f c b*) ::$_h$ *A*
using *center* **unfolding** *fiber-def* **by** (*rule sigma-first*)
qed

definition *contractible-right-homotopy* ::
expr ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr*
where
contractible-right-homotopy A B f c ≡
lambda B (λ*b*. *second* (*fiber-center A B f c b*))

lemma *contractible-right-homotopy-type*:
assumes *B*: *B* ::$_h$ *universe i*
and *f*: *f* ::$_h$ *arrow A B*
and *c*: *c* ::$_h$ *contractible-map A B f*
shows *contractible-right-homotopy A B f c* ::$_h$
homotopy B (λ-. *B*)
(*compose B A B f* (*contractible-inverse A B f c*))
(*identity-function B*)
unfolding *contractible-right-homotopy-def homotopy-def*
proof (*rule pi-intro*[*OF B*])
fix *b*
assume *b*: *b* ::$_h$ *B*
have *center*: *fiber-center A B f c b* ::$_h$ *fiber A B f b*
by (*rule fiber-center-type*[*OF c b*])
have *path*: *second* (*fiber-center A B f c b*) ::$_h$
identity-type B
(*app f* (*first* (*fiber-center A B f c b*))) *b*
using *center* **unfolding** *fiber-def* **by** (*rule sigma-second*)

```
  show second (fiber-center A B f c b) ::h
      identity-type B
        (app
          (compose B A B f (contractible-inverse A B f c)) b)
        (app (identity-function B) b)
    using path unfolding contractible-inverse-def by simp
qed

definition reflexive-fiber-point ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr
where
  reflexive-fiber-point A B f a ≡
    pair a (reflexivity B (app f a))

lemma reflexive-fiber-point-type:
  assumes A: A ::h universe i
    and B: B ::h universe i
    and f: f ::h arrow A B
    and a: a ::h A
  shows reflexive-fiber-point A B f a ::h
    fiber A B f (app f a)
proof −
  have fa: app f a ::h B by (rule arrow-elim[OF f a])
  have refl: reflexivity B (app f a) ::h
      identity-type B (app f a) (app f a)
    by (rule identity-intro[OF B fa])
  show reflexive-fiber-point A B f a ::h
      fiber A B f (app f a)
    unfolding reflexive-fiber-point-def fiber-def
    by (rule sigma-intro[where B=λx. identity-type B (app f x) (app f a),
          OF a refl])
qed

definition contractible-left-homotopy ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr
where
  contractible-left-homotopy A B f c ≡
    lambda A (λa.
      ap
        (fiber A B f (app f a))
        A
        (fiber-first-map A B f (app f a))
        (fiber-center A B f c (app f a))
        (reflexive-fiber-point A B f a)
        (app
          (fiber-contraction A B f c (app f a))
          (reflexive-fiber-point A B f a)))

lemma contractible-left-homotopy-type:
```

```
  assumes A: A ::_h universe i
    and B: B ::_h universe i
    and f: f ::_h arrow A B
    and c: c ::_h contractible-map A B f
  shows contractible-left-homotopy A B f c ::_h
    homotopy A (λ-. A)
      (compose A B A (contractible-inverse A B f c) f)
      (identity-function A)
  unfolding contractible-left-homotopy-def homotopy-def
proof (rule pi-intro[OF A])
  fix a
  assume a: a ::_h A
  have fa: app f a ::_h B by (rule arrow-elim[OF f a])
  have fib: fiber A B f (app f a) ::_h universe i
    by (rule fiber-form[OF A B f fa])
  have center: fiber-center A B f c (app f a) ::_h
      fiber A B f (app f a)
    by (rule fiber-center-type[OF c fa])
  have point: reflexive-fiber-point A B f a ::_h
      fiber A B f (app f a)
    by (rule reflexive-fiber-point-type[OF A B f a])
  have contraction: fiber-contraction A B f c (app f a) ::_h
      pi-type (fiber A B f (app f a))
        (λz. identity-type (fiber A B f (app f a))
          (fiber-center A B f c (app f a)) z)
    by (rule fiber-contraction-type[OF c fa])
  have center-to-point:
      app (fiber-contraction A B f c (app f a))
        (reflexive-fiber-point A B f a) ::_h
      identity-type (fiber A B f (app f a))
        (fiber-center A B f c (app f a))
        (reflexive-fiber-point A B f a)
    by (rule pi-elim[OF contraction point])
  have first-map: fiber-first-map A B f (app f a) ::_h
      arrow (fiber A B f (app f a)) A
    by (rule fiber-first-map-type[OF A B f fa])
  have projected:
      ap (fiber A B f (app f a)) A
        (fiber-first-map A B f (app f a))
        (fiber-center A B f c (app f a))
        (reflexive-fiber-point A B f a)
        (app (fiber-contraction A B f c (app f a))
          (reflexive-fiber-point A B f a)) ::_h
      identity-type A
        (app (fiber-first-map A B f (app f a))
          (fiber-center A B f c (app f a)))
        (app (fiber-first-map A B f (app f a))
          (reflexive-fiber-point A B f a))
    by (rule ap-type[OF fib A first-map center point center-to-point])
```

```
  show ap (fiber A B f (app f a)) A
       (fiber-first-map A B f (app f a))
       (fiber-center A B f c (app f a))
       (reflexive-fiber-point A B f a)
       (app (fiber-contraction A B f c (app f a))
         (reflexive-fiber-point A B f a)) ::_h
      identity-type A
       (app
         (compose A B A (contractible-inverse A B f c) f) a)
       (app (identity-function A) a)
    using projected
    unfolding fiber-first-map-def reflexive-fiber-point-def
      contractible-inverse-def by simp
qed

definition contractible-left-inverse ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr
where
  contractible-left-inverse A B f c ≡
    pair
      (contractible-inverse A B f c)
      (contractible-left-homotopy A B f c)

lemma contractible-left-inverse-type:
  assumes A: A ::_h universe i
    and B: B ::_h universe i
    and f: f ::_h arrow A B
    and c: c ::_h contractible-map A B f
  shows contractible-left-inverse A B f c ::_h
    left-inverse A B f
  unfolding contractible-left-inverse-def left-inverse-def
proof (rule sigma-intro)
  show contractible-inverse A B f c ::_h arrow B A
    by (rule contractible-inverse-type[OF B c])
next
  show contractible-left-homotopy A B f c ::_h
      homotopy A (λ-. A)
        (compose A B A (contractible-inverse A B f c) f)
        (identity-function A)
    by (rule contractible-left-homotopy-type[OF A B f c])
qed

definition contractible-right-inverse ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr
where
  contractible-right-inverse A B f c ≡
    pair
      (contractible-inverse A B f c)
      (contractible-right-homotopy A B f c)
```

lemma *contractible-right-inverse-type*:
  assumes *A*: $A ::_h$ *universe i*
    and *B*: $B ::_h$ *universe i*
    and *f*: $f ::_h$ *arrow A B*
    and *c*: $c ::_h$ *contractible-map A B f*
  shows *contractible-right-inverse A B f c* $::_h$
    right-inverse A B f
  unfolding *contractible-right-inverse-def right-inverse-def*
proof (*rule sigma-intro*)
  show *contractible-inverse A B f c* $::_h$ *arrow B A*
    by (*rule contractible-inverse-type*[*OF B c*])
next
  show *contractible-right-homotopy A B f c* $::_h$
      homotopy B (λ-. *B*)
        (*compose B A B f* (*contractible-inverse A B f c*))
        (*identity-function B*)
    by (*rule contractible-right-homotopy-type*[*OF B f c*])
qed

definition *contractible-map-to-biinvertible* ::
  expr ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr*
where
  contractible-map-to-biinvertible A B f c ≡
    pair
      (*contractible-left-inverse A B f c*)
      (*contractible-right-inverse A B f c*)

theorem *contractible-map-to-biinvertible-type*:
  assumes *A*: $A ::_h$ *universe i*
    and *B*: $B ::_h$ *universe i*
    and *f*: $f ::_h$ *arrow A B*
    and *c*: $c ::_h$ *contractible-map A B f*
  shows *contractible-map-to-biinvertible A B f c* $::_h$
    biinvertible A B f
  unfolding *contractible-map-to-biinvertible-def biinvertible-def*
  by (*rule product-intro*[*OF*
      contractible-left-inverse-type[*OF A B f c*]
      contractible-right-inverse-type[*OF A B f c*]])

definition *contractible-map-to-biinvertible-map* ::
  expr ⇒ *expr* ⇒ *expr* ⇒ *expr*
where
  contractible-map-to-biinvertible-map A B f ≡
    lambda (*contractible-map A B f*)
      (λ*c*. *contractible-map-to-biinvertible A B f c*)

corollary *contractible-map-to-biinvertible-map-type*:
  assumes *A*: $A ::_h$ *universe i*

```
    and B: B ::_h universe i
    and f: f ::_h arrow A B
  shows contractible-map-to-biinvertible-map A B f ::_h
    arrow (contractible-map A B f) (biinvertible A B f)
  unfolding contractible-map-to-biinvertible-map-def
proof (rule arrow-intro[OF contractible-map-form[OF A B f]])
  fix c
  assume c: c ::_h contractible-map A B f
  show contractible-map-to-biinvertible A B f c ::_h
      biinvertible A B f
    by (rule contractible-map-to-biinvertible-type[OF A B f c])
qed
```

27 Equivalences preserve contractibility

```
definition equivalence-function :: expr ⇒ expr where
  equivalence-function e ≡ first e

definition equivalence-biinvertible :: expr ⇒ expr where
  equivalence-biinvertible e ≡ second e

lemma equivalence-function-type:
  assumes e: e ::_h equivalence A B
  shows equivalence-function e ::_h arrow A B
  unfolding equivalence-function-def
  using e unfolding equivalence-def by (rule sigma-first)

lemma equivalence-biinvertible-type:
  assumes e: e ::_h equivalence A B
  shows equivalence-biinvertible e ::_h
    biinvertible A B (equivalence-function e)
  unfolding equivalence-biinvertible-def equivalence-function-def
  using e unfolding equivalence-def by (rule sigma-second)

definition equivalence-contractible-center ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr
where
  equivalence-contractible-center A B e c ≡
    app (equivalence-function e) (contraction-center c)

definition equivalence-contractible-paths ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr
where
  equivalence-contractible-paths A B e c ≡
    lambda B (λb.
      path-concat B
        (equivalence-contractible-center A B e c)
        (app (equivalence-function e)
          (app
```

```
      (right-inverse-function A B
        (equivalence-function e) (equivalence-biinvertible e))
      b))
  b
  (ap A B
    (equivalence-function e)
    (contraction-center c)
    (app
      (right-inverse-function A B
        (equivalence-function e) (equivalence-biinvertible e))
      b)
    (app
      (contraction-paths c)
      (app
        (right-inverse-function A B
          (equivalence-function e) (equivalence-biinvertible e))
        b)))
  (app
    (right-inverse-homotopy A B
      (equivalence-function e) (equivalence-biinvertible e))
    b))

lemma equivalence-contractible-paths-type:
  assumes A: A ::h universe i
    and B: B ::h universe i
    and e: e ::h equivalence A B
    and c: c ::h contractible A
  shows equivalence-contractible-paths A B e c ::h
    pi-type B (λb.
      identity-type B (equivalence-contractible-center A B e c) b)
  unfolding equivalence-contractible-paths-def
proof (rule pi-intro[OF B])
  fix b
  assume b: b ::h B
  have f: equivalence-function e ::h arrow A B
    by (rule equivalence-function-type[OF e])
  have w: equivalence-biinvertible e ::h
      biinvertible A B (equivalence-function e)
    by (rule equivalence-biinvertible-type[OF e])
  have center-A: contraction-center c ::h A
    by (rule contraction-center-type[OF c])
  have center-B: app (equivalence-function e) (contraction-center c)
      ::h B
    by (rule arrow-elim[OF f center-A])
  have inverse: right-inverse-function A B
      (equivalence-function e) (equivalence-biinvertible e)
      ::h arrow B A
    by (rule right-inverse-function-type[OF w])
  have inverse-b: app
```

```
    (right-inverse-function A B
      (equivalence-function e) (equivalence-biinvertible e))
    b ::_h A
  by (rule arrow-elim[OF inverse b])
have contraction: contraction-paths c ::_h
    pi-type A (λx. identity-type A (contraction-center c) x)
  by (rule contraction-paths-type[OF c])
have center-to-inverse: app (contraction-paths c)
    (app
      (right-inverse-function A B
        (equivalence-function e) (equivalence-biinvertible e))
      b) ::_h
    identity-type A
      (contraction-center c)
      (app
        (right-inverse-function A B
          (equivalence-function e) (equivalence-biinvertible e))
        b)
  by (rule pi-elim[OF contraction inverse-b])
have mapped: ap A B
    (equivalence-function e)
    (contraction-center c)
    (app
      (right-inverse-function A B
        (equivalence-function e) (equivalence-biinvertible e))
      b)
    (app (contraction-paths c)
      (app
        (right-inverse-function A B
          (equivalence-function e) (equivalence-biinvertible e))
        b)) ::_h
    identity-type B
      (app (equivalence-function e) (contraction-center c))
      (app (equivalence-function e)
        (app
          (right-inverse-function A B
            (equivalence-function e) (equivalence-biinvertible e))
          b))
  by (rule ap-type[OF A B f center-A inverse-b center-to-inverse])
have inverse-image: app (equivalence-function e)
    (app
      (right-inverse-function A B
        (equivalence-function e) (equivalence-biinvertible e))
      b) ::_h B
  by (rule arrow-elim[OF f inverse-b])
have right-homotopy: right-inverse-homotopy A B
    (equivalence-function e) (equivalence-biinvertible e) ::_h
    homotopy B (λ-. B)
      (compose B A B
```

```
      (equivalence-function e)
      (right-inverse-function A B
        (equivalence-function e) (equivalence-biinvertible e)))
    (identity-function B)
  by (rule right-inverse-homotopy-type[OF w])
have right-homotopy-pi: right-inverse-homotopy A B
    (equivalence-function e) (equivalence-biinvertible e) ::_h
    pi-type B (λx.
      identity-type B
        (app
          (compose B A B
            (equivalence-function e)
            (right-inverse-function A B
              (equivalence-function e) (equivalence-biinvertible e)))
          x)
        (app (identity-function B) x))
  using right-homotopy unfolding homotopy-def .
have right-path-raw: app
    (right-inverse-homotopy A B
      (equivalence-function e) (equivalence-biinvertible e))
    b ::_h
    identity-type B
      (app
        (compose B A B
          (equivalence-function e)
          (right-inverse-function A B
            (equivalence-function e) (equivalence-biinvertible e)))
        b)
      (app (identity-function B) b)
  by (rule pi-elim[OF right-homotopy-pi b])
have right-path: app
    (right-inverse-homotopy A B
      (equivalence-function e) (equivalence-biinvertible e))
    b ::_h
    identity-type B
      (app (equivalence-function e)
        (app
          (right-inverse-function A B
            (equivalence-function e) (equivalence-biinvertible e))
          b))
      b
  using right-path-raw by simp
show path-concat B
    (equivalence-contractible-center A B e c)
    (app (equivalence-function e)
      (app
        (right-inverse-function A B
          (equivalence-function e) (equivalence-biinvertible e))
        b))
```

```
      b
      (ap A B
        (equivalence-function e)
        (contraction-center c)
        (app
          (right-inverse-function A B
            (equivalence-function e) (equivalence-biinvertible e))
          b)
        (app (contraction-paths c)
          (app
            (right-inverse-function A B
              (equivalence-function e) (equivalence-biinvertible e))
            b)))
      (app
        (right-inverse-homotopy A B
          (equivalence-function e) (equivalence-biinvertible e))
        b) ::_h
      identity-type B (equivalence-contractible-center A B e c) b
    unfolding equivalence-contractible-center-def
    by (rule path-concat-type[OF B center-B inverse-image b mapped right-path])
qed

definition equivalence-preserves-contractible ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr
where
  equivalence-preserves-contractible A B e c ≡
    pair
      (equivalence-contractible-center A B e c)
      (equivalence-contractible-paths A B e c)

theorem equivalence-preserves-contractible-type:
  assumes A: A ::_h universe i
    and B: B ::_h universe i
    and e: e ::_h equivalence A B
    and c: c ::_h contractible A
  shows equivalence-preserves-contractible A B e c ::_h
    contractible B
proof −
  have f: equivalence-function e ::_h arrow A B
    by (rule equivalence-function-type[OF e])
  have center-A: contraction-center c ::_h A
    by (rule contraction-center-type[OF c])
  have center-B: equivalence-contractible-center A B e c ::_h B
    unfolding equivalence-contractible-center-def
    by (rule arrow-elim[OF f center-A])
  have paths: equivalence-contractible-paths A B e c ::_h
      pi-type B (λb.
        identity-type B (equivalence-contractible-center A B e c) b)
    by (rule equivalence-contractible-paths-type[OF A B e c])
```

```
  show equivalence-preserves-contractible A B e c ::_h
      contractible B
    unfolding equivalence-preserves-contractible-def contractible-def
    by (rule sigma-intro[where B=
          λb. pi-type B
            (λx. identity-type B b x), OF center-B paths])
qed
```

28 From a bi-inverse to a single quasi-inverse

A bi-invertible map carries independent left and right inverses. We first bundle them into a single quasi-inverse: the left inverse g together with its retraction homotopy η is kept, and a compatible section homotopy $f \circ g \sim id$ is manufactured from the right inverse via $g \sim h$.

```
definition quasi-theta ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr
where
  quasi-theta A B f w b ≡
    path-concat A
      (app (left-inverse-function A B f w) b)
      (app (left-inverse-function A B f w)
        (app f (app (right-inverse-function A B f w) b)))
      (app (right-inverse-function A B f w) b)
      (ap B A (left-inverse-function A B f w) b
        (app f (app (right-inverse-function A B f w) b))
        (path-inverse B (app f (app (right-inverse-function A B f w) b)) b
          (app (right-inverse-homotopy A B f w) b)))
      (app (left-inverse-homotopy A B f w)
        (app (right-inverse-function A B f w) b))

lemma quasi-theta-type:
  assumes A: A ::_h universe i
    and B: B ::_h universe i
    and f: f ::_h arrow A B
    and w: w ::_h biinvertible A B f
    and b: b ::_h B
  shows quasi-theta A B f w b ::_h
    identity-type A
      (app (left-inverse-function A B f w) b)
      (app (right-inverse-function A B f w) b)
proof −
  have g: left-inverse-function A B f w ::_h arrow B A
    by (rule left-inverse-function-type[OF w])
  have h: right-inverse-function A B f w ::_h arrow B A
    by (rule right-inverse-function-type[OF w])
  have eta: left-inverse-homotopy A B f w ::_h
      homotopy A (λ-. A)
        (compose A B A (left-inverse-function A B f w) f)
```

```
    (identity-function A)
  by (rule left-inverse-homotopy-type[OF w])
have e0: right-inverse-homotopy A B f w ::_h
    homotopy B (λ-. B)
      (compose B A B f (right-inverse-function A B f w))
      (identity-function B)
  by (rule right-inverse-homotopy-type[OF w])
have hb: app (right-inverse-function A B f w) b ::_h A
  by (rule arrow-elim[OF h b])
have gb: app (left-inverse-function A B f w) b ::_h A
  by (rule arrow-elim[OF g b])
have fhb: app f (app (right-inverse-function A B f w) b) ::_h B
  by (rule arrow-elim[OF f hb])
have glifhb: app (left-inverse-function A B f w)
    (app f (app (right-inverse-function A B f w) b)) ::_h A
  by (rule arrow-elim[OF g fhb])
have e0pi: right-inverse-homotopy A B f w ::_h
    pi-type B (λx. identity-type B
      (app (compose B A B f (right-inverse-function A B f w)) x)
      (app (identity-function B) x))
  using e0 unfolding homotopy-def .
have e0b: app (right-inverse-homotopy A B f w) b ::_h
    identity-type B (app f (app (right-inverse-function A B f w) b)) b
  using pi-elim[OF e0pi b] by simp
have e0b-inv: path-inverse B
    (app f (app (right-inverse-function A B f w) b)) b
    (app (right-inverse-homotopy A B f w) b) ::_h
    identity-type B b (app f (app (right-inverse-function A B f w) b))
  by (rule path-inverse-type[OF B fhb b e0b])
have ap-g: ap B A (left-inverse-function A B f w) b
    (app f (app (right-inverse-function A B f w) b))
    (path-inverse B (app f (app (right-inverse-function A B f w) b)) b
      (app (right-inverse-homotopy A B f w) b)) ::_h
    identity-type A
      (app (left-inverse-function A B f w) b)
      (app (left-inverse-function A B f w)
        (app f (app (right-inverse-function A B f w) b)))
  by (rule ap-type[OF B A g b fhb e0b-inv])
have etapi: left-inverse-homotopy A B f w ::_h
    pi-type A (λx. identity-type A
      (app (compose A B A (left-inverse-function A B f w) f) x)
      (app (identity-function A) x))
  using eta unfolding homotopy-def .
have eta-hb: app (left-inverse-homotopy A B f w)
    (app (right-inverse-function A B f w) b) ::_h
    identity-type A
      (app (left-inverse-function A B f w)
        (app f (app (right-inverse-function A B f w) b)))
      (app (right-inverse-function A B f w) b)
```

```
    using pi-elim[OF etapi hb] by simp
  show quasi-theta A B f w b ::h
      identity-type A
        (app (left-inverse-function A B f w) b)
        (app (right-inverse-function A B f w) b)
    unfolding quasi-theta-def
    by (rule path-concat-type[OF A gb glifhb hb ap-g eta-hb])
qed

definition quasi-rho-point ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr
where
  quasi-rho-point A B f w b ≡
    path-concat B
      (app f (app (left-inverse-function A B f w) b))
      (app f (app (right-inverse-function A B f w) b))
      b
      (ap A B f
        (app (left-inverse-function A B f w) b)
        (app (right-inverse-function A B f w) b)
        (quasi-theta A B f w b))
      (app (right-inverse-homotopy A B f w) b)

lemma quasi-rho-point-type:
  assumes A: A ::h universe i
    and B: B ::h universe i
    and f: f ::h arrow A B
    and w: w ::h biinvertible A B f
    and b: b ::h B
  shows quasi-rho-point A B f w b ::h
    identity-type B (app f (app (left-inverse-function A B f w) b)) b
proof −
  have g: left-inverse-function A B f w ::h arrow B A
    by (rule left-inverse-function-type[OF w])
  have h: right-inverse-function A B f w ::h arrow B A
    by (rule right-inverse-function-type[OF w])
  have e0: right-inverse-homotopy A B f w ::h
      homotopy B (λ-. B)
        (compose B A B f (right-inverse-function A B f w))
        (identity-function B)
    by (rule right-inverse-homotopy-type[OF w])
  have hb: app (right-inverse-function A B f w) b ::h A
    by (rule arrow-elim[OF h b])
  have gb: app (left-inverse-function A B f w) b ::h A
    by (rule arrow-elim[OF g b])
  have fgb: app f (app (left-inverse-function A B f w) b) ::h B
    by (rule arrow-elim[OF f gb])
  have fhb: app f (app (right-inverse-function A B f w) b) ::h B
    by (rule arrow-elim[OF f hb])
```

```
  have theta: quasi-theta A B f w b ::_h
      identity-type A
        (app (left-inverse-function A B f w) b)
        (app (right-inverse-function A B f w) b)
    by (rule quasi-theta-type[OF A B f w b])
  have ap-f-theta: ap A B f
      (app (left-inverse-function A B f w) b)
      (app (right-inverse-function A B f w) b)
      (quasi-theta A B f w b) ::_h
      identity-type B
        (app f (app (left-inverse-function A B f w) b))
        (app f (app (right-inverse-function A B f w) b))
    by (rule ap-type[OF A B f gb hb theta])
  have e0pi: right-inverse-homotopy A B f w ::_h
      pi-type B (λx. identity-type B
        (app (compose B A B f (right-inverse-function A B f w)) x)
        (app (identity-function B) x))
    using e0 unfolding homotopy-def .
  have e0b: app (right-inverse-homotopy A B f w) b ::_h
      identity-type B (app f (app (right-inverse-function A B f w) b)) b
    using pi-elim[OF e0pi b] by simp
  show quasi-rho-point A B f w b ::_h
      identity-type B (app f (app (left-inverse-function A B f w) b)) b
    unfolding quasi-rho-point-def
    by (rule path-concat-type[OF B fgb fhb b ap-f-theta e0b])
qed

definition quasi-rho ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr
where
  quasi-rho A B f w ≡
    lambda B (λb. quasi-rho-point A B f w b)

lemma quasi-rho-type:
  assumes A: A ::_h universe i
    and B: B ::_h universe i
    and f: f ::_h arrow A B
    and w: w ::_h biinvertible A B f
  shows quasi-rho A B f w ::_h
    homotopy B (λ-. B)
      (compose B A B f (left-inverse-function A B f w))
      (identity-function B)
  unfolding quasi-rho-def homotopy-def
proof (rule pi-intro[OF B])
  fix b
  assume b: b ::_h B
  have g: left-inverse-function A B f w ::_h arrow B A
    by (rule left-inverse-function-type[OF w])
  have point: quasi-rho-point A B f w b ::_h
```

```
    identity-type B (app f (app (left-inverse-function A B f w) b)) b
  by (rule quasi-rho-point-type[OF A B f w b])
show quasi-rho-point A B f w b ::h
    identity-type B
      (app (compose B A B f (left-inverse-function A B f w)) b)
      (app (identity-function B) b)
  using point by simp
qed

lemma quasi-rho-beta [simp]:
  app (quasi-rho A B f w) b ≡ quasi-rho-point A B f w b
  unfolding quasi-rho-def by simp
```

29 Coherent (half-adjoint) inverse data

The quasi-inverse $(g,\ \eta,\ \varrho)$ is upgraded to half-adjoint data by the standard eta trick: a coherent section homotopy ε is defined so that the triangle identity $ap\ f\ (\eta\ a) = \varepsilon\ (f\ a)$ holds.

```
definition coherent-eps-point ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr
where
  coherent-eps-point A B f w b ≡
    path-concat B
      (app f (app (left-inverse-function A B f w) b))
      (app f (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) b))))
      b
      (path-inverse B
        (app f (app (left-inverse-function A B f w)
          (app f (app (left-inverse-function A B f w) b))))
        (app f (app (left-inverse-function A B f w) b))
        (quasi-rho-point A B f w
          (app f (app (left-inverse-function A B f w) b))))
      (path-concat B
        (app f (app (left-inverse-function A B f w)
          (app f (app (left-inverse-function A B f w) b))))
        (app f (app (left-inverse-function A B f w) b))
        b
        (ap A B f
          (app (left-inverse-function A B f w)
            (app f (app (left-inverse-function A B f w) b)))
          (app (left-inverse-function A B f w) b)
          (app (left-inverse-homotopy A B f w)
            (app (left-inverse-function A B f w) b)))
        (quasi-rho-point A B f w b))

lemma coherent-eps-point-type:
  assumes A: A ::h universe i
```

```
    and B: B ::_h universe i
    and f: f ::_h arrow A B
    and w: w ::_h biinvertible A B f
    and b: b ::_h B
  shows coherent-eps-point A B f w b ::_h
    identity-type B (app f (app (left-inverse-function A B f w) b)) b
proof −
  have g: left-inverse-function A B f w ::_h arrow B A
    by (rule left-inverse-function-type[OF w])
  have eta: left-inverse-homotopy A B f w ::_h
      homotopy A (λ-. A)
        (compose A B A (left-inverse-function A B f w) f)
        (identity-function A)
    by (rule left-inverse-homotopy-type[OF w])
  have gb: app (left-inverse-function A B f w) b ::_h A
    by (rule arrow-elim[OF g b])
  have fgb: app f (app (left-inverse-function A B f w) b) ::_h B
    by (rule arrow-elim[OF f gb])
  have gfgb: app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) b)) ::_h A
    by (rule arrow-elim[OF g fgb])
  have fgfgb: app f (app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) b))) ::_h B
    by (rule arrow-elim[OF f gfgb])
  have rho-b: quasi-rho-point A B f w b ::_h
      identity-type B (app f (app (left-inverse-function A B f w) b)) b
    by (rule quasi-rho-point-type[OF A B f w b])
  have rho-fgb: quasi-rho-point A B f w
      (app f (app (left-inverse-function A B f w) b)) ::_h
      identity-type B
        (app f (app (left-inverse-function A B f w)
          (app f (app (left-inverse-function A B f w) b))))
        (app f (app (left-inverse-function A B f w) b))
    by (rule quasi-rho-point-type[OF A B f w fgb])
  have rho-fgb-inv: path-inverse B
      (app f (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) b))))
      (app f (app (left-inverse-function A B f w) b))
      (quasi-rho-point A B f w
        (app f (app (left-inverse-function A B f w) b))) ::_h
      identity-type B
        (app f (app (left-inverse-function A B f w) b))
        (app f (app (left-inverse-function A B f w)
          (app f (app (left-inverse-function A B f w) b))))
    by (rule path-inverse-type[OF B fgfgb fgb rho-fgb])
  have etapi: left-inverse-homotopy A B f w ::_h
      pi-type A (λx. identity-type A
        (app (compose A B A (left-inverse-function A B f w) f) x)
        (app (identity-function A) x))
```

```
    using eta unfolding homotopy-def .
  have eta-gb: app (left-inverse-homotopy A B f w)
      (app (left-inverse-function A B f w) b) ::_h
      identity-type A
        (app (left-inverse-function A B f w)
          (app f (app (left-inverse-function A B f w) b)))
        (app (left-inverse-function A B f w) b)
    using pi-elim[OF etapi gb] by simp
  have apf-eta: ap A B f
      (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) b)))
      (app (left-inverse-function A B f w) b)
      (app (left-inverse-homotopy A B f w)
        (app (left-inverse-function A B f w) b)) ::_h
      identity-type B
        (app f (app (left-inverse-function A B f w)
          (app f (app (left-inverse-function A B f w) b))))
        (app f (app (left-inverse-function A B f w) b))
    by (rule ap-type[OF A B f gfgb gb eta-gb])
  have inner: path-concat B
      (app f (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) b))))
      (app f (app (left-inverse-function A B f w) b))
      b
      (ap A B f
        (app (left-inverse-function A B f w)
          (app f (app (left-inverse-function A B f w) b)))
        (app (left-inverse-function A B f w) b)
        (app (left-inverse-homotopy A B f w)
          (app (left-inverse-function A B f w) b)))
      (quasi-rho-point A B f w b) ::_h
      identity-type B
        (app f (app (left-inverse-function A B f w)
          (app f (app (left-inverse-function A B f w) b))))
        b
    by (rule path-concat-type[OF B fgfgb fgb b apf-eta rho-b])
  show coherent-eps-point A B f w b ::_h
      identity-type B (app f (app (left-inverse-function A B f w) b)) b
    unfolding coherent-eps-point-def
    by (rule path-concat-type[OF B fgb fgfgb b rho-fgb-inv inner])
qed
```

Auxiliary two-cell *coherent-D*: the functorial image $ap\ (f \circ g)\ (ap\ f\ (\eta\ a))$ agrees with $ap\ f\ (\eta\ (g\ (f\ a)))$. This uses the self-action *homotopy-self* of η transported through *ap f*.

```
definition coherent-D ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr
where
  coherent-D A B f w a ≡
```

```
path-concat
  (identity-type B
    (app f (app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a)))))
    (app f (app (left-inverse-function A B f w) (app f a))))
  (ap B B (compose B A B f (left-inverse-function A B f w))
    (app f (app (left-inverse-function A B f w) (app f a)))
    (app f a)
    (ap A B f (app (left-inverse-function A B f w) (app f a)) a
      (app (left-inverse-homotopy A B f w) a)))
  (ap A B f
    (app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a))))
    (app (left-inverse-function A B f w) (app f a))
    (ap B A (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a)))
      (app f a)
      (ap A B f (app (left-inverse-function A B f w) (app f a)) a
        (app (left-inverse-homotopy A B f w) a))))
  (ap A B f
    (app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a))))
    (app (left-inverse-function A B f w) (app f a))
    (app (left-inverse-homotopy A B f w)
      (app (left-inverse-function A B f w) (app f a))))
  (ap-compose B A B f (left-inverse-function A B f w)
    (app f (app (left-inverse-function A B f w) (app f a)))
    (app f a)
    (ap A B f (app (left-inverse-function A B f w) (app f a)) a
      (app (left-inverse-homotopy A B f w) a)))
  (ap-congruence A B f
    (app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a))))
    (app (left-inverse-function A B f w) (app f a))
    (ap B A (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a)))
      (app f a)
      (ap A B f (app (left-inverse-function A B f w) (app f a)) a
        (app (left-inverse-homotopy A B f w) a)))
    (app (left-inverse-homotopy A B f w)
      (app (left-inverse-function A B f w) (app f a)))
    (path-concat
      (identity-type A
        (app (left-inverse-function A B f w)
          (app f (app (left-inverse-function A B f w) (app f a))))
        (app (left-inverse-function A B f w) (app f a)))
      (ap B A (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a)))
        (app f a)
```

```
    (ap A B f (app (left-inverse-function A B f w) (app f a)) a
      (app (left-inverse-homotopy A B f w) a)))
   (ap A A (compose A B A (left-inverse-function A B f w) f)
    (app (left-inverse-function A B f w) (app f a)) a
    (app (left-inverse-homotopy A B f w) a))
   (app (left-inverse-homotopy A B f w)
    (app (left-inverse-function A B f w) (app f a)))
   (path-inverse
    (identity-type A
     (app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a))))
     (app (left-inverse-function A B f w) (app f a)))
    (ap A A (compose A B A (left-inverse-function A B f w) f)
     (app (left-inverse-function A B f w) (app f a)) a
     (app (left-inverse-homotopy A B f w) a))
    (ap B A (left-inverse-function A B f w)
     (app f (app (left-inverse-function A B f w) (app f a)))
     (app f a)
     (ap A B f (app (left-inverse-function A B f w) (app f a)) a
      (app (left-inverse-homotopy A B f w) a)))
    (ap-compose A B A (left-inverse-function A B f w) f
     (app (left-inverse-function A B f w) (app f a)) a
     (app (left-inverse-homotopy A B f w) a)))
   (homotopy-self A (compose A B A (left-inverse-function A B f w) f)
    (left-inverse-homotopy A B f w) a)))

lemma coherent-D-type:
 assumes A: A ::_h universe i
  and B: B ::_h universe i
  and f: f ::_h arrow A B
  and w: w ::_h biinvertible A B f
  and a: a ::_h A
 shows coherent-D A B f w a ::_h
  identity-type
   (identity-type B
    (app f (app (left-inverse-function A B f w)
     (app f (app (left-inverse-function A B f w) (app f a)))))
    (app f (app (left-inverse-function A B f w) (app f a))))
   (ap B B (compose B A B f (left-inverse-function A B f w))
    (app f (app (left-inverse-function A B f w) (app f a)))
    (app f a)
    (ap A B f (app (left-inverse-function A B f w) (app f a)) a
     (app (left-inverse-homotopy A B f w) a)))
   (ap A B f
    (app (left-inverse-function A B f w)
     (app f (app (left-inverse-function A B f w) (app f a))))
    (app (left-inverse-function A B f w) (app f a))
    (app (left-inverse-homotopy A B f w)
     (app (left-inverse-function A B f w) (app f a))))
```

proof −
have *g*: *left-inverse-function A B f w* ::$_h$ *arrow B A*
by (*rule left-inverse-function-type*[*OF w*])
have *eta*: *left-inverse-homotopy A B f w* ::$_h$
homotopy A (λ-. *A*)
(*compose A B A* (*left-inverse-function A B f w*) *f*)
(*identity-function A*)
by (*rule left-inverse-homotopy-type*[*OF w*])
have *gf*: *compose A B A* (*left-inverse-function A B f w*) *f* ::$_h$ *arrow A A*
by (*rule compose-type*[*OF A f g*])
have *fg*: *compose B A B f* (*left-inverse-function A B f w*) ::$_h$ *arrow B B*
by (*rule compose-type*[*OF B g f*])
have *fa*: *app f a* ::$_h$ *B* **by** (*rule arrow-elim*[*OF f a*])
have *gfa*: *app* (*left-inverse-function A B f w*) (*app f a*) ::$_h$ *A*
by (*rule arrow-elim*[*OF g fa*])
have *fgfa*: *app f* (*app* (*left-inverse-function A B f w*) (*app f a*)) ::$_h$ *B*
by (*rule arrow-elim*[*OF f gfa*])
have *gfgfa*: *app* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*))) ::$_h$ *A*
by (*rule arrow-elim*[*OF g fgfa*])
have *fgfgfa*: *app f* (*app* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))) ::$_h$ *B*
by (*rule arrow-elim*[*OF f gfgfa*])
have *etapi*: *left-inverse-homotopy A B f w* ::$_h$
pi-type A (λ*x*. *identity-type A*
(*app* (*compose A B A* (*left-inverse-function A B f w*) *f*) *x*)
(*app* (*identity-function A*) *x*))
using *eta* **unfolding** *homotopy-def* **.**
have *eta-a*: *app* (*left-inverse-homotopy A B f w*) *a* ::$_h$
identity-type A (*app* (*left-inverse-function A B f w*) (*app f a*)) *a*
using *pi-elim*[*OF etapi a*] **by** *simp*
have *s*: *ap A B f* (*app* (*left-inverse-function A B f w*) (*app f a*)) *a*
(*app* (*left-inverse-homotopy A B f w*) *a*) ::$_h$
identity-type B (*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))
(*app f a*)
by (*rule ap-type*[*OF A B f gfa a eta-a*])
have *Lp*: *ap B B* (*compose B A B f* (*left-inverse-function A B f w*))
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))
(*app f a*)
(*ap A B f* (*app* (*left-inverse-function A B f w*) (*app f a*)) *a*
(*app* (*left-inverse-homotopy A B f w*) *a*)) ::$_h$
identity-type B
(*app f* (*app* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))))
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))
using *ap-type*[*OF B B fg fgfa fa s*] **by** *simp*
have *apg-s*: *ap B A* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))
(*app f a*)

```
    (ap A B f (app (left-inverse-function A B f w) (app f a)) a
      (app (left-inverse-homotopy A B f w) a)) ::h
    identity-type A
      (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a))))
      (app (left-inverse-function A B f w) (app f a))
  by (rule ap-type[OF B A g fgfa fa s])
have eta-gfa: app (left-inverse-homotopy A B f w)
    (app (left-inverse-function A B f w) (app f a)) ::h
    identity-type A
      (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a))))
      (app (left-inverse-function A B f w) (app f a))
  using pi-elim[OF etapi gfa] by simp
have mid: ap A B f
    (app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a))))
    (app (left-inverse-function A B f w) (app f a))
    (ap B A (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a)))
      (app f a)
      (ap A B f (app (left-inverse-function A B f w) (app f a)) a
        (app (left-inverse-homotopy A B f w) a))) ::h
    identity-type B
      (app f (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a)))))
      (app f (app (left-inverse-function A B f w) (app f a)))
  by (rule ap-type[OF A B f gfgfa gfa apg-s])
have apfeta: ap A B f
    (app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a))))
    (app (left-inverse-function A B f w) (app f a))
    (app (left-inverse-homotopy A B f w)
      (app (left-inverse-function A B f w) (app f a))) ::h
    identity-type B
      (app f (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a)))))
      (app f (app (left-inverse-function A B f w) (app f a)))
  by (rule ap-type[OF A B f gfgfa gfa eta-gfa])
have gspace: identity-type A
    (app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a))))
    (app (left-inverse-function A B f w) (app f a)) ::h universe i
  by (rule identity-form[OF A gfgfa gfa])
have apgf: ap A A (compose A B A (left-inverse-function A B f w) f)
    (app (left-inverse-function A B f w) (app f a)) a
    (app (left-inverse-homotopy A B f w) a) ::h
    identity-type A
      (app (left-inverse-function A B f w)
```

```
      (app f (app (left-inverse-function A B f w) (app f a))))
    (app (left-inverse-function A B f w) (app f a))
  using ap-type[OF A A gf gfa a eta-a] by simp
have AC2: ap-compose A B A (left-inverse-function A B f w) f
    (app (left-inverse-function A B f w) (app f a)) a
    (app (left-inverse-homotopy A B f w) a) ::_h
    identity-type
      (identity-type A
        (app (left-inverse-function A B f w)
          (app f (app (left-inverse-function A B f w) (app f a))))
        (app (left-inverse-function A B f w) (app f a)))
      (ap A A (compose A B A (left-inverse-function A B f w) f)
        (app (left-inverse-function A B f w) (app f a)) a
        (app (left-inverse-homotopy A B f w) a))
      (ap B A (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a)))
        (app f a)
        (ap A B f (app (left-inverse-function A B f w) (app f a)) a
          (app (left-inverse-homotopy A B f w) a)))
  by (rule ap-compose-type[OF A B A f g gfa a eta-a])
have AC2inv: path-inverse
    (identity-type A
      (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a))))
      (app (left-inverse-function A B f w) (app f a)))
    (ap A A (compose A B A (left-inverse-function A B f w) f)
      (app (left-inverse-function A B f w) (app f a)) a
      (app (left-inverse-homotopy A B f w) a))
    (ap B A (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a)))
      (app f a)
      (ap A B f (app (left-inverse-function A B f w) (app f a)) a
        (app (left-inverse-homotopy A B f w) a)))
    (ap-compose A B A (left-inverse-function A B f w) f
      (app (left-inverse-function A B f w) (app f a)) a
      (app (left-inverse-homotopy A B f w) a)) ::_h
    identity-type
      (identity-type A
        (app (left-inverse-function A B f w)
          (app f (app (left-inverse-function A B f w) (app f a))))
        (app (left-inverse-function A B f w) (app f a)))
      (ap B A (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a)))
        (app f a)
        (ap A B f (app (left-inverse-function A B f w) (app f a)) a
          (app (left-inverse-homotopy A B f w) a)))
      (ap A A (compose A B A (left-inverse-function A B f w) f)
        (app (left-inverse-function A B f w) (app f a)) a
        (app (left-inverse-homotopy A B f w) a))
```

by (*rule path-inverse-type*[*OF gspace apgf apg-s AC2*])
have *SELF*: *homotopy-self A* (*compose A B A* (*left-inverse-function A B f w*) *f*)
(*left-inverse-homotopy A B f w*) *a* ::$_h$
identity-type
(*identity-type A*
(*app* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*))))
(*app* (*left-inverse-function A B f w*) (*app f a*)))
(*ap A A* (*compose A B A* (*left-inverse-function A B f w*) *f*)
(*app* (*left-inverse-function A B f w*) (*app f a*)) *a*
(*app* (*left-inverse-homotopy A B f w*) *a*))
(*app* (*left-inverse-homotopy A B f w*)
(*app* (*left-inverse-function A B f w*) (*app f a*)))
using *homotopy-self-type*[*OF A gf eta a*] **by** *simp*
have *GS*: *path-concat*
(*identity-type A*
(*app* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*))))
(*app* (*left-inverse-function A B f w*) (*app f a*)))
(*ap B A* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))
(*app f a*)
(*ap A B f* (*app* (*left-inverse-function A B f w*) (*app f a*)) *a*
(*app* (*left-inverse-homotopy A B f w*) *a*)))
(*ap A A* (*compose A B A* (*left-inverse-function A B f w*) *f*)
(*app* (*left-inverse-function A B f w*) (*app f a*)) *a*
(*app* (*left-inverse-homotopy A B f w*) *a*))
(*app* (*left-inverse-homotopy A B f w*)
(*app* (*left-inverse-function A B f w*) (*app f a*)))
(*path-inverse*
(*identity-type A*
(*app* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*))))
(*app* (*left-inverse-function A B f w*) (*app f a*)))
(*ap A A* (*compose A B A* (*left-inverse-function A B f w*) *f*)
(*app* (*left-inverse-function A B f w*) (*app f a*)) *a*
(*app* (*left-inverse-homotopy A B f w*) *a*))
(*ap B A* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))
(*app f a*)
(*ap A B f* (*app* (*left-inverse-function A B f w*) (*app f a*)) *a*
(*app* (*left-inverse-homotopy A B f w*) *a*)))
(*ap-compose A B A* (*left-inverse-function A B f w*) *f*
(*app* (*left-inverse-function A B f w*) (*app f a*)) *a*
(*app* (*left-inverse-homotopy A B f w*) *a*)))
(*homotopy-self A* (*compose A B A* (*left-inverse-function A B f w*) *f*)
(*left-inverse-homotopy A B f w*) *a*) ::$_h$
identity-type
(*identity-type A*

(*app* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*))))
(*app* (*left-inverse-function A B f w*) (*app f a*)))
(*ap B A* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))
(*app f a*)
(*ap A B f* (*app* (*left-inverse-function A B f w*) (*app f a*)) *a*
(*app* (*left-inverse-homotopy A B f w*) *a*)))
(*app* (*left-inverse-homotopy A B f w*)
(*app* (*left-inverse-function A B f w*) (*app f a*)))
by (*rule path-concat-type*[*OF gspace apg-s apgf eta-gfa AC2inv SELF*])
have *space*: *identity-type B*
(*app f* (*app* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))))
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))
$::_h$ *universe i*
by (*rule identity-form*[*OF B fgfgfa fgfa*])
have *AC1*: *ap-compose B A B f* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))
(*app f a*)
(*ap A B f* (*app* (*left-inverse-function A B f w*) (*app f a*)) *a*
(*app* (*left-inverse-homotopy A B f w*) *a*)) $::_h$
identity-type
(*identity-type B*
(*app f* (*app* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))))
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*))))
(*ap B B* (*compose B A B f* (*left-inverse-function A B f w*))
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))
(*app f a*)
(*ap A B f* (*app* (*left-inverse-function A B f w*) (*app f a*)) *a*
(*app* (*left-inverse-homotopy A B f w*) *a*)))
(*ap A B f*
(*app* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*))))
(*app* (*left-inverse-function A B f w*) (*app f a*))
(*ap B A* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))
(*app f a*)
(*ap A B f* (*app* (*left-inverse-function A B f w*) (*app f a*)) *a*
(*app* (*left-inverse-homotopy A B f w*) *a*))))
by (*rule ap-compose-type*[*OF B A B g f fgfa fa s*])
have *APGS*: *ap-congruence A B f*
(*app* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*))))
(*app* (*left-inverse-function A B f w*) (*app f a*))
(*ap B A* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))
(*app f a*)

$$
\begin{aligned}
&\quad (ap\ A\ B\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a))\ a \\
&\qquad (app\ (left\text{-}inverse\text{-}homotopy\ A\ B\ f\ w)\ a))) \\
&(app\ (left\text{-}inverse\text{-}homotopy\ A\ B\ f\ w) \\
&\quad (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a))) \\
&(path\text{-}concat \\
&\quad (identity\text{-}type\ A \\
&\qquad (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w) \\
&\qquad\quad (app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))) \\
&\qquad (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a))) \\
&\quad (ap\ B\ A\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w) \\
&\qquad (app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a))) \\
&\qquad (app\ f\ a) \\
&\qquad (ap\ A\ B\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a))\ a \\
&\qquad\quad (app\ (left\text{-}inverse\text{-}homotopy\ A\ B\ f\ w)\ a))) \\
&\quad (ap\ A\ A\ (compose\ A\ B\ A\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ f) \\
&\qquad (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a))\ a \\
&\qquad (app\ (left\text{-}inverse\text{-}homotopy\ A\ B\ f\ w)\ a)) \\
&\quad (app\ (left\text{-}inverse\text{-}homotopy\ A\ B\ f\ w) \\
&\qquad (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a))) \\
&\quad (path\text{-}inverse \\
&\qquad (identity\text{-}type\ A \\
&\qquad\quad (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w) \\
&\qquad\qquad (app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))) \\
&\qquad\quad (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a))) \\
&\qquad (ap\ A\ A\ (compose\ A\ B\ A\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ f) \\
&\qquad\quad (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a))\ a \\
&\qquad\quad (app\ (left\text{-}inverse\text{-}homotopy\ A\ B\ f\ w)\ a)) \\
&\qquad (ap\ B\ A\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w) \\
&\qquad\quad (app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a))) \\
&\qquad\quad (app\ f\ a) \\
&\qquad\quad (ap\ A\ B\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a))\ a \\
&\qquad\qquad (app\ (left\text{-}inverse\text{-}homotopy\ A\ B\ f\ w)\ a))) \\
&\qquad (ap\text{-}compose\ A\ B\ A\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ f \\
&\qquad\quad (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a))\ a \\
&\qquad\quad (app\ (left\text{-}inverse\text{-}homotopy\ A\ B\ f\ w)\ a))) \\
&\quad (homotopy\text{-}self\ A\ (compose\ A\ B\ A\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ f) \\
&\qquad (left\text{-}inverse\text{-}homotopy\ A\ B\ f\ w)\ a))\ ::_h \\
&identity\text{-}type \\
&\quad (identity\text{-}type\ B \\
&\qquad (app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w) \\
&\qquad\quad (app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a))))) \\
&\qquad (app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))) \\
&\quad (ap\ A\ B\ f \\
&\qquad (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w) \\
&\qquad\quad (app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))) \\
&\qquad (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)) \\
&\qquad (ap\ B\ A\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w) \\
&\qquad\quad (app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a))) \\
&\qquad\quad (app\ f\ a)
\end{aligned}
$$

```
          (ap A B f (app (left-inverse-function A B f w) (app f a)) a
            (app (left-inverse-homotopy A B f w) a))))
        (ap A B f
          (app (left-inverse-function A B f w)
            (app f (app (left-inverse-function A B f w) (app f a))))
          (app (left-inverse-function A B f w) (app f a))
          (app (left-inverse-homotopy A B f w)
            (app (left-inverse-function A B f w) (app f a))))
    by (rule ap-congruence-type[OF A B f gfgfa gfa apg-s eta-gfa GS])
  show coherent-D A B f w a ::h
      identity-type
        (identity-type B
          (app f (app (left-inverse-function A B f w)
            (app f (app (left-inverse-function A B f w) (app f a)))))
          (app f (app (left-inverse-function A B f w) (app f a))))
        (ap B B (compose B A B f (left-inverse-function A B f w))
          (app f (app (left-inverse-function A B f w) (app f a)))
          (app f a)
          (ap A B f (app (left-inverse-function A B f w) (app f a)) a
            (app (left-inverse-homotopy A B f w) a)))
        (ap A B f
          (app (left-inverse-function A B f w)
            (app f (app (left-inverse-function A B f w) (app f a))))
          (app (left-inverse-function A B f w) (app f a))
          (app (left-inverse-homotopy A B f w)
            (app (left-inverse-function A B f w) (app f a))))
    unfolding coherent-D-def
    by (rule path-concat-type[OF space Lp mid apfeta AC1 APGS])
qed
```

Auxiliary two-cell *coherent-E*: the naturality square of ϱ at the path $ap\ f\ (\eta\ a)$.

```
definition coherent-E ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr
where
  coherent-E A B f w a ≡
    path-concat
      (identity-type B
        (app f (app (left-inverse-function A B f w)
          (app f (app (left-inverse-function A B f w) (app f a)))))
        (app f a))
      (path-concat B
        (app f (app (left-inverse-function A B f w)
          (app f (app (left-inverse-function A B f w) (app f a)))))
        (app f (app (left-inverse-function A B f w) (app f a)))
        (app f a)
        (ap B B (compose B A B f (left-inverse-function A B f w))
          (app f (app (left-inverse-function A B f w) (app f a)))
          (app f a)
```

```
    (ap A B f (app (left-inverse-function A B f w) (app f a)) a
      (app (left-inverse-homotopy A B f w) a)))
  (quasi-rho-point A B f w (app f a)))
(path-concat B
  (app f (app (left-inverse-function A B f w)
    (app f (app (left-inverse-function A B f w) (app f a)))))
  (app f (app (left-inverse-function A B f w) (app f a)))
  (app f a)
  (quasi-rho-point A B f w
    (app f (app (left-inverse-function A B f w) (app f a))))
  (ap B B (identity-function B)
    (app f (app (left-inverse-function A B f w) (app f a)))
    (app f a)
    (ap A B f (app (left-inverse-function A B f w) (app f a)) a
      (app (left-inverse-homotopy A B f w) a))))
(path-concat B
  (app f (app (left-inverse-function A B f w)
    (app f (app (left-inverse-function A B f w) (app f a)))))
  (app f (app (left-inverse-function A B f w) (app f a)))
  (app f a)
  (quasi-rho-point A B f w
    (app f (app (left-inverse-function A B f w) (app f a))))
  (ap A B f (app (left-inverse-function A B f w) (app f a)) a
    (app (left-inverse-homotopy A B f w) a)))
(homotopy-naturality B B
  (compose B A B f (left-inverse-function A B f w))
  (identity-function B)
  (quasi-rho A B f w)
  (app f (app (left-inverse-function A B f w) (app f a)))
  (app f a)
  (ap A B f (app (left-inverse-function A B f w) (app f a)) a
    (app (left-inverse-homotopy A B f w) a)))
(path-whisker-left B
  (app f (app (left-inverse-function A B f w)
    (app f (app (left-inverse-function A B f w) (app f a)))))
  (app f (app (left-inverse-function A B f w) (app f a)))
  (app f a)
  (quasi-rho-point A B f w
    (app f (app (left-inverse-function A B f w) (app f a))))
  (ap B B (identity-function B)
    (app f (app (left-inverse-function A B f w) (app f a)))
    (app f a)
    (ap A B f (app (left-inverse-function A B f w) (app f a)) a
      (app (left-inverse-homotopy A B f w) a)))
  (ap A B f (app (left-inverse-function A B f w) (app f a)) a
    (app (left-inverse-homotopy A B f w) a))
  (ap-identity B
    (app f (app (left-inverse-function A B f w) (app f a)))
    (app f a)
```

```
      (ap A B f (app (left-inverse-function A B f w) (app f a)) a
        (app (left-inverse-homotopy A B f w) a))))

lemma coherent-E-type:
  assumes A: A ::_h universe i
    and B: B ::_h universe i
    and f: f ::_h arrow A B
    and w: w ::_h biinvertible A B f
    and a: a ::_h A
  shows coherent-E A B f w a ::_h
    identity-type
      (identity-type B
        (app f (app (left-inverse-function A B f w)
          (app f (app (left-inverse-function A B f w) (app f a)))))
        (app f a))
      (path-concat B
        (app f (app (left-inverse-function A B f w)
          (app f (app (left-inverse-function A B f w) (app f a)))))
        (app f (app (left-inverse-function A B f w) (app f a)))
        (app f a)
        (ap B B (compose B A B f (left-inverse-function A B f w))
          (app f (app (left-inverse-function A B f w) (app f a)))
          (app f a)
          (ap A B f (app (left-inverse-function A B f w) (app f a)) a
            (app (left-inverse-homotopy A B f w) a)))
        (quasi-rho-point A B f w (app f a)))
      (path-concat B
        (app f (app (left-inverse-function A B f w)
          (app f (app (left-inverse-function A B f w) (app f a)))))
        (app f (app (left-inverse-function A B f w) (app f a)))
        (app f a)
        (quasi-rho-point A B f w
          (app f (app (left-inverse-function A B f w) (app f a))))
        (ap A B f (app (left-inverse-function A B f w) (app f a)) a
          (app (left-inverse-homotopy A B f w) a)))
proof −
  have g: left-inverse-function A B f w ::_h arrow B A
    by (rule left-inverse-function-type[OF w])
  have eta: left-inverse-homotopy A B f w ::_h
      homotopy A (λ-. A)
        (compose A B A (left-inverse-function A B f w) f)
        (identity-function A)
    by (rule left-inverse-homotopy-type[OF w])
  have fg: compose B A B f (left-inverse-function A B f w) ::_h arrow B B
    by (rule compose-type[OF B g f])
  have idB: identity-function B ::_h arrow B B
    by (rule identity-function-type[OF B])
  have rho-hom: quasi-rho A B f w ::_h
      homotopy B (λ-. B)
```

(*compose B A B f* (*left-inverse-function A B f w*))
(*identity-function B*)
by (*rule quasi-rho-type*[*OF A B f w*])
have *fa*: *app f a* ::$_h$ *B* **by** (*rule arrow-elim*[*OF f a*])
have *gfa*: *app* (*left-inverse-function A B f w*) (*app f a*) ::$_h$ *A*
by (*rule arrow-elim*[*OF g fa*])
have *fgfa*: *app f* (*app* (*left-inverse-function A B f w*) (*app f a*)) ::$_h$ *B*
by (*rule arrow-elim*[*OF f gfa*])
have *gfgfa*: *app* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*))) ::$_h$ *A*
by (*rule arrow-elim*[*OF g fgfa*])
have *fgfgfa*: *app f* (*app* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))) ::$_h$ *B*
by (*rule arrow-elim*[*OF f gfgfa*])
have *etapi*: *left-inverse-homotopy A B f w* ::$_h$
pi-type A (λ*x*. *identity-type A*
(*app* (*compose A B A* (*left-inverse-function A B f w*) *f*) *x*)
(*app* (*identity-function A*) *x*))
using *eta* **unfolding** *homotopy-def* **.**
have *eta-a*: *app* (*left-inverse-homotopy A B f w*) *a* ::$_h$
identity-type A (*app* (*left-inverse-function A B f w*) (*app f a*)) *a*
using *pi-elim*[*OF etapi a*] **by** *simp*
have *s*: *ap A B f* (*app* (*left-inverse-function A B f w*) (*app f a*)) *a*
(*app* (*left-inverse-homotopy A B f w*) *a*) ::$_h$
identity-type B (*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))
(*app f a*)
by (*rule ap-type*[*OF A B f gfa a eta-a*])
have *Lp*: *ap B B* (*compose B A B f* (*left-inverse-function A B f w*))
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))
(*app f a*)
(*ap A B f* (*app* (*left-inverse-function A B f w*) (*app f a*)) *a*
(*app* (*left-inverse-homotopy A B f w*) *a*)) ::$_h$
identity-type B
(*app f* (*app* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))))
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))
using *ap-type*[*OF B B fg fgfa fa s*] **by** *simp*
have *rho-fa*: *quasi-rho-point A B f w* (*app f a*) ::$_h$
identity-type B (*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))
(*app f a*)
by (*rule quasi-rho-point-type*[*OF A B f w fa*])
have *rho-fgfa*: *quasi-rho-point A B f w*
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*))) ::$_h$
identity-type B
(*app f* (*app* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))))
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))
by (*rule quasi-rho-point-type*[*OF A B f w fgfa*])
have *ap-idB-s*: *ap B B* (*identity-function B*)

$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))$
$(app\ f\ a)$
$(ap\ A\ B\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a))\ a$
$(app\ (left\text{-}inverse\text{-}homotopy\ A\ B\ f\ w)\ a))\ ::_h$
$identity\text{-}type\ B\ (app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))$
$(app\ f\ a)$
using $ap\text{-}type[OF\ B\ B\ idB\ fgfa\ fa\ s]$ **by** $simp$
have NAT: $homotopy\text{-}naturality\ B\ B$
$(compose\ B\ A\ B\ f\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w))$
$(identity\text{-}function\ B)$
$(quasi\text{-}rho\ A\ B\ f\ w)$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))$
$(app\ f\ a)$
$(ap\ A\ B\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a))\ a$
$(app\ (left\text{-}inverse\text{-}homotopy\ A\ B\ f\ w)\ a))\ ::_h$
$identity\text{-}type$
$(identity\text{-}type\ B$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))))$
$(app\ f\ a))$
$(path\text{-}concat\ B$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))))$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))$
$(app\ f\ a)$
$(ap\ B\ B\ (compose\ B\ A\ B\ f\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w))$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))$
$(app\ f\ a)$
$(ap\ A\ B\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a))\ a$
$(app\ (left\text{-}inverse\text{-}homotopy\ A\ B\ f\ w)\ a)))$
$(quasi\text{-}rho\text{-}point\ A\ B\ f\ w\ (app\ f\ a)))$
$(path\text{-}concat\ B$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))))$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))$
$(app\ f\ a)$
$(quasi\text{-}rho\text{-}point\ A\ B\ f\ w$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a))))$
$(ap\ B\ B\ (identity\text{-}function\ B)$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))$
$(app\ f\ a)$
$(ap\ A\ B\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a))\ a$
$(app\ (left\text{-}inverse\text{-}homotopy\ A\ B\ f\ w)\ a))))$
using $homotopy\text{-}naturality\text{-}type[OF\ B\ B\ fg\ idB\ rho\text{-}hom\ fgfa\ fa\ s]$ **by** $simp$
have $AI\text{-}E$: $ap\text{-}identity\ B$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))$
$(app\ f\ a)$
$(ap\ A\ B\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a))\ a$
$(app\ (left\text{-}inverse\text{-}homotopy\ A\ B\ f\ w)\ a))\ ::_h$

identity-type
(*identity-type B* (*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))
(*app f a*))
(*ap B B* (*identity-function B*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))
(*app f a*)
(*ap A B f* (*app* (*left-inverse-function A B f w*) (*app f a*)) *a*
(*app* (*left-inverse-homotopy A B f w*) *a*)))
(*ap A B f* (*app* (*left-inverse-function A B f w*) (*app f a*)) *a*
(*app* (*left-inverse-homotopy A B f w*) *a*))
by (*rule ap-identity-type*[*OF B fgfa fa s*])
have *WH-E*: *path-whisker-left B*
(*app f* (*app* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))))
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))
(*app f a*)
(*quasi-rho-point A B f w*
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*))))
(*ap B B* (*identity-function B*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))
(*app f a*)
(*ap A B f* (*app* (*left-inverse-function A B f w*) (*app f a*)) *a*
(*app* (*left-inverse-homotopy A B f w*) *a*)))
(*ap A B f* (*app* (*left-inverse-function A B f w*) (*app f a*)) *a*
(*app* (*left-inverse-homotopy A B f w*) *a*))
(*ap-identity B*
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))
(*app f a*)
(*ap A B f* (*app* (*left-inverse-function A B f w*) (*app f a*)) *a*
(*app* (*left-inverse-homotopy A B f w*) *a*))) $::_h$
identity-type
(*identity-type B*
(*app f* (*app* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))))
(*app f a*))
(*path-concat B*
(*app f* (*app* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))))
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))
(*app f a*)
(*quasi-rho-point A B f w*
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*))))
(*ap B B* (*identity-function B*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))
(*app f a*)
(*ap A B f* (*app* (*left-inverse-function A B f w*) (*app f a*)) *a*
(*app* (*left-inverse-homotopy A B f w*) *a*))))
(*path-concat B*
(*app f* (*app* (*left-inverse-function A B f w*)

(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))))
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))
(*app f a*)
(*quasi-rho-point A B f w*
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*))))
(*ap A B f* (*app* (*left-inverse-function A B f w*) (*app f a*)) *a*
(*app* (*left-inverse-homotopy A B f w*) *a*)))
by (*rule path-whisker-left-type*[*OF B fgfgfa fgfa fa rho-fgfa ap-idB-s s AI-E*])
have *Espace*: *identity-type B*
(*app f* (*app* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))))
(*app f a*) ::$_h$ *universe i*
by (*rule identity-form*[*OF B fgfgfa fa*])
have *Lrhofa*: *path-concat B*
(*app f* (*app* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))))
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))
(*app f a*)
(*ap B B* (*compose B A B f* (*left-inverse-function A B f w*))
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))
(*app f a*)
(*ap A B f* (*app* (*left-inverse-function A B f w*) (*app f a*)) *a*
(*app* (*left-inverse-homotopy A B f w*) *a*)))
(*quasi-rho-point A B f w* (*app f a*)) ::$_h$
identity-type B
(*app f* (*app* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))))
(*app f a*)
by (*rule path-concat-type*[*OF B fgfgfa fgfa fa Lp rho-fa*])
have *midE*: *path-concat B*
(*app f* (*app* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))))
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))
(*app f a*)
(*quasi-rho-point A B f w*
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*))))
(*ap B B* (*identity-function B*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))
(*app f a*)
(*ap A B f* (*app* (*left-inverse-function A B f w*) (*app f a*)) *a*
(*app* (*left-inverse-homotopy A B f w*) *a*))) ::$_h$
identity-type B
(*app f* (*app* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))))
(*app f a*)
by (*rule path-concat-type*[*OF B fgfgfa fgfa fa rho-fgfa ap-idB-s*])
have *rhofgfaS*: *path-concat B*
(*app f* (*app* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))))

```
    (app f (app (left-inverse-function A B f w) (app f a)))
    (app f a)
    (quasi-rho-point A B f w
      (app f (app (left-inverse-function A B f w) (app f a))))
    (ap A B f (app (left-inverse-function A B f w) (app f a)) a
      (app (left-inverse-homotopy A B f w) a)) ::h
    identity-type B
      (app f (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a)))))
      (app f a)
  by (rule path-concat-type[OF B fgfgfa fgfa fa rho-fgfa s])
show coherent-E A B f w a ::h
    identity-type
      (identity-type B
        (app f (app (left-inverse-function A B f w)
          (app f (app (left-inverse-function A B f w) (app f a)))))
        (app f a))
      (path-concat B
        (app f (app (left-inverse-function A B f w)
          (app f (app (left-inverse-function A B f w) (app f a)))))
        (app f (app (left-inverse-function A B f w) (app f a)))
        (app f a)
        (ap B B (compose B A B f (left-inverse-function A B f w))
          (app f (app (left-inverse-function A B f w) (app f a)))
          (app f a)
          (ap A B f (app (left-inverse-function A B f w) (app f a)) a
            (app (left-inverse-homotopy A B f w) a)))
        (quasi-rho-point A B f w (app f a)))
      (path-concat B
        (app f (app (left-inverse-function A B f w)
          (app f (app (left-inverse-function A B f w) (app f a)))))
        (app f (app (left-inverse-function A B f w) (app f a)))
        (app f a)
        (quasi-rho-point A B f w
          (app f (app (left-inverse-function A B f w) (app f a))))
        (ap A B f (app (left-inverse-function A B f w) (app f a)) a
          (app (left-inverse-homotopy A B f w) a)))
  unfolding coherent-E-def
  by (rule path-concat-type[OF Espace Lrhofa midE rhofgfaS NAT WH-E])
qed
```

Combining *coherent-D* and *coherent-E* gives the inner naturality cell $ap\ f\ (\eta\ (g\ (f\ a))) \cdot \varrho\ (f\ a) = \varrho\ (f\ (g\ (f\ a))) \cdot ap\ f\ (\eta\ a)$.

```
definition coherent-inner ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr
where
  coherent-inner A B f w a ≡
    path-concat
      (identity-type B
```

```
  (app f (app (left-inverse-function A B f w)
    (app f (app (left-inverse-function A B f w) (app f a)))))
  (app f a))
(path-concat B
  (app f (app (left-inverse-function A B f w)
    (app f (app (left-inverse-function A B f w) (app f a)))))
  (app f (app (left-inverse-function A B f w) (app f a)))
  (app f a)
  (ap A B f
    (app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a))))
    (app (left-inverse-function A B f w) (app f a))
    (app (left-inverse-homotopy A B f w)
      (app (left-inverse-function A B f w) (app f a))))
  (quasi-rho-point A B f w (app f a)))
(path-concat B
  (app f (app (left-inverse-function A B f w)
    (app f (app (left-inverse-function A B f w) (app f a)))))
  (app f (app (left-inverse-function A B f w) (app f a)))
  (app f a)
  (ap B B (compose B A B f (left-inverse-function A B f w))
    (app f (app (left-inverse-function A B f w) (app f a)))
    (app f a)
    (ap A B f (app (left-inverse-function A B f w) (app f a)) a
      (app (left-inverse-homotopy A B f w) a)))
  (quasi-rho-point A B f w (app f a)))
(path-concat B
  (app f (app (left-inverse-function A B f w)
    (app f (app (left-inverse-function A B f w) (app f a)))))
  (app f (app (left-inverse-function A B f w) (app f a)))
  (app f a)
  (quasi-rho-point A B f w
    (app f (app (left-inverse-function A B f w) (app f a))))
  (ap A B f (app (left-inverse-function A B f w) (app f a)) a
    (app (left-inverse-homotopy A B f w) a)))
(path-whisker-right B
  (app f (app (left-inverse-function A B f w)
    (app f (app (left-inverse-function A B f w) (app f a)))))
  (app f (app (left-inverse-function A B f w) (app f a)))
  (app f a)
  (ap A B f
    (app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a))))
    (app (left-inverse-function A B f w) (app f a))
    (app (left-inverse-homotopy A B f w)
      (app (left-inverse-function A B f w) (app f a))))
  (ap B B (compose B A B f (left-inverse-function A B f w))
    (app f (app (left-inverse-function A B f w) (app f a)))
    (app f a)
```

(*ap A B f* (*app* (*left-inverse-function A B f w*) (*app f a*)) *a*
(*app* (*left-inverse-homotopy A B f w*) *a*)))
(*quasi-rho-point A B f w* (*app f a*))
(*path-inverse*
(*identity-type B*
(*app f* (*app* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))))
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*))))
(*ap B B* (*compose B A B f* (*left-inverse-function A B f w*))
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))
(*app f a*)
(*ap A B f* (*app* (*left-inverse-function A B f w*) (*app f a*)) *a*
(*app* (*left-inverse-homotopy A B f w*) *a*)))
(*ap A B f*
(*app* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*))))
(*app* (*left-inverse-function A B f w*) (*app f a*))
(*app* (*left-inverse-homotopy A B f w*)
(*app* (*left-inverse-function A B f w*) (*app f a*))))
(*coherent-D A B f w a*)))
(*coherent-E A B f w a*)

lemma *coherent-inner-type*:
assumes *A*: $A ::_h$ *universe i*
and *B*: $B ::_h$ *universe i*
and *f*: $f ::_h$ *arrow A B*
and *w*: $w ::_h$ *biinvertible A B f*
and *a*: $a ::_h A$
shows *coherent-inner A B f w a* $::_h$
identity-type
(*identity-type B*
(*app f* (*app* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))))
(*app f a*))
(*path-concat B*
(*app f* (*app* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))))
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))
(*app f a*)
(*ap A B f*
(*app* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*))))
(*app* (*left-inverse-function A B f w*) (*app f a*))
(*app* (*left-inverse-homotopy A B f w*)
(*app* (*left-inverse-function A B f w*) (*app f a*))))
(*quasi-rho-point A B f w* (*app f a*)))
(*path-concat B*
(*app f* (*app* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))))

```
      (app f (app (left-inverse-function A B f w) (app f a)))
      (app f a)
      (quasi-rho-point A B f w
        (app f (app (left-inverse-function A B f w) (app f a))))
      (ap A B f (app (left-inverse-function A B f w) (app f a)) a
        (app (left-inverse-homotopy A B f w) a)))
proof −
  have g: left-inverse-function A B f w ::h arrow B A
    by (rule left-inverse-function-type[OF w])
  have eta: left-inverse-homotopy A B f w ::h
      homotopy A (λ-. A)
        (compose A B A (left-inverse-function A B f w) f)
        (identity-function A)
    by (rule left-inverse-homotopy-type[OF w])
  have fg: compose B A B f (left-inverse-function A B f w) ::h arrow B B
    by (rule compose-type[OF B g f])
  have fa: app f a ::h B by (rule arrow-elim[OF f a])
  have gfa: app (left-inverse-function A B f w) (app f a) ::h A
    by (rule arrow-elim[OF g fa])
  have fgfa: app f (app (left-inverse-function A B f w) (app f a)) ::h B
    by (rule arrow-elim[OF f gfa])
  have gfgfa: app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a))) ::h A
    by (rule arrow-elim[OF g fgfa])
  have fgfgfa: app f (app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a)))) ::h B
    by (rule arrow-elim[OF f gfgfa])
  have etapi: left-inverse-homotopy A B f w ::h
      pi-type A (λx. identity-type A
        (app (compose A B A (left-inverse-function A B f w) f) x)
        (app (identity-function A) x))
    using eta unfolding homotopy-def .
  have eta-a: app (left-inverse-homotopy A B f w) a ::h
      identity-type A (app (left-inverse-function A B f w) (app f a)) a
    using pi-elim[OF etapi a] by simp
  have eta-gfa: app (left-inverse-homotopy A B f w)
      (app (left-inverse-function A B f w) (app f a)) ::h
      identity-type A
        (app (left-inverse-function A B f w)
          (app f (app (left-inverse-function A B f w) (app f a))))
        (app (left-inverse-function A B f w) (app f a))
    using pi-elim[OF etapi gfa] by simp
  have s: ap A B f (app (left-inverse-function A B f w) (app f a)) a
      (app (left-inverse-homotopy A B f w) a) ::h
      identity-type B (app f (app (left-inverse-function A B f w) (app f a)))
        (app f a)
    by (rule ap-type[OF A B f gfa a eta-a])
  have Lp: ap B B (compose B A B f (left-inverse-function A B f w))
      (app f (app (left-inverse-function A B f w) (app f a)))
```

```
    (app f a)
    (ap A B f (app (left-inverse-function A B f w) (app f a)) a
      (app (left-inverse-homotopy A B f w) a)) ::_h
    identity-type B
      (app f (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a)))))
      (app f (app (left-inverse-function A B f w) (app f a)))
  using ap-type[OF B B fg fgfa fa s] by simp
have apfeta: ap A B f
    (app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a))))
    (app (left-inverse-function A B f w) (app f a))
    (app (left-inverse-homotopy A B f w)
      (app (left-inverse-function A B f w) (app f a))) ::_h
    identity-type B
      (app f (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a)))))
      (app f (app (left-inverse-function A B f w) (app f a)))
  by (rule ap-type[OF A B f gfgfa gfa eta-gfa])
have rho-fa: quasi-rho-point A B f w (app f a) ::_h
    identity-type B (app f (app (left-inverse-function A B f w) (app f a)))
      (app f a)
  by (rule quasi-rho-point-type[OF A B f w fa])
have rho-fgfa: quasi-rho-point A B f w
    (app f (app (left-inverse-function A B f w) (app f a))) ::_h
    identity-type B
      (app f (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a)))))
      (app f (app (left-inverse-function A B f w) (app f a)))
  by (rule quasi-rho-point-type[OF A B f w fgfa])
have space-D: identity-type B
    (app f (app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a)))))
    (app f (app (left-inverse-function A B f w) (app f a)))
    ::_h universe i
  by (rule identity-form[OF B fgfgfa fgfa])
have Dinv: path-inverse
    (identity-type B
      (app f (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a)))))
      (app f (app (left-inverse-function A B f w) (app f a))))
    (ap B B (compose B A B f (left-inverse-function A B f w))
      (app f (app (left-inverse-function A B f w) (app f a)))
      (app f a)
      (ap A B f (app (left-inverse-function A B f w) (app f a)) a
        (app (left-inverse-homotopy A B f w) a)))
    (ap A B f
      (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a)))))
```

```
    (app (left-inverse-function A B f w) (app f a))
    (app (left-inverse-homotopy A B f w)
      (app (left-inverse-function A B f w) (app f a))))
  (coherent-D A B f w a) ::_h
  identity-type
    (identity-type B
      (app f (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a)))))
      (app f (app (left-inverse-function A B f w) (app f a))))
    (ap A B f
      (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a))))
      (app (left-inverse-function A B f w) (app f a))
      (app (left-inverse-homotopy A B f w)
        (app (left-inverse-function A B f w) (app f a))))
    (ap B B (compose B A B f (left-inverse-function A B f w))
      (app f (app (left-inverse-function A B f w) (app f a)))
      (app f a)
      (ap A B f (app (left-inverse-function A B f w) (app f a)) a
        (app (left-inverse-homotopy A B f w) a)))
  by (rule path-inverse-type[OF space-D Lp apfeta coherent-D-type[OF A B f w
a]])
have WD: path-whisker-right B
  (app f (app (left-inverse-function A B f w)
    (app f (app (left-inverse-function A B f w) (app f a)))))
  (app f (app (left-inverse-function A B f w) (app f a)))
  (app f a)
  (ap A B f
    (app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a))))
    (app (left-inverse-function A B f w) (app f a))
    (app (left-inverse-homotopy A B f w)
      (app (left-inverse-function A B f w) (app f a))))
  (ap B B (compose B A B f (left-inverse-function A B f w))
    (app f (app (left-inverse-function A B f w) (app f a)))
    (app f a)
    (ap A B f (app (left-inverse-function A B f w) (app f a)) a
      (app (left-inverse-homotopy A B f w) a)))
  (quasi-rho-point A B f w (app f a))
  (path-inverse
    (identity-type B
      (app f (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a)))))
      (app f (app (left-inverse-function A B f w) (app f a))))
    (ap B B (compose B A B f (left-inverse-function A B f w))
      (app f (app (left-inverse-function A B f w) (app f a)))
      (app f a)
      (ap A B f (app (left-inverse-function A B f w) (app f a)) a
        (app (left-inverse-homotopy A B f w) a)))
```

```
    (ap A B f
      (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a))))
      (app (left-inverse-function A B f w) (app f a))
      (app (left-inverse-homotopy A B f w)
        (app (left-inverse-function A B f w) (app f a))))
    (coherent-D A B f w a)) ::_h
  identity-type
    (identity-type B
      (app f (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a)))))
      (app f a))
    (path-concat B
      (app f (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a)))))
      (app f (app (left-inverse-function A B f w) (app f a)))
      (app f a)
      (ap A B f
        (app (left-inverse-function A B f w)
          (app f (app (left-inverse-function A B f w) (app f a))))
        (app (left-inverse-function A B f w) (app f a))
        (app (left-inverse-homotopy A B f w)
          (app (left-inverse-function A B f w) (app f a))))
      (quasi-rho-point A B f w (app f a)))
    (path-concat B
      (app f (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a)))))
      (app f (app (left-inverse-function A B f w) (app f a)))
      (app f a)
      (ap B B (compose B A B f (left-inverse-function A B f w))
        (app f (app (left-inverse-function A B f w) (app f a)))
        (app f a)
        (ap A B f (app (left-inverse-function A B f w) (app f a)) a
          (app (left-inverse-homotopy A B f w) a)))
      (quasi-rho-point A B f w (app f a)))
  by (rule path-whisker-right-type[OF B fgfgfa fgfa fa apfeta Lp rho-fa Dinv])
have innerspace: identity-type B
    (app f (app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a)))))
    (app f a) ::_h universe i
  by (rule identity-form[OF B fgfgfa fa])
have apfeta-rhofa: path-concat B
    (app f (app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a)))))
    (app f (app (left-inverse-function A B f w) (app f a)))
    (app f a)
    (ap A B f
      (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a))))
```

```
      (app (left-inverse-function A B f w) (app f a))
      (app (left-inverse-homotopy A B f w)
        (app (left-inverse-function A B f w) (app f a))))
    (quasi-rho-point A B f w (app f a)) ::_h
    identity-type B
      (app f (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a)))))
      (app f a)
  by (rule path-concat-type[OF B fgfgfa fgfa fa apfeta rho-fa])
have L-rhofa: path-concat B
    (app f (app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a)))))
    (app f (app (left-inverse-function A B f w) (app f a)))
    (app f a)
    (ap B B (compose B A B f (left-inverse-function A B f w))
      (app f (app (left-inverse-function A B f w) (app f a)))
      (app f a)
      (ap A B f (app (left-inverse-function A B f w) (app f a)) a
        (app (left-inverse-homotopy A B f w) a)))
    (quasi-rho-point A B f w (app f a)) ::_h
    identity-type B
      (app f (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a)))))
      (app f a)
  by (rule path-concat-type[OF B fgfgfa fgfa fa Lp rho-fa])
have rhofgfa-s: path-concat B
    (app f (app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a)))))
    (app f (app (left-inverse-function A B f w) (app f a)))
    (app f a)
    (quasi-rho-point A B f w
      (app f (app (left-inverse-function A B f w) (app f a))))
    (ap A B f (app (left-inverse-function A B f w) (app f a)) a
      (app (left-inverse-homotopy A B f w) a)) ::_h
    identity-type B
      (app f (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a)))))
      (app f a)
  by (rule path-concat-type[OF B fgfgfa fgfa fa rho-fgfa s])
show coherent-inner A B f w a ::_h
    identity-type
      (identity-type B
        (app f (app (left-inverse-function A B f w)
          (app f (app (left-inverse-function A B f w) (app f a)))))
        (app f a))
      (path-concat B
        (app f (app (left-inverse-function A B f w)
          (app f (app (left-inverse-function A B f w) (app f a)))))
        (app f (app (left-inverse-function A B f w) (app f a)))
```

```
      (app f a)
      (ap A B f
        (app (left-inverse-function A B f w)
          (app f (app (left-inverse-function A B f w) (app f a))))
        (app (left-inverse-function A B f w) (app f a))
        (app (left-inverse-homotopy A B f w)
          (app (left-inverse-function A B f w) (app f a))))
      (quasi-rho-point A B f w (app f a)))
    (path-concat B
      (app f (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a)))))
      (app f (app (left-inverse-function A B f w) (app f a)))
      (app f a)
      (quasi-rho-point A B f w
        (app f (app (left-inverse-function A B f w) (app f a))))
      (ap A B f (app (left-inverse-function A B f w) (app f a)) a
        (app (left-inverse-homotopy A B f w) a)))
  unfolding coherent-inner-def
  by (rule path-concat-type[OF innerspace apfeta-rhofa L-rhofa rhofgfa-s
      WD coherent-E-type[OF A B f w a]])
qed
```

The half-adjoint triangle identity: $ap\ f\ (\eta\ a) = \varepsilon\ (f\ a)$.

```
definition coherent-tau ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr
where
  coherent-tau A B f w a ≡
    path-inverse
      (identity-type B (app f (app (left-inverse-function A B f w) (app f a)))
        (app f a))
      (path-concat B
        (app f (app (left-inverse-function A B f w) (app f a)))
        (app f (app (left-inverse-function A B f w)
          (app f (app (left-inverse-function A B f w) (app f a)))))
        (app f a)
        (path-inverse B
          (app f (app (left-inverse-function A B f w)
            (app f (app (left-inverse-function A B f w) (app f a)))))
          (app f (app (left-inverse-function A B f w) (app f a)))
          (quasi-rho-point A B f w
            (app f (app (left-inverse-function A B f w) (app f a)))))
        (path-concat B
          (app f (app (left-inverse-function A B f w)
            (app f (app (left-inverse-function A B f w) (app f a)))))
          (app f (app (left-inverse-function A B f w) (app f a)))
          (app f a)
          (ap A B f
            (app (left-inverse-function A B f w)
              (app f (app (left-inverse-function A B f w) (app f a))))
```

```
      (app (left-inverse-function A B f w) (app f a))
      (app (left-inverse-homotopy A B f w)
        (app (left-inverse-function A B f w) (app f a))))
    (quasi-rho-point A B f w (app f a))))
(ap A B f (app (left-inverse-function A B f w) (app f a)) a
  (app (left-inverse-homotopy A B f w) a))
(path-concat
  (identity-type B (app f (app (left-inverse-function A B f w) (app f a)))
    (app f a))
  (path-concat B
    (app f (app (left-inverse-function A B f w) (app f a)))
    (app f (app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a)))))
    (app f a)
    (path-inverse B
      (app f (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a)))))
      (app f (app (left-inverse-function A B f w) (app f a)))
      (quasi-rho-point A B f w
        (app f (app (left-inverse-function A B f w) (app f a)))))
    (path-concat B
      (app f (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a)))))
      (app f (app (left-inverse-function A B f w) (app f a)))
      (app f a)
      (ap A B f
        (app (left-inverse-function A B f w)
          (app f (app (left-inverse-function A B f w) (app f a))))
        (app (left-inverse-function A B f w) (app f a))
        (app (left-inverse-homotopy A B f w)
          (app (left-inverse-function A B f w) (app f a))))
      (quasi-rho-point A B f w (app f a))))
  (path-concat B
    (app f (app (left-inverse-function A B f w) (app f a)))
    (app f (app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a)))))
    (app f a)
    (path-inverse B
      (app f (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a)))))
      (app f (app (left-inverse-function A B f w) (app f a)))
      (quasi-rho-point A B f w
        (app f (app (left-inverse-function A B f w) (app f a)))))
    (path-concat B
      (app f (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a)))))
      (app f (app (left-inverse-function A B f w) (app f a)))
      (app f a)
      (quasi-rho-point A B f w
```

```
      (app f (app (left-inverse-function A B f w) (app f a))))
    (ap A B f (app (left-inverse-function A B f w) (app f a)) a
      (app (left-inverse-homotopy A B f w) a))))
  (ap A B f (app (left-inverse-function A B f w) (app f a)) a
    (app (left-inverse-homotopy A B f w) a))
  (path-whisker-left B
    (app f (app (left-inverse-function A B f w) (app f a)))
    (app f (app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a)))))
    (app f a)
    (path-inverse B
      (app f (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a)))))
      (app f (app (left-inverse-function A B f w) (app f a)))
      (quasi-rho-point A B f w
        (app f (app (left-inverse-function A B f w) (app f a)))))
    (path-concat B
      (app f (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a)))))
      (app f (app (left-inverse-function A B f w) (app f a)))
      (app f a)
      (ap A B f
        (app (left-inverse-function A B f w)
          (app f (app (left-inverse-function A B f w) (app f a))))
        (app (left-inverse-function A B f w) (app f a))
        (app (left-inverse-homotopy A B f w)
          (app (left-inverse-function A B f w) (app f a))))
      (quasi-rho-point A B f w (app f a)))
    (path-concat B
      (app f (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a)))))
      (app f (app (left-inverse-function A B f w) (app f a)))
      (app f a)
      (quasi-rho-point A B f w
        (app f (app (left-inverse-function A B f w) (app f a))))
      (ap A B f (app (left-inverse-function A B f w) (app f a)) a
        (app (left-inverse-homotopy A B f w) a)))
    (coherent-inner A B f w a))
  (path-inverse-concat-left B
    (app f (app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a)))))
    (app f (app (left-inverse-function A B f w) (app f a)))
    (app f a)
    (quasi-rho-point A B f w
      (app f (app (left-inverse-function A B f w) (app f a))))
    (ap A B f (app (left-inverse-function A B f w) (app f a)) a
      (app (left-inverse-homotopy A B f w) a))))
```

lemma *coherent-tau-type*:

```
  assumes A: A ::_h universe i
    and B: B ::_h universe i
    and f: f ::_h arrow A B
    and w: w ::_h biinvertible A B f
    and a: a ::_h A
  shows coherent-tau A B f w a ::_h
    identity-type
      (identity-type B (app f (app (left-inverse-function A B f w) (app f a)))
        (app f a))
      (ap A B f (app (left-inverse-function A B f w) (app f a)) a
        (app (left-inverse-homotopy A B f w) a))
      (coherent-eps-point A B f w (app f a))
  unfolding coherent-tau-def coherent-eps-point-def
proof −
  have g: left-inverse-function A B f w ::_h arrow B A
    by (rule left-inverse-function-type[OF w])
  have eta: left-inverse-homotopy A B f w ::_h
      homotopy A (λ-. A)
        (compose A B A (left-inverse-function A B f w) f)
        (identity-function A)
    by (rule left-inverse-homotopy-type[OF w])
  have fa: app f a ::_h B by (rule arrow-elim[OF f a])
  have gfa: app (left-inverse-function A B f w) (app f a) ::_h A
    by (rule arrow-elim[OF g fa])
  have fgfa: app f (app (left-inverse-function A B f w) (app f a)) ::_h B
    by (rule arrow-elim[OF f gfa])
  have gfgfa: app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a))) ::_h A
    by (rule arrow-elim[OF g fgfa])
  have fgfgfa: app f (app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a)))) ::_h B
    by (rule arrow-elim[OF f gfgfa])
  have etapi: left-inverse-homotopy A B f w ::_h
      pi-type A (λx. identity-type A
        (app (compose A B A (left-inverse-function A B f w) f) x)
        (app (identity-function A) x))
    using eta unfolding homotopy-def .
  have eta-a: app (left-inverse-homotopy A B f w) a ::_h
      identity-type A (app (left-inverse-function A B f w) (app f a)) a
    using pi-elim[OF etapi a] by simp
  have eta-gfa: app (left-inverse-homotopy A B f w)
      (app (left-inverse-function A B f w) (app f a)) ::_h
      identity-type A
        (app (left-inverse-function A B f w)
          (app f (app (left-inverse-function A B f w) (app f a))))
        (app (left-inverse-function A B f w) (app f a))
    using pi-elim[OF etapi gfa] by simp
  have s: ap A B f (app (left-inverse-function A B f w) (app f a)) a
      (app (left-inverse-homotopy A B f w) a) ::_h
```

identity-type B (app f (app (left-inverse-function A B f w) (app f a)))
(app f a)
by (*rule ap-type*[*OF A B f gfa a eta-a*])
have *apfeta*: *ap A B f*
(app (left-inverse-function A B f w)
(app f (app (left-inverse-function A B f w) (app f a))))
(app (left-inverse-function A B f w) (app f a))
(app (left-inverse-homotopy A B f w)
(app (left-inverse-function A B f w) (app f a))) $::_h$
identity-type B
(app f (app (left-inverse-function A B f w)
(app f (app (left-inverse-function A B f w) (app f a)))))
(app f (app (left-inverse-function A B f w) (app f a)))
by (*rule ap-type*[*OF A B f gfgfa gfa eta-gfa*])
have *rho-fa*: *quasi-rho-point A B f w (app f a)* $::_h$
identity-type B (app f (app (left-inverse-function A B f w) (app f a)))
(app f a)
by (*rule quasi-rho-point-type*[*OF A B f w fa*])
have *rho-fgfa*: *quasi-rho-point A B f w*
(app f (app (left-inverse-function A B f w) (app f a))) $::_h$
identity-type B
(app f (app (left-inverse-function A B f w)
(app f (app (left-inverse-function A B f w) (app f a)))))
(app f (app (left-inverse-function A B f w) (app f a)))
by (*rule quasi-rho-point-type*[*OF A B f w fgfa*])
have *rho-fgfa-inv*: *path-inverse B*
(app f (app (left-inverse-function A B f w)
(app f (app (left-inverse-function A B f w) (app f a)))))
(app f (app (left-inverse-function A B f w) (app f a)))
(quasi-rho-point A B f w
(app f (app (left-inverse-function A B f w) (app f a)))) $::_h$
identity-type B
(app f (app (left-inverse-function A B f w) (app f a)))
(app f (app (left-inverse-function A B f w)
(app f (app (left-inverse-function A B f w) (app f a)))))
by (*rule path-inverse-type*[*OF B fgfgfa fgfa rho-fgfa*])
have *SPACE*: *identity-type B*
(app f (app (left-inverse-function A B f w) (app f a))) (app f a)
$::_h$ *universe i*
by (*rule identity-form*[*OF B fgfa fa*])
have *innerE*: *path-concat B*
(app f (app (left-inverse-function A B f w)
(app f (app (left-inverse-function A B f w) (app f a)))))
(app f (app (left-inverse-function A B f w) (app f a)))
(app f a)
(ap A B f
(app (left-inverse-function A B f w)
(app f (app (left-inverse-function A B f w) (app f a))))
(app (left-inverse-function A B f w) (app f a))

(*app* (*left-inverse-homotopy A B f w*)
(*app* (*left-inverse-function A B f w*) (*app f a*))))
(*quasi-rho-point A B f w* (*app f a*)) ::$_h$
identity-type B
(*app f* (*app* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))))
(*app f a*)
by (*rule path-concat-type*[*OF B fgfgfa fgfa fa apfeta rho-fa*])
have *rhofgfa-s*: *path-concat B*
(*app f* (*app* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))))
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))
(*app f a*)
(*quasi-rho-point A B f w*
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*))))
(*ap A B f* (*app* (*left-inverse-function A B f w*) (*app f a*)) *a*
(*app* (*left-inverse-homotopy A B f w*) *a*)) ::$_h$
identity-type B
(*app f* (*app* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))))
(*app f a*)
by (*rule path-concat-type*[*OF B fgfgfa fgfa fa rho-fgfa s*])
have *eps*: *path-concat B*
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))
(*app f* (*app* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))))
(*app f a*)
(*path-inverse B*
(*app f* (*app* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))))
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))
(*quasi-rho-point A B f w*
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))))
(*path-concat B*
(*app f* (*app* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))))
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))
(*app f a*)
(*ap A B f*
(*app* (*left-inverse-function A B f w*)
(*app f* (*app* (*left-inverse-function A B f w*) (*app f a*))))
(*app* (*left-inverse-function A B f w*) (*app f a*))
(*app* (*left-inverse-homotopy A B f w*)
(*app* (*left-inverse-function A B f w*) (*app f a*))))
(*quasi-rho-point A B f w* (*app f a*)) ::$_h$
identity-type B (*app f* (*app* (*left-inverse-function A B f w*) (*app f a*)))
(*app f a*)
by (*rule path-concat-type*[*OF B fgfa fgfgfa fa rho-fgfa-inv innerE*])
have *MID*: *path-concat B*

$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))))$
$(app\ f\ a)$
$(path\text{-}inverse\ B$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))))$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))$
$(quasi\text{-}rho\text{-}point\ A\ B\ f\ w$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))))$
$(path\text{-}concat\ B$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))))$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))$
$(app\ f\ a)$
$(quasi\text{-}rho\text{-}point\ A\ B\ f\ w$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a))))$
$(ap\ A\ B\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a))\ a$
$(app\ (left\text{-}inverse\text{-}homotopy\ A\ B\ f\ w)\ a)))\ ::_h$
$identity\text{-}type\ B\ (app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))$
$(app\ f\ a)$
by $(rule\ path\text{-}concat\text{-}type[OF\ B\ fgfa\ fgfgfa\ fa\ rho\text{-}fgfa\text{-}inv\ rhofgfa\text{-}s])$
have WL: $path\text{-}whisker\text{-}left\ B$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))))$
$(app\ f\ a)$
$(path\text{-}inverse\ B$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))))$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))$
$(quasi\text{-}rho\text{-}point\ A\ B\ f\ w$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))))$
$(path\text{-}concat\ B$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))))$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))$
$(app\ f\ a)$
$(ap\ A\ B\ f$
$(app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a))))$
$(app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a))$
$(app\ (left\text{-}inverse\text{-}homotopy\ A\ B\ f\ w)$
$(app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a))))$
$(quasi\text{-}rho\text{-}point\ A\ B\ f\ w\ (app\ f\ a)))$
$(path\text{-}concat\ B$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))))$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))$

```
  (app f a)
  (quasi-rho-point A B f w
    (app f (app (left-inverse-function A B f w) (app f a))))
  (ap A B f (app (left-inverse-function A B f w) (app f a)) a
    (app (left-inverse-homotopy A B f w) a)))
(coherent-inner A B f w a) ::_h
identity-type
  (identity-type B (app f (app (left-inverse-function A B f w) (app f a)))
    (app f a))
  (path-concat B
    (app f (app (left-inverse-function A B f w) (app f a)))
    (app f (app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a)))))
    (app f a)
    (path-inverse B
      (app f (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a)))))
      (app f (app (left-inverse-function A B f w) (app f a)))
      (quasi-rho-point A B f w
        (app f (app (left-inverse-function A B f w) (app f a)))))
    (path-concat B
      (app f (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a)))))
      (app f (app (left-inverse-function A B f w) (app f a)))
      (app f a)
      (ap A B f
        (app (left-inverse-function A B f w)
          (app f (app (left-inverse-function A B f w) (app f a))))
        (app (left-inverse-function A B f w) (app f a))
        (app (left-inverse-homotopy A B f w)
          (app (left-inverse-function A B f w) (app f a))))
      (quasi-rho-point A B f w (app f a)))))
  (path-concat B
    (app f (app (left-inverse-function A B f w) (app f a)))
    (app f (app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a)))))
    (app f a)
    (path-inverse B
      (app f (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a)))))
      (app f (app (left-inverse-function A B f w) (app f a)))
      (quasi-rho-point A B f w
        (app f (app (left-inverse-function A B f w) (app f a)))))
    (path-concat B
      (app f (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a)))))
      (app f (app (left-inverse-function A B f w) (app f a)))
      (app f a)
      (quasi-rho-point A B f w
```

```
        (app f (app (left-inverse-function A B f w) (app f a))))
      (ap A B f (app (left-inverse-function A B f w) (app f a)) a
        (app (left-inverse-homotopy A B f w) a))))
  by (rule path-whisker-left-type[OF B fgfa fgfgfa fa rho-fgfa-inv
       innerE rhofgfa-s coherent-inner-type[OF A B f w a]])
have CANCEL: path-inverse-concat-left B
    (app f (app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a)))))
    (app f (app (left-inverse-function A B f w) (app f a)))
    (app f a)
    (quasi-rho-point A B f w
      (app f (app (left-inverse-function A B f w) (app f a))))
    (ap A B f (app (left-inverse-function A B f w) (app f a)) a
      (app (left-inverse-homotopy A B f w) a)) ::_h
    identity-type
      (identity-type B (app f (app (left-inverse-function A B f w) (app f a)))
        (app f a))
      (path-concat B
        (app f (app (left-inverse-function A B f w) (app f a)))
        (app f (app (left-inverse-function A B f w)
          (app f (app (left-inverse-function A B f w) (app f a)))))
        (app f a)
        (path-inverse B
          (app f (app (left-inverse-function A B f w)
            (app f (app (left-inverse-function A B f w) (app f a)))))
          (app f (app (left-inverse-function A B f w) (app f a)))
          (quasi-rho-point A B f w
            (app f (app (left-inverse-function A B f w) (app f a)))))
        (path-concat B
          (app f (app (left-inverse-function A B f w)
            (app f (app (left-inverse-function A B f w) (app f a)))))
          (app f (app (left-inverse-function A B f w) (app f a)))
          (app f a)
          (quasi-rho-point A B f w
            (app f (app (left-inverse-function A B f w) (app f a))))
          (ap A B f (app (left-inverse-function A B f w) (app f a)) a
            (app (left-inverse-homotopy A B f w) a))))
      (ap A B f (app (left-inverse-function A B f w) (app f a)) a
        (app (left-inverse-homotopy A B f w) a))
  by (rule path-inverse-concat-left-type[OF B fgfgfa fgfa fa rho-fgfa s])
have EQ: path-concat
    (identity-type B (app f (app (left-inverse-function A B f w) (app f a)))
      (app f a))
    (path-concat B
      (app f (app (left-inverse-function A B f w) (app f a)))
      (app f (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a)))))
      (app f a)
      (path-inverse B
```

```
    (app f (app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a)))))
    (app f (app (left-inverse-function A B f w) (app f a)))
    (quasi-rho-point A B f w
      (app f (app (left-inverse-function A B f w) (app f a)))))
  (path-concat B
    (app f (app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a)))))
    (app f (app (left-inverse-function A B f w) (app f a)))
    (app f a)
    (ap A B f
      (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a))))
      (app (left-inverse-function A B f w) (app f a))
      (app (left-inverse-homotopy A B f w)
        (app (left-inverse-function A B f w) (app f a))))
    (quasi-rho-point A B f w (app f a))))
(path-concat B
  (app f (app (left-inverse-function A B f w) (app f a)))
  (app f (app (left-inverse-function A B f w)
    (app f (app (left-inverse-function A B f w) (app f a)))))
  (app f a)
  (path-inverse B
    (app f (app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a)))))
    (app f (app (left-inverse-function A B f w) (app f a)))
    (quasi-rho-point A B f w
      (app f (app (left-inverse-function A B f w) (app f a)))))
  (path-concat B
    (app f (app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a)))))
    (app f (app (left-inverse-function A B f w) (app f a)))
    (app f a)
    (quasi-rho-point A B f w
      (app f (app (left-inverse-function A B f w) (app f a))))
    (ap A B f (app (left-inverse-function A B f w) (app f a)) a
      (app (left-inverse-homotopy A B f w) a))))
(ap A B f (app (left-inverse-function A B f w) (app f a)) a
  (app (left-inverse-homotopy A B f w) a))
(path-whisker-left B
  (app f (app (left-inverse-function A B f w) (app f a)))
  (app f (app (left-inverse-function A B f w)
    (app f (app (left-inverse-function A B f w) (app f a)))))
  (app f a)
  (path-inverse B
    (app f (app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a)))))
    (app f (app (left-inverse-function A B f w) (app f a)))
    (quasi-rho-point A B f w
```

```
      (app f (app (left-inverse-function A B f w) (app f a)))))
  (path-concat B
    (app f (app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a)))))
    (app f (app (left-inverse-function A B f w) (app f a)))
    (app f a)
    (ap A B f
      (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a))))
      (app (left-inverse-function A B f w) (app f a))
      (app (left-inverse-homotopy A B f w)
        (app (left-inverse-function A B f w) (app f a))))
    (quasi-rho-point A B f w (app f a)))
  (path-concat B
    (app f (app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a)))))
    (app f (app (left-inverse-function A B f w) (app f a)))
    (app f a)
    (quasi-rho-point A B f w
      (app f (app (left-inverse-function A B f w) (app f a))))
    (ap A B f (app (left-inverse-function A B f w) (app f a)) a
      (app (left-inverse-homotopy A B f w) a)))
  (coherent-inner A B f w a))
(path-inverse-concat-left B
  (app f (app (left-inverse-function A B f w)
    (app f (app (left-inverse-function A B f w) (app f a)))))
  (app f (app (left-inverse-function A B f w) (app f a)))
  (app f a)
  (quasi-rho-point A B f w
    (app f (app (left-inverse-function A B f w) (app f a))))
  (ap A B f (app (left-inverse-function A B f w) (app f a)) a
    (app (left-inverse-homotopy A B f w) a))) ::_h
identity-type
  (identity-type B (app f (app (left-inverse-function A B f w) (app f a)))
    (app f a))
  (path-concat B
    (app f (app (left-inverse-function A B f w) (app f a)))
    (app f (app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a)))))
    (app f a)
    (path-inverse B
      (app f (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a)))))
      (app f (app (left-inverse-function A B f w) (app f a)))
      (quasi-rho-point A B f w
        (app f (app (left-inverse-function A B f w) (app f a)))))
    (path-concat B
      (app f (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a)))))
```

```
      (app f (app (left-inverse-function A B f w) (app f a)))
      (app f a)
      (ap A B f
        (app (left-inverse-function A B f w)
          (app f (app (left-inverse-function A B f w) (app f a))))
        (app (left-inverse-function A B f w) (app f a))
        (app (left-inverse-homotopy A B f w)
          (app (left-inverse-function A B f w) (app f a))))
      (quasi-rho-point A B f w (app f a))))
    (ap A B f (app (left-inverse-function A B f w) (app f a)) a
      (app (left-inverse-homotopy A B f w) a))
  by (rule path-concat-type[OF SPACE eps MID s WL CANCEL])
show path-inverse
    (identity-type B (app f (app (left-inverse-function A B f w) (app f a)))
      (app f a))
    (path-concat B
      (app f (app (left-inverse-function A B f w) (app f a)))
      (app f (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a)))))
      (app f a)
      (path-inverse B
        (app f (app (left-inverse-function A B f w)
          (app f (app (left-inverse-function A B f w) (app f a)))))
        (app f (app (left-inverse-function A B f w) (app f a)))
        (quasi-rho-point A B f w
          (app f (app (left-inverse-function A B f w) (app f a)))))
      (path-concat B
        (app f (app (left-inverse-function A B f w)
          (app f (app (left-inverse-function A B f w) (app f a)))))
        (app f (app (left-inverse-function A B f w) (app f a)))
        (app f a)
        (ap A B f
          (app (left-inverse-function A B f w)
            (app f (app (left-inverse-function A B f w) (app f a))))
          (app (left-inverse-function A B f w) (app f a))
          (app (left-inverse-homotopy A B f w)
            (app (left-inverse-function A B f w) (app f a))))
        (quasi-rho-point A B f w (app f a))))
    (ap A B f (app (left-inverse-function A B f w) (app f a)) a
      (app (left-inverse-homotopy A B f w) a))
    (path-concat
      (identity-type B (app f (app (left-inverse-function A B f w) (app f a)))
        (app f a))
      (path-concat B
        (app f (app (left-inverse-function A B f w) (app f a)))
        (app f (app (left-inverse-function A B f w)
          (app f (app (left-inverse-function A B f w) (app f a)))))
        (app f a)
        (path-inverse B
```

```
    (app f (app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a)))))
    (app f (app (left-inverse-function A B f w) (app f a)))
    (quasi-rho-point A B f w
      (app f (app (left-inverse-function A B f w) (app f a)))))
  (path-concat B
    (app f (app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a)))))
    (app f (app (left-inverse-function A B f w) (app f a)))
    (app f a)
    (ap A B f
      (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a))))
      (app (left-inverse-function A B f w) (app f a))
      (app (left-inverse-homotopy A B f w)
        (app (left-inverse-function A B f w) (app f a))))
    (quasi-rho-point A B f w (app f a))))
(path-concat B
  (app f (app (left-inverse-function A B f w) (app f a)))
  (app f (app (left-inverse-function A B f w)
    (app f (app (left-inverse-function A B f w) (app f a)))))
  (app f a)
  (path-inverse B
    (app f (app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a)))))
    (app f (app (left-inverse-function A B f w) (app f a)))
    (quasi-rho-point A B f w
      (app f (app (left-inverse-function A B f w) (app f a)))))
  (path-concat B
    (app f (app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a)))))
    (app f (app (left-inverse-function A B f w) (app f a)))
    (app f a)
    (quasi-rho-point A B f w
      (app f (app (left-inverse-function A B f w) (app f a))))
    (ap A B f (app (left-inverse-function A B f w) (app f a)) a
      (app (left-inverse-homotopy A B f w) a))))
(ap A B f (app (left-inverse-function A B f w) (app f a)) a
  (app (left-inverse-homotopy A B f w) a))
(path-whisker-left B
  (app f (app (left-inverse-function A B f w) (app f a)))
  (app f (app (left-inverse-function A B f w)
    (app f (app (left-inverse-function A B f w) (app f a)))))
  (app f a)
  (path-inverse B
    (app f (app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a)))))
    (app f (app (left-inverse-function A B f w) (app f a)))
    (quasi-rho-point A B f w
```

```
      (app f (app (left-inverse-function A B f w) (app f a)))))
    (path-concat B
      (app f (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a)))))
      (app f (app (left-inverse-function A B f w) (app f a)))
      (app f a)
      (ap A B f
        (app (left-inverse-function A B f w)
          (app f (app (left-inverse-function A B f w) (app f a))))
        (app (left-inverse-function A B f w) (app f a))
        (app (left-inverse-homotopy A B f w)
          (app (left-inverse-function A B f w) (app f a))))
      (quasi-rho-point A B f w (app f a)))
    (path-concat B
      (app f (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a)))))
      (app f (app (left-inverse-function A B f w) (app f a)))
      (app f a)
      (quasi-rho-point A B f w
        (app f (app (left-inverse-function A B f w) (app f a))))
      (ap A B f (app (left-inverse-function A B f w) (app f a)) a
        (app (left-inverse-homotopy A B f w) a)))
    (coherent-inner A B f w a))
  (path-inverse-concat-left B
    (app f (app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a)))))
    (app f (app (left-inverse-function A B f w) (app f a)))
    (app f a)
    (quasi-rho-point A B f w
      (app f (app (left-inverse-function A B f w) (app f a))))
    (ap A B f (app (left-inverse-function A B f w) (app f a)) a
      (app (left-inverse-homotopy A B f w) a)))) ::_h
identity-type
  (identity-type B (app f (app (left-inverse-function A B f w) (app f a)))
    (app f a))
  (ap A B f (app (left-inverse-function A B f w) (app f a)) a
    (app (left-inverse-homotopy A B f w) a))
  (path-concat B
    (app f (app (left-inverse-function A B f w) (app f a)))
    (app f (app (left-inverse-function A B f w)
      (app f (app (left-inverse-function A B f w) (app f a)))))
    (app f a)
    (path-inverse B
      (app f (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a)))))
      (app f (app (left-inverse-function A B f w) (app f a)))
      (quasi-rho-point A B f w
        (app f (app (left-inverse-function A B f w) (app f a)))))
    (path-concat B
```

```
      (app f (app (left-inverse-function A B f w)
        (app f (app (left-inverse-function A B f w) (app f a)))))
      (app f (app (left-inverse-function A B f w) (app f a)))
      (app f a)
      (ap A B f
        (app (left-inverse-function A B f w)
          (app f (app (left-inverse-function A B f w) (app f a))))
        (app (left-inverse-function A B f w) (app f a))
        (app (left-inverse-homotopy A B f w)
          (app (left-inverse-function A B f w) (app f a))))
      (quasi-rho-point A B f w (app f a))))
  by (rule path-inverse-type[OF SPACE eps s EQ])
qed
```

The coherent section homotopy bundled as a term, so that its naturality square is available.

```
definition coherent-eps ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr
where
  coherent-eps A B f w ≡ lambda B (λb. coherent-eps-point A B f w b)

lemma coherent-eps-type:
  assumes A: A ::h universe i
    and B: B ::h universe i
    and f: f ::h arrow A B
    and w: w ::h biinvertible A B f
  shows coherent-eps A B f w ::h
    homotopy B (λ-. B)
      (compose B A B f (left-inverse-function A B f w))
      (identity-function B)
  unfolding coherent-eps-def homotopy-def
proof (rule pi-intro[OF B])
  fix b
  assume b: b ::h B
  have point: coherent-eps-point A B f w b ::h
      identity-type B (app f (app (left-inverse-function A B f w) b)) b
    by (rule coherent-eps-point-type[OF A B f w b])
  show coherent-eps-point A B f w b ::h
      identity-type B
        (app (compose B A B f (left-inverse-function A B f w)) b)
        (app (identity-function B) b)
    using point by simp
qed

lemma coherent-eps-beta [simp]:
  app (coherent-eps A B f w) b ≡ coherent-eps-point A B f w b
  unfolding coherent-eps-def by simp
```

30 Contractibility of the fibers of a bi-invertible map

Given a fibre point $(x,\ p)$ with $p : f\ x = b$, the first component of the path to the centre $(g\ b,\ \varepsilon\ b)$ is $ap\ g\ p^{-1} \cdot \eta\ x : g\ b = x$.

definition *fiber-gamma* ::
  expr ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr*
where
  fiber-gamma A B f w b x p ≡
    path-concat A
      (*app* (*left-inverse-function A B f w*) *b*)
      (*app* (*left-inverse-function A B f w*) (*app f x*))
      x
      (*ap B A* (*left-inverse-function A B f w*) *b* (*app f x*)
        (*path-inverse B* (*app f x*) *b p*))
      (*app* (*left-inverse-homotopy A B f w*) *x*)

lemma *fiber-gamma-type*:
  assumes *A*: $A ::_h$ *universe i*
    and *B*: $B ::_h$ *universe i*
    and *f*: $f ::_h$ *arrow A B*
    and *w*: $w ::_h$ *biinvertible A B f*
    and *b*: $b ::_h B$
    and *x*: $x ::_h A$
    and *p*: $p ::_h$ *identity-type B* (*app f x*) *b*
  shows *fiber-gamma A B f w b x p* $::_h$
    identity-type A (*app* (*left-inverse-function A B f w*) *b*) *x*
proof −
  have *g*: *left-inverse-function A B f w* $::_h$ *arrow B A*
    by (*rule left-inverse-function-type*[*OF w*])
  have *eta*: *left-inverse-homotopy A B f w* $::_h$
      homotopy A (λ-. *A*)
        (*compose A B A* (*left-inverse-function A B f w*) *f*)
        (*identity-function A*)
    by (*rule left-inverse-homotopy-type*[*OF w*])
  have *fx*: *app f x* $::_h$ *B* **by** (*rule arrow-elim*[*OF f x*])
  have *gb*: *app* (*left-inverse-function A B f w*) *b* $::_h$ *A*
    by (*rule arrow-elim*[*OF g b*])
  have *gfx*: *app* (*left-inverse-function A B f w*) (*app f x*) $::_h$ *A*
    by (*rule arrow-elim*[*OF g fx*])
  have *pinv*: *path-inverse B* (*app f x*) *b p* $::_h$
      identity-type B b (*app f x*)
    by (*rule path-inverse-type*[*OF B fx b p*])
  have *apg*: *ap B A* (*left-inverse-function A B f w*) *b* (*app f x*)
      (*path-inverse B* (*app f x*) *b p*) $::_h$
      identity-type A (*app* (*left-inverse-function A B f w*) *b*)
        (*app* (*left-inverse-function A B f w*) (*app f x*))
    by (*rule ap-type*[*OF B A g b fx pinv*])

```
  have etapi: left-inverse-homotopy A B f w ::_h
      pi-type A (λy. identity-type A
        (app (compose A B A (left-inverse-function A B f w) f) y)
        (app (identity-function A) y))
    using eta unfolding homotopy-def .
  have etax: app (left-inverse-homotopy A B f w) x ::_h
      identity-type A (app (left-inverse-function A B f w) (app f x)) x
    using pi-elim[OF etapi x] by simp
  show fiber-gamma A B f w b x p ::_h
      identity-type A (app (left-inverse-function A B f w) b) x
    unfolding fiber-gamma-def
    by (rule path-concat-type[OF A gb gfx x apg etax])
qed
```

The key computation $ap\ f\ \gamma = \varepsilon\ b \cdot p^{-1}$, obtained from functoriality of *ap*, the triangle identity *coherent-tau*, and naturality of ε.

```
definition fiber-apf-gamma ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr
where
  fiber-apf-gamma A B f w b x p ≡
    path-concat
      (identity-type B (app f (app (left-inverse-function A B f w) b))
        (app f x))
      (ap A B f (app (left-inverse-function A B f w) b) x
        (fiber-gamma A B f w b x p))
      (path-concat B (app f (app (left-inverse-function A B f w) b))
        (app f (app (left-inverse-function A B f w) (app f x)))
        (app f x)
        (ap B B (compose B A B f (left-inverse-function A B f w)) b
          (app f x) (path-inverse B (app f x) b p))
        (coherent-eps-point A B f w (app f x)))
      (path-concat B (app f (app (left-inverse-function A B f w) b)) b
        (app f x)
        (coherent-eps-point A B f w b)
        (path-inverse B (app f x) b p))
      (path-concat
        (identity-type B (app f (app (left-inverse-function A B f w) b))
          (app f x))
        (ap A B f (app (left-inverse-function A B f w) b) x
          (fiber-gamma A B f w b x p))
        (path-concat B (app f (app (left-inverse-function A B f w) b))
          (app f (app (left-inverse-function A B f w) (app f x)))
          (app f x)
          (ap B B (compose B A B f (left-inverse-function A B f w)) b
            (app f x) (path-inverse B (app f x) b p))
          (ap A B f (app (left-inverse-function A B f w) (app f x)) x
            (app (left-inverse-homotopy A B f w) x)))
        (path-concat B (app f (app (left-inverse-function A B f w) b))
          (app f (app (left-inverse-function A B f w) (app f x)))
```

```
  (app f x)
  (ap B B (compose B A B f (left-inverse-function A B f w)) b
    (app f x) (path-inverse B (app f x) b p))
  (coherent-eps-point A B f w (app f x)))
(path-concat
  (identity-type B (app f (app (left-inverse-function A B f w) b))
    (app f x))
  (ap A B f (app (left-inverse-function A B f w) b) x
    (fiber-gamma A B f w b x p))
  (path-concat B (app f (app (left-inverse-function A B f w) b))
    (app f (app (left-inverse-function A B f w) (app f x)))
    (app f x)
    (ap A B f (app (left-inverse-function A B f w) b)
      (app (left-inverse-function A B f w) (app f x))
      (ap B A (left-inverse-function A B f w) b (app f x)
        (path-inverse B (app f x) b p)))
    (ap A B f (app (left-inverse-function A B f w) (app f x)) x
      (app (left-inverse-homotopy A B f w) x)))
  (path-concat B (app f (app (left-inverse-function A B f w) b))
    (app f (app (left-inverse-function A B f w) (app f x)))
    (app f x)
    (ap B B (compose B A B f (left-inverse-function A B f w)) b
      (app f x) (path-inverse B (app f x) b p))
    (ap A B f (app (left-inverse-function A B f w) (app f x)) x
      (app (left-inverse-homotopy A B f w) x)))
  (ap-concat A B f (app (left-inverse-function A B f w) b)
    (app (left-inverse-function A B f w) (app f x)) x
    (ap B A (left-inverse-function A B f w) b (app f x)
      (path-inverse B (app f x) b p))
    (app (left-inverse-homotopy A B f w) x))
  (path-whisker-right B
    (app f (app (left-inverse-function A B f w) b))
    (app f (app (left-inverse-function A B f w) (app f x)))
    (app f x)
    (ap A B f (app (left-inverse-function A B f w) b)
      (app (left-inverse-function A B f w) (app f x))
      (ap B A (left-inverse-function A B f w) b (app f x)
        (path-inverse B (app f x) b p)))
    (ap B B (compose B A B f (left-inverse-function A B f w)) b
      (app f x) (path-inverse B (app f x) b p))
    (ap A B f (app (left-inverse-function A B f w) (app f x)) x
      (app (left-inverse-homotopy A B f w) x))
    (path-inverse
      (identity-type B (app f (app (left-inverse-function A B f w) b))
        (app f (app (left-inverse-function A B f w) (app f x))))
      (ap B B (compose B A B f (left-inverse-function A B f w)) b
        (app f x) (path-inverse B (app f x) b p))
      (ap A B f (app (left-inverse-function A B f w) b)
        (app (left-inverse-function A B f w) (app f x))
```

```
            (ap B A (left-inverse-function A B f w) b (app f x)
              (path-inverse B (app f x) b p)))
          (ap-compose B A B f (left-inverse-function A B f w) b
            (app f x) (path-inverse B (app f x) b p)))))
      (path-whisker-left B
        (app f (app (left-inverse-function A B f w) b))
        (app f (app (left-inverse-function A B f w) (app f x)))
        (app f x)
        (ap B B (compose B A B f (left-inverse-function A B f w)) b
          (app f x) (path-inverse B (app f x) b p))
        (ap A B f (app (left-inverse-function A B f w) (app f x)) x
          (app (left-inverse-homotopy A B f w) x))
        (coherent-eps-point A B f w (app f x))
        (coherent-tau A B f w x)))
    (path-concat
      (identity-type B (app f (app (left-inverse-function A B f w) b))
        (app f x))
      (path-concat B (app f (app (left-inverse-function A B f w) b))
        (app f (app (left-inverse-function A B f w) (app f x)))
        (app f x)
        (ap B B (compose B A B f (left-inverse-function A B f w)) b
          (app f x) (path-inverse B (app f x) b p))
        (coherent-eps-point A B f w (app f x)))
      (path-concat B (app f (app (left-inverse-function A B f w) b)) b
        (app f x)
        (coherent-eps-point A B f w b)
        (ap B B (identity-function B) b (app f x)
          (path-inverse B (app f x) b p)))
      (path-concat B (app f (app (left-inverse-function A B f w) b)) b
        (app f x)
        (coherent-eps-point A B f w b)
        (path-inverse B (app f x) b p))
      (homotopy-naturality B B
        (compose B A B f (left-inverse-function A B f w))
        (identity-function B)
        (coherent-eps A B f w)
        b (app f x) (path-inverse B (app f x) b p))
      (path-whisker-left B
        (app f (app (left-inverse-function A B f w) b)) b (app f x)
        (coherent-eps-point A B f w b)
        (ap B B (identity-function B) b (app f x)
          (path-inverse B (app f x) b p))
        (path-inverse B (app f x) b p)
        (ap-identity B b (app f x) (path-inverse B (app f x) b p))))))

lemma fiber-apf-gamma-type:
  assumes A: A ::_h universe i
    and B: B ::_h universe i
    and f: f ::_h arrow A B
```

```
    and w: w ::_h biinvertible A B f
    and b: b ::_h B
    and x: x ::_h A
    and p: p ::_h identity-type B (app f x) b
  shows fiber-apf-gamma A B f w b x p ::_h
    identity-type
      (identity-type B (app f (app (left-inverse-function A B f w) b))
        (app f x))
      (ap A B f (app (left-inverse-function A B f w) b) x
        (fiber-gamma A B f w b x p))
      (path-concat B (app f (app (left-inverse-function A B f w) b)) b
        (app f x)
        (coherent-eps-point A B f w b)
        (path-inverse B (app f x) b p))
proof −
  have g: left-inverse-function A B f w ::_h arrow B A
    by (rule left-inverse-function-type[OF w])
  have eta: left-inverse-homotopy A B f w ::_h
      homotopy A (λ-. A)
        (compose A B A (left-inverse-function A B f w) f)
        (identity-function A)
    by (rule left-inverse-homotopy-type[OF w])
  have fg: compose B A B f (left-inverse-function A B f w) ::_h arrow B B
    by (rule compose-type[OF B g f])
  have idB: identity-function B ::_h arrow B B
    by (rule identity-function-type[OF B])
  have eps-hom: coherent-eps A B f w ::_h
      homotopy B (λ-. B)
        (compose B A B f (left-inverse-function A B f w))
        (identity-function B)
    by (rule coherent-eps-type[OF A B f w])
  have fx: app f x ::_h B by (rule arrow-elim[OF f x])
  have gb: app (left-inverse-function A B f w) b ::_h A
    by (rule arrow-elim[OF g b])
  have gfx: app (left-inverse-function A B f w) (app f x) ::_h A
    by (rule arrow-elim[OF g fx])
  have fgb: app f (app (left-inverse-function A B f w) b) ::_h B
    by (rule arrow-elim[OF f gb])
  have fgfx: app f (app (left-inverse-function A B f w) (app f x)) ::_h B
    by (rule arrow-elim[OF f gfx])
  have pinv: path-inverse B (app f x) b p ::_h
      identity-type B b (app f x)
    by (rule path-inverse-type[OF B fx b p])
  have apg: ap B A (left-inverse-function A B f w) b (app f x)
      (path-inverse B (app f x) b p) ::_h
      identity-type A (app (left-inverse-function A B f w) b)
        (app (left-inverse-function A B f w) (app f x))
    by (rule ap-type[OF B A g b fx pinv])
  have etapi: left-inverse-homotopy A B f w ::_h
```

pi-type A (λy. *identity-type A*
(*app* (*compose A B A* (*left-inverse-function A B f w*) *f*) *y*)
(*app* (*identity-function A*) *y*))
using *eta* **unfolding** *homotopy-def* **.**
have *etax*: *app* (*left-inverse-homotopy A B f w*) *x* ::$_h$
identity-type A (*app* (*left-inverse-function A B f w*) (*app f x*)) *x*
using *pi-elim*[*OF etapi x*] **by** *simp*
have *gamma*: *fiber-gamma A B f w b x p* ::$_h$
identity-type A (*app* (*left-inverse-function A B f w*) *b*) *x*
by (*rule fiber-gamma-type*[*OF A B f w b x p*])
have *space*: *identity-type B* (*app f* (*app* (*left-inverse-function A B f w*) *b*))
(*app f x*) ::$_h$ *universe i*
by (*rule identity-form*[*OF B fgb fx*])
have *space2*: *identity-type B* (*app f* (*app* (*left-inverse-function A B f w*) *b*))
(*app f* (*app* (*left-inverse-function A B f w*) (*app f x*)))
::$_h$ *universe i*
by (*rule identity-form*[*OF B fgb fgfx*])
have *APFGAM*: *ap A B f* (*app* (*left-inverse-function A B f w*) *b*) *x*
(*fiber-gamma A B f w b x p*) ::$_h$
identity-type B (*app f* (*app* (*left-inverse-function A B f w*) *b*))
(*app f x*)
by (*rule ap-type*[*OF A B f gb x gamma*])
have *APF-APG*: *ap A B f* (*app* (*left-inverse-function A B f w*) *b*)
(*app* (*left-inverse-function A B f w*) (*app f x*))
(*ap B A* (*left-inverse-function A B f w*) *b* (*app f x*)
(*path-inverse B* (*app f x*) *b p*)) ::$_h$
identity-type B (*app f* (*app* (*left-inverse-function A B f w*) *b*))
(*app f* (*app* (*left-inverse-function A B f w*) (*app f x*)))
by (*rule ap-type*[*OF A B f gb gfx apg*])
have *APF-ETAX*: *ap A B f* (*app* (*left-inverse-function A B f w*) (*app f x*)) *x*
(*app* (*left-inverse-homotopy A B f w*) *x*) ::$_h$
identity-type B (*app f* (*app* (*left-inverse-function A B f w*) (*app f x*)))
(*app f x*)
by (*rule ap-type*[*OF A B f gfx x etax*])
have *EPSFX*: *coherent-eps-point A B f w* (*app f x*) ::$_h$
identity-type B (*app f* (*app* (*left-inverse-function A B f w*) (*app f x*)))
(*app f x*)
by (*rule coherent-eps-point-type*[*OF A B f w fx*])
have *EPSB*: *coherent-eps-point A B f w b* ::$_h$
identity-type B (*app f* (*app* (*left-inverse-function A B f w*) *b*)) *b*
by (*rule coherent-eps-point-type*[*OF A B f w b*])
have *APFG-COMPOSE*: *ap B B* (*compose B A B f* (*left-inverse-function A B f w*)) *b*
(*app f x*) (*path-inverse B* (*app f x*) *b p*) ::$_h$
identity-type B (*app f* (*app* (*left-inverse-function A B f w*) *b*))
(*app f* (*app* (*left-inverse-function A B f w*) (*app f x*)))
using *ap-type*[*OF B B fg b fx pinv*] **by** *simp*
have *ap-idQ*: *ap B B* (*identity-function B*) *b* (*app f x*)
(*path-inverse B* (*app f x*) *b p*) ::$_h$

identity-type B b (app f x)
using *ap-type*[*OF B B idB b fx pinv*] **by** *simp*
have *ACON*: *ap-concat A B f (app (left-inverse-function A B f w) b)*
(app (left-inverse-function A B f w) (app f x)) x
(ap B A (left-inverse-function A B f w) b (app f x)
(path-inverse B (app f x) b p))
(app (left-inverse-homotopy A B f w) x) $::_h$
identity-type
(identity-type B (app f (app (left-inverse-function A B f w) b))
(app f x))
(ap A B f (app (left-inverse-function A B f w) b) x
(fiber-gamma A B f w b x p))
(path-concat B (app f (app (left-inverse-function A B f w) b))
(app f (app (left-inverse-function A B f w) (app f x)))
(app f x)
(ap A B f (app (left-inverse-function A B f w) b)
(app (left-inverse-function A B f w) (app f x))
(ap B A (left-inverse-function A B f w) b (app f x)
(path-inverse B (app f x) b p)))
(ap A B f (app (left-inverse-function A B f w) (app f x)) x
(app (left-inverse-homotopy A B f w) x)))
using *ap-concat-type*[*OF A B f gb gfx x apg etax*]
unfolding *fiber-gamma-def* **by** *simp*
have *AC-fg*: *ap-compose B A B f (left-inverse-function A B f w) b*
(app f x) (path-inverse B (app f x) b p) $::_h$
identity-type
(identity-type B (app f (app (left-inverse-function A B f w) b))
(app f (app (left-inverse-function A B f w) (app f x))))
(ap B B (compose B A B f (left-inverse-function A B f w)) b
(app f x) (path-inverse B (app f x) b p))
(ap A B f (app (left-inverse-function A B f w) b)
(app (left-inverse-function A B f w) (app f x))
(ap B A (left-inverse-function A B f w) b (app f x)
(path-inverse B (app f x) b p)))
by (*rule ap-compose-type*[*OF B A B g f b fx pinv*])
have *ACFGinv*: *path-inverse*
(identity-type B (app f (app (left-inverse-function A B f w) b))
(app f (app (left-inverse-function A B f w) (app f x))))
(ap B B (compose B A B f (left-inverse-function A B f w)) b
(app f x) (path-inverse B (app f x) b p))
(ap A B f (app (left-inverse-function A B f w) b)
(app (left-inverse-function A B f w) (app f x))
(ap B A (left-inverse-function A B f w) b (app f x)
(path-inverse B (app f x) b p)))
(ap-compose B A B f (left-inverse-function A B f w) b
(app f x) (path-inverse B (app f x) b p)) $::_h$
identity-type
(identity-type B (app f (app (left-inverse-function A B f w) b))
(app f (app (left-inverse-function A B f w) (app f x))))

$(ap\ A\ B\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ b)$
$(app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ x))$
$(ap\ B\ A\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ b\ (app\ f\ x)$
$(path\text{-}inverse\ B\ (app\ f\ x)\ b\ p)))$
$(ap\ B\ B\ (compose\ B\ A\ B\ f\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w))\ b$
$(app\ f\ x)\ (path\text{-}inverse\ B\ (app\ f\ x)\ b\ p))$
by $(rule\ path\text{-}inverse\text{-}type[OF\ space2\ APFG\text{-}COMPOSE\ APF\text{-}APG\ AC\text{-}fg])$
have WR: $path\text{-}whisker\text{-}right\ B$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ b))$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ x)))$
$(app\ f\ x)$
$(ap\ A\ B\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ b)$
$(app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ x))$
$(ap\ B\ A\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ b\ (app\ f\ x)$
$(path\text{-}inverse\ B\ (app\ f\ x)\ b\ p)))$
$(ap\ B\ B\ (compose\ B\ A\ B\ f\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w))\ b$
$(app\ f\ x)\ (path\text{-}inverse\ B\ (app\ f\ x)\ b\ p))$
$(ap\ A\ B\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ x))\ x$
$(app\ (left\text{-}inverse\text{-}homotopy\ A\ B\ f\ w)\ x))$
$(path\text{-}inverse$
$(identity\text{-}type\ B\ (app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ b))$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ x))))$
$(ap\ B\ B\ (compose\ B\ A\ B\ f\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w))\ b$
$(app\ f\ x)\ (path\text{-}inverse\ B\ (app\ f\ x)\ b\ p))$
$(ap\ A\ B\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ b)$
$(app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ x))$
$(ap\ B\ A\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ b\ (app\ f\ x)$
$(path\text{-}inverse\ B\ (app\ f\ x)\ b\ p)))$
$(ap\text{-}compose\ B\ A\ B\ f\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ b$
$(app\ f\ x)\ (path\text{-}inverse\ B\ (app\ f\ x)\ b\ p)))\ ::_h$
$identity\text{-}type$
$(identity\text{-}type\ B\ (app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ b))$
$(app\ f\ x))$
$(path\text{-}concat\ B\ (app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ b))$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ x)))$
$(app\ f\ x)$
$(ap\ A\ B\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ b)$
$(app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ x))$
$(ap\ B\ A\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ b\ (app\ f\ x)$
$(path\text{-}inverse\ B\ (app\ f\ x)\ b\ p)))$
$(ap\ A\ B\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ x))\ x$
$(app\ (left\text{-}inverse\text{-}homotopy\ A\ B\ f\ w)\ x)))$
$(path\text{-}concat\ B\ (app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ b))$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ x)))$
$(app\ f\ x)$
$(ap\ B\ B\ (compose\ B\ A\ B\ f\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w))\ b$
$(app\ f\ x)\ (path\text{-}inverse\ B\ (app\ f\ x)\ b\ p))$
$(ap\ A\ B\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ x))\ x$
$(app\ (left\text{-}inverse\text{-}homotopy\ A\ B\ f\ w)\ x)))$

by (*rule path-whisker-right-type*[*OF B fgb fgfx fx APF-APG APFG-COMPOSE APF-ETAX ACFGinv*])
have *WL*: *path-whisker-left B*
(*app f* (*app* (*left-inverse-function A B f w*) *b*))
(*app f* (*app* (*left-inverse-function A B f w*) (*app f x*)))
(*app f x*)
(*ap B B* (*compose B A B f* (*left-inverse-function A B f w*)) *b*
(*app f x*) (*path-inverse B* (*app f x*) *b p*))
(*ap A B f* (*app* (*left-inverse-function A B f w*) (*app f x*)) *x*
(*app* (*left-inverse-homotopy A B f w*) *x*))
(*coherent-eps-point A B f w* (*app f x*))
(*coherent-tau A B f w x*) ::$_h$
identity-type
(*identity-type B* (*app f* (*app* (*left-inverse-function A B f w*) *b*))
(*app f x*))
(*path-concat B* (*app f* (*app* (*left-inverse-function A B f w*) *b*))
(*app f* (*app* (*left-inverse-function A B f w*) (*app f x*)))
(*app f x*)
(*ap B B* (*compose B A B f* (*left-inverse-function A B f w*)) *b*
(*app f x*) (*path-inverse B* (*app f x*) *b p*))
(*ap A B f* (*app* (*left-inverse-function A B f w*) (*app f x*)) *x*
(*app* (*left-inverse-homotopy A B f w*) *x*)))
(*path-concat B* (*app f* (*app* (*left-inverse-function A B f w*) *b*))
(*app f* (*app* (*left-inverse-function A B f w*) (*app f x*)))
(*app f x*)
(*ap B B* (*compose B A B f* (*left-inverse-function A B f w*)) *b*
(*app f x*) (*path-inverse B* (*app f x*) *b p*))
(*coherent-eps-point A B f w* (*app f x*)))
by (*rule path-whisker-left-type*[*OF B fgb fgfx fx APFG-COMPOSE APF-ETAX EPSFX coherent-tau-type*[*OF A B f w x*]])
have *APFAPG-APFETAX*: *path-concat B*
(*app f* (*app* (*left-inverse-function A B f w*) *b*))
(*app f* (*app* (*left-inverse-function A B f w*) (*app f x*)))
(*app f x*)
(*ap A B f* (*app* (*left-inverse-function A B f w*) *b*)
(*app* (*left-inverse-function A B f w*) (*app f x*))
(*ap B A* (*left-inverse-function A B f w*) *b* (*app f x*)
(*path-inverse B* (*app f x*) *b p*)))
(*ap A B f* (*app* (*left-inverse-function A B f w*) (*app f x*)) *x*
(*app* (*left-inverse-homotopy A B f w*) *x*)) ::$_h$
identity-type B (*app f* (*app* (*left-inverse-function A B f w*) *b*))
(*app f x*)
by (*rule path-concat-type*[*OF B fgb fgfx fx APF-APG APF-ETAX*])
have *APFGCOMP-APFETAX*: *path-concat B*
(*app f* (*app* (*left-inverse-function A B f w*) *b*))
(*app f* (*app* (*left-inverse-function A B f w*) (*app f x*)))
(*app f x*)
(*ap B B* (*compose B A B f* (*left-inverse-function A B f w*)) *b*
(*app f x*) (*path-inverse B* (*app f x*) *b p*))

(*ap A B f* (*app* (*left-inverse-function A B f w*) (*app f x*)) *x*
(*app* (*left-inverse-homotopy A B f w*) *x*)) ::$_h$
identity-type B (*app f* (*app* (*left-inverse-function A B f w*) *b*))
(*app f x*)
by (*rule path-concat-type*[*OF B fgb fgfx fx APFG-COMPOSE APF-ETAX*])
have *APFGCOMP-EPSFX*: *path-concat B*
(*app f* (*app* (*left-inverse-function A B f w*) *b*))
(*app f* (*app* (*left-inverse-function A B f w*) (*app f x*)))
(*app f x*)
(*ap B B* (*compose B A B f* (*left-inverse-function A B f w*)) *b*
(*app f x*) (*path-inverse B* (*app f x*) *b p*))
(*coherent-eps-point A B f w* (*app f x*)) ::$_h$
identity-type B (*app f* (*app* (*left-inverse-function A B f w*) *b*))
(*app f x*)
by (*rule path-concat-type*[*OF B fgb fgfx fx APFG-COMPOSE EPSFX*])
have *inner1*: *path-concat*
(*identity-type B* (*app f* (*app* (*left-inverse-function A B f w*) *b*))
(*app f x*))
(*ap A B f* (*app* (*left-inverse-function A B f w*) *b*) *x*
(*fiber-gamma A B f w b x p*))
(*path-concat B* (*app f* (*app* (*left-inverse-function A B f w*) *b*))
(*app f* (*app* (*left-inverse-function A B f w*) (*app f x*)))
(*app f x*)
(*ap A B f* (*app* (*left-inverse-function A B f w*) *b*)
(*app* (*left-inverse-function A B f w*) (*app f x*))
(*ap B A* (*left-inverse-function A B f w*) *b* (*app f x*)
(*path-inverse B* (*app f x*) *b p*)))
(*ap A B f* (*app* (*left-inverse-function A B f w*) (*app f x*)) *x*
(*app* (*left-inverse-homotopy A B f w*) *x*)))
(*path-concat B* (*app f* (*app* (*left-inverse-function A B f w*) *b*))
(*app f* (*app* (*left-inverse-function A B f w*) (*app f x*)))
(*app f x*)
(*ap B B* (*compose B A B f* (*left-inverse-function A B f w*)) *b*
(*app f x*) (*path-inverse B* (*app f x*) *b p*))
(*ap A B f* (*app* (*left-inverse-function A B f w*) (*app f x*)) *x*
(*app* (*left-inverse-homotopy A B f w*) *x*)))
(*ap-concat A B f* (*app* (*left-inverse-function A B f w*) *b*)
(*app* (*left-inverse-function A B f w*) (*app f x*)) *x*
(*ap B A* (*left-inverse-function A B f w*) *b* (*app f x*)
(*path-inverse B* (*app f x*) *b p*))
(*app* (*left-inverse-homotopy A B f w*) *x*))
(*path-whisker-right B*
(*app f* (*app* (*left-inverse-function A B f w*) *b*))
(*app f* (*app* (*left-inverse-function A B f w*) (*app f x*)))
(*app f x*)
(*ap A B f* (*app* (*left-inverse-function A B f w*) *b*)
(*app* (*left-inverse-function A B f w*) (*app f x*))
(*ap B A* (*left-inverse-function A B f w*) *b* (*app f x*)
(*path-inverse B* (*app f x*) *b p*)))

(*ap B B* (*compose B A B f* (*left-inverse-function A B f w*)) *b*
(*app f x*) (*path-inverse B* (*app f x*) *b p*))
(*ap A B f* (*app* (*left-inverse-function A B f w*) (*app f x*)) *x*
(*app* (*left-inverse-homotopy A B f w*) *x*))
(*path-inverse*
(*identity-type B* (*app f* (*app* (*left-inverse-function A B f w*) *b*))
(*app f* (*app* (*left-inverse-function A B f w*) (*app f x*))))
(*ap B B* (*compose B A B f* (*left-inverse-function A B f w*)) *b*
(*app f x*) (*path-inverse B* (*app f x*) *b p*))
(*ap A B f* (*app* (*left-inverse-function A B f w*) *b*)
(*app* (*left-inverse-function A B f w*) (*app f x*))
(*ap B A* (*left-inverse-function A B f w*) *b* (*app f x*)
(*path-inverse B* (*app f x*) *b p*)))
(*ap-compose B A B f* (*left-inverse-function A B f w*) *b*
(*app f x*) (*path-inverse B* (*app f x*) *b p*)))) ::$_h$
identity-type
(*identity-type B* (*app f* (*app* (*left-inverse-function A B f w*) *b*))
(*app f x*))
(*ap A B f* (*app* (*left-inverse-function A B f w*) *b*) *x*
(*fiber-gamma A B f w b x p*))
(*path-concat B* (*app f* (*app* (*left-inverse-function A B f w*) *b*))
(*app f* (*app* (*left-inverse-function A B f w*) (*app f x*)))
(*app f x*)
(*ap B B* (*compose B A B f* (*left-inverse-function A B f w*)) *b*
(*app f x*) (*path-inverse B* (*app f x*) *b p*))
(*ap A B f* (*app* (*left-inverse-function A B f w*) (*app f x*)) *x*
(*app* (*left-inverse-homotopy A B f w*) *x*)))
by (*rule path-concat-type*[*OF space APFGAM*
APFAPG-APFETAX APFGCOMP-APFETAX ACON WR])
note *CHAIN123* = *path-concat-type*[*OF space APFGAM*
APFGCOMP-APFETAX APFGCOMP-EPSFX inner1 WL]
have *NATeps*: *homotopy-naturality B B*
(*compose B A B f* (*left-inverse-function A B f w*))
(*identity-function B*)
(*coherent-eps A B f w*)
b (*app f x*) (*path-inverse B* (*app f x*) *b p*) ::$_h$
identity-type
(*identity-type B* (*app f* (*app* (*left-inverse-function A B f w*) *b*))
(*app f x*))
(*path-concat B* (*app f* (*app* (*left-inverse-function A B f w*) *b*))
(*app f* (*app* (*left-inverse-function A B f w*) (*app f x*)))
(*app f x*)
(*ap B B* (*compose B A B f* (*left-inverse-function A B f w*)) *b*
(*app f x*) (*path-inverse B* (*app f x*) *b p*))
(*coherent-eps-point A B f w* (*app f x*)))
(*path-concat B* (*app f* (*app* (*left-inverse-function A B f w*) *b*)) *b*
(*app f x*)
(*coherent-eps-point A B f w b*)
(*ap B B* (*identity-function B*) *b* (*app f x*)

```
      (path-inverse B (app f x) b p)))
  using homotopy-naturality-type[OF B B fg idB eps-hom b fx pinv] by simp
 have WHnat: path-whisker-left B
    (app f (app (left-inverse-function A B f w) b)) b (app f x)
    (coherent-eps-point A B f w b)
    (ap B B (identity-function B) b (app f x)
      (path-inverse B (app f x) b p))
    (path-inverse B (app f x) b p)
    (ap-identity B b (app f x) (path-inverse B (app f x) b p)) ::_h
    identity-type
      (identity-type B (app f (app (left-inverse-function A B f w) b))
        (app f x))
      (path-concat B (app f (app (left-inverse-function A B f w) b)) b
        (app f x)
        (coherent-eps-point A B f w b)
        (ap B B (identity-function B) b (app f x)
          (path-inverse B (app f x) b p)))
      (path-concat B (app f (app (left-inverse-function A B f w) b)) b
        (app f x)
        (coherent-eps-point A B f w b)
        (path-inverse B (app f x) b p))
  by (rule path-whisker-left-type[OF B fgb b fx EPSB ap-idQ pinv
       ap-identity-type[OF B b fx pinv]])
 have EPSB-APIDQ: path-concat B
    (app f (app (left-inverse-function A B f w) b)) b (app f x)
    (coherent-eps-point A B f w b)
    (ap B B (identity-function B) b (app f x)
      (path-inverse B (app f x) b p)) ::_h
    identity-type B (app f (app (left-inverse-function A B f w) b))
      (app f x)
  by (rule path-concat-type[OF B fgb b fx EPSB ap-idQ])
 have EPSB-Q: path-concat B
    (app f (app (left-inverse-function A B f w) b)) b (app f x)
    (coherent-eps-point A B f w b)
    (path-inverse B (app f x) b p) ::_h
    identity-type B (app f (app (left-inverse-function A B f w) b))
      (app f x)
  by (rule path-concat-type[OF B fgb b fx EPSB pinv])
 note NATeps' = path-concat-type[OF space APFGCOMP-EPSFX EPSB-APIDQ
EPSB-Q
      NATeps WHnat]
 show fiber-apf-gamma A B f w b x p ::_h
    identity-type
      (identity-type B (app f (app (left-inverse-function A B f w) b))
        (app f x))
      (ap A B f (app (left-inverse-function A B f w) b) x
        (fiber-gamma A B f w b x p))
      (path-concat B (app f (app (left-inverse-function A B f w) b)) b
        (app f x)
```

```
      (coherent-eps-point A B f w b)
      (path-inverse B (app f x) b p))
  unfolding fiber-apf-gamma-def
  by (rule path-concat-type[OF space APFGAM APFGCOMP-EPSFX EPSB-Q
      CHAIN123 NATeps'])
qed
```

The second component of the fibre path: transport of the centre proof along γ equals p.

```
definition fiber-q ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr
where
  fiber-q A B f w b x p ≡
    path-concat (identity-type B (app f x) b)
      (transport A (λa. identity-type B (app f a) b)
        (app (left-inverse-function A B f w) b) x
        (fiber-gamma A B f w b x p)
        (coherent-eps-point A B f w b))
      (path-concat B (app f x) (app f (app (left-inverse-function A B f w) b))
        b
        (path-inverse B (app f (app (left-inverse-function A B f w) b))
          (app f x)
          (ap A B f (app (left-inverse-function A B f w) b) x
            (fiber-gamma A B f w b x p)))
        (coherent-eps-point A B f w b))
      p
      (transport-identity-left A B f b
        (app (left-inverse-function A B f w) b) x
        (fiber-gamma A B f w b x p)
        (coherent-eps-point A B f w b))
      (path-concat (identity-type B (app f x) b)
        (path-concat B (app f x)
          (app f (app (left-inverse-function A B f w) b)) b
          (path-inverse B (app f (app (left-inverse-function A B f w) b))
            (app f x)
            (ap A B f (app (left-inverse-function A B f w) b) x
              (fiber-gamma A B f w b x p)))
          (coherent-eps-point A B f w b))
        (path-concat B (app f x)
          (app f (app (left-inverse-function A B f w) b)) b
          (path-inverse B (app f (app (left-inverse-function A B f w) b))
            (app f x)
            (ap A B f (app (left-inverse-function A B f w) b) x
              (fiber-gamma A B f w b x p)))
          (path-concat B (app f (app (left-inverse-function A B f w) b))
            (app f x) b
            (ap A B f (app (left-inverse-function A B f w) b) x
              (fiber-gamma A B f w b x p))
            p))
```

```
p
(path-whisker-left B (app f x)
  (app f (app (left-inverse-function A B f w) b)) b
  (path-inverse B (app f (app (left-inverse-function A B f w) b))
    (app f x)
    (ap A B f (app (left-inverse-function A B f w) b) x
      (fiber-gamma A B f w b x p)))
  (coherent-eps-point A B f w b)
  (path-concat B (app f (app (left-inverse-function A B f w) b))
    (app f x) b
    (ap A B f (app (left-inverse-function A B f w) b) x
      (fiber-gamma A B f w b x p))
    p)
  (path-concat (identity-type B
      (app f (app (left-inverse-function A B f w) b)) b)
    (coherent-eps-point A B f w b)
    (path-concat B (app f (app (left-inverse-function A B f w) b))
      (app f x) b
      (path-concat B (app f (app (left-inverse-function A B f w) b))
        b (app f x)
        (coherent-eps-point A B f w b)
        (path-inverse B (app f x) b p))
      p)
    (path-concat B (app f (app (left-inverse-function A B f w) b))
      (app f x) b
      (ap A B f (app (left-inverse-function A B f w) b) x
        (fiber-gamma A B f w b x p))
      p)
    (path-inverse (identity-type B
        (app f (app (left-inverse-function A B f w) b)) b)
      (path-concat B (app f (app (left-inverse-function A B f w) b))
        (app f x) b
        (path-concat B (app f (app (left-inverse-function A B f w) b))
          b (app f x)
          (coherent-eps-point A B f w b)
          (path-inverse B (app f x) b p))
        p)
      (coherent-eps-point A B f w b)
      (path-concat-inverse-right B
        (app f (app (left-inverse-function A B f w) b))
        (app f x) b
        (coherent-eps-point A B f w b) p))
    (path-inverse (identity-type B
        (app f (app (left-inverse-function A B f w) b)) b)
      (path-concat B (app f (app (left-inverse-function A B f w) b))
        (app f x) b
        (ap A B f (app (left-inverse-function A B f w) b) x
          (fiber-gamma A B f w b x p))
        p)
```

```
      (path-concat B (app f (app (left-inverse-function A B f w) b))
        (app f x) b
        (path-concat B (app f (app (left-inverse-function A B f w) b))
          b (app f x)
          (coherent-eps-point A B f w b)
          (path-inverse B (app f x) b p))
        p)
      (path-whisker-right B
        (app f (app (left-inverse-function A B f w) b))
        (app f x) b
        (ap A B f (app (left-inverse-function A B f w) b) x
          (fiber-gamma A B f w b x p))
        (path-concat B (app f (app (left-inverse-function A B f w) b))
          b (app f x)
          (coherent-eps-point A B f w b)
          (path-inverse B (app f x) b p))
        p
        (fiber-apf-gamma A B f w b x p)))))
    (path-inverse-concat-left B
      (app f (app (left-inverse-function A B f w) b)) (app f x) b
      (ap A B f (app (left-inverse-function A B f w) b) x
        (fiber-gamma A B f w b x p))
      p))

lemma fiber-q-type:
  assumes A: A ::_h universe i
    and B: B ::_h universe i
    and f: f ::_h arrow A B
    and w: w ::_h biinvertible A B f
    and b: b ::_h B
    and x: x ::_h A
    and p: p ::_h identity-type B (app f x) b
  shows fiber-q A B f w b x p ::_h
    identity-type (identity-type B (app f x) b)
      (transport A (λa. identity-type B (app f a) b)
        (app (left-inverse-function A B f w) b) x
        (fiber-gamma A B f w b x p)
        (coherent-eps-point A B f w b))
      p
proof −
  have g: left-inverse-function A B f w ::_h arrow B A
    by (rule left-inverse-function-type[OF w])
  have fx: app f x ::_h B by (rule arrow-elim[OF f x])
  have gb: app (left-inverse-function A B f w) b ::_h A
    by (rule arrow-elim[OF g b])
  have fgb: app f (app (left-inverse-function A B f w) b) ::_h B
    by (rule arrow-elim[OF f gb])
  have gamma: fiber-gamma A B f w b x p ::_h
      identity-type A (app (left-inverse-function A B f w) b) x
```

by (*rule fiber-gamma-type*[*OF A B f w b x p*])
have *apfg*: *ap A B f* (*app* (*left-inverse-function A B f w*) *b*) *x*
(*fiber-gamma A B f w b x p*) ::$_h$
identity-type B (*app f* (*app* (*left-inverse-function A B f w*) *b*))
(*app f x*)
by (*rule ap-type*[*OF A B f gb x gamma*])
have *apfg-inv*: *path-inverse B*
(*app f* (*app* (*left-inverse-function A B f w*) *b*)) (*app f x*)
(*ap A B f* (*app* (*left-inverse-function A B f w*) *b*) *x*
(*fiber-gamma A B f w b x p*)) ::$_h$
identity-type B (*app f x*)
(*app f* (*app* (*left-inverse-function A B f w*) *b*))
by (*rule path-inverse-type*[*OF B fgb fx apfg*])
have *epsb*: *coherent-eps-point A B f w b* ::$_h$
identity-type B (*app f* (*app* (*left-inverse-function A B f w*) *b*)) *b*
by (*rule coherent-eps-point-type*[*OF A B f w b*])
have *pinv*: *path-inverse B* (*app f x*) *b p* ::$_h$
identity-type B b (*app f x*)
by (*rule path-inverse-type*[*OF B fx b p*])
have *SPACEq*: *identity-type B* (*app f x*) *b* ::$_h$ *universe i*
by (*rule identity-form*[*OF B fx b*])
have *SPACE2*: *identity-type B* (*app f* (*app* (*left-inverse-function A B f w*) *b*))
b ::$_h$ *universe i*
by (*rule identity-form*[*OF B fgb b*])
have *transp*: *transport A* (λ*a*. *identity-type B* (*app f a*) *b*)
(*app* (*left-inverse-function A B f w*) *b*) *x*
(*fiber-gamma A B f w b x p*)
(*coherent-eps-point A B f w b*) ::$_h$
identity-type B (*app f x*) *b*
proof (*rule transport-type*[*OF A*])
fix *a*
assume *a*: *a* ::$_h$ *A*
have *fa*: *app f a* ::$_h$ *B* **by** (*rule arrow-elim*[*OF f a*])
show *identity-type B* (*app f a*) *b* ::$_h$ *universe i*
by (*rule identity-form*[*OF B fa b*])
next
show *app* (*left-inverse-function A B f w*) *b* ::$_h$ *A* **by** (*rule gb*)
next
show *x* ::$_h$ *A* **by** (*rule x*)
next
show *fiber-gamma A B f w b x p* ::$_h$
identity-type A (*app* (*left-inverse-function A B f w*) *b*) *x*
by (*rule gamma*)
next
show *coherent-eps-point A B f w b* ::$_h$
identity-type B (*app f* (*app* (*left-inverse-function A B f w*) *b*)) *b*
by (*rule epsb*)
qed
have *TILrhs*: *path-concat B* (*app f x*)

$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ b))\ b$
$(path\text{-}inverse\ B\ (app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ b))$
$(app\ f\ x)$
$(ap\ A\ B\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ b)\ x$
$(fiber\text{-}gamma\ A\ B\ f\ w\ b\ x\ p)))$
$(coherent\text{-}eps\text{-}point\ A\ B\ f\ w\ b) ::_h$
$identity\text{-}type\ B\ (app\ f\ x)\ b$
by $(rule\ path\text{-}concat\text{-}type[OF\ B\ fx\ fgb\ b\ apfg\text{-}inv\ epsb])$
have $apfg\text{-}p$: $path\text{-}concat\ B\ (app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ b))$
$(app\ f\ x)\ b$
$(ap\ A\ B\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ b)\ x$
$(fiber\text{-}gamma\ A\ B\ f\ w\ b\ x\ p))$
$p ::_h$
$identity\text{-}type\ B\ (app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ b))\ b$
by $(rule\ path\text{-}concat\text{-}type[OF\ B\ fgb\ fx\ b\ apfg\ p])$
have $epsb\text{-}pinv$: $path\text{-}concat\ B$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ b))\ b\ (app\ f\ x)$
$(coherent\text{-}eps\text{-}point\ A\ B\ f\ w\ b)$
$(path\text{-}inverse\ B\ (app\ f\ x)\ b\ p) ::_h$
$identity\text{-}type\ B\ (app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ b))$
$(app\ f\ x)$
by $(rule\ path\text{-}concat\text{-}type[OF\ B\ fgb\ b\ fx\ epsb\ pinv])$
have $epsb\text{-}pinv\text{-}p$: $path\text{-}concat\ B$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ b))\ (app\ f\ x)\ b$
$(path\text{-}concat\ B\ (app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ b))\ b$
$(app\ f\ x)$
$(coherent\text{-}eps\text{-}point\ A\ B\ f\ w\ b)$
$(path\text{-}inverse\ B\ (app\ f\ x)\ b\ p))$
$p ::_h$
$identity\text{-}type\ B\ (app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ b))\ b$
by $(rule\ path\text{-}concat\text{-}type[OF\ B\ fgb\ fx\ b\ epsb\text{-}pinv\ p])$
have TIL: $transport\text{-}identity\text{-}left\ A\ B\ f\ b$
$(app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ b)\ x$
$(fiber\text{-}gamma\ A\ B\ f\ w\ b\ x\ p)$
$(coherent\text{-}eps\text{-}point\ A\ B\ f\ w\ b) ::_h$
$identity\text{-}type\ (identity\text{-}type\ B\ (app\ f\ x)\ b)$
$(transport\ A\ (\lambda a.\ identity\text{-}type\ B\ (app\ f\ a)\ b)$
$(app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ b)\ x$
$(fiber\text{-}gamma\ A\ B\ f\ w\ b\ x\ p)$
$(coherent\text{-}eps\text{-}point\ A\ B\ f\ w\ b))$
$(path\text{-}concat\ B\ (app\ f\ x)$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ b))\ b$
$(path\text{-}inverse\ B\ (app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ b))$
$(app\ f\ x)$
$(ap\ A\ B\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ b)\ x$
$(fiber\text{-}gamma\ A\ B\ f\ w\ b\ x\ p)))$
$(coherent\text{-}eps\text{-}point\ A\ B\ f\ w\ b))$
by $(rule\ transport\text{-}identity\text{-}left\text{-}type[OF\ A\ B\ f\ b\ gb\ x\ gamma\ epsb])$
have $APFG$: $fiber\text{-}apf\text{-}gamma\ A\ B\ f\ w\ b\ x\ p ::_h$

identity-type
(*identity-type B* (*app f* (*app* (*left-inverse-function A B f w*) *b*))
(*app f x*))
(*ap A B f* (*app* (*left-inverse-function A B f w*) *b*) *x*
(*fiber-gamma A B f w b x p*))
(*path-concat B* (*app f* (*app* (*left-inverse-function A B f w*) *b*)) *b*
(*app f x*)
(*coherent-eps-point A B f w b*)
(*path-inverse B* (*app f x*) *b p*))
by (*rule fiber-apf-gamma-type*[*OF A B f w b x p*])
have *PCIR*: *path-concat-inverse-right B*
(*app f* (*app* (*left-inverse-function A B f w*) *b*)) (*app f x*) *b*
(*coherent-eps-point A B f w b*) *p* ::$_h$
identity-type
(*identity-type B* (*app f* (*app* (*left-inverse-function A B f w*) *b*)) *b*)
(*path-concat B* (*app f* (*app* (*left-inverse-function A B f w*) *b*))
(*app f x*) *b*
(*path-concat B* (*app f* (*app* (*left-inverse-function A B f w*) *b*))
b (*app f x*)
(*coherent-eps-point A B f w b*)
(*path-inverse B* (*app f x*) *b p*))
p)
(*coherent-eps-point A B f w b*)
by (*rule path-concat-inverse-right-type*[*OF B fgb fx b epsb p*])
have *WRAPFG*: *path-whisker-right B*
(*app f* (*app* (*left-inverse-function A B f w*) *b*)) (*app f x*) *b*
(*ap A B f* (*app* (*left-inverse-function A B f w*) *b*) *x*
(*fiber-gamma A B f w b x p*))
(*path-concat B* (*app f* (*app* (*left-inverse-function A B f w*) *b*)) *b*
(*app f x*)
(*coherent-eps-point A B f w b*)
(*path-inverse B* (*app f x*) *b p*))
p
(*fiber-apf-gamma A B f w b x p*) ::$_h$
identity-type
(*identity-type B* (*app f* (*app* (*left-inverse-function A B f w*) *b*)) *b*)
(*path-concat B* (*app f* (*app* (*left-inverse-function A B f w*) *b*))
(*app f x*) *b*
(*ap A B f* (*app* (*left-inverse-function A B f w*) *b*) *x*
(*fiber-gamma A B f w b x p*))
p)
(*path-concat B* (*app f* (*app* (*left-inverse-function A B f w*) *b*))
(*app f x*) *b*
(*path-concat B* (*app f* (*app* (*left-inverse-function A B f w*) *b*)) *b*
(*app f x*)
(*coherent-eps-point A B f w b*)
(*path-inverse B* (*app f x*) *b p*))
p)
by (*rule path-whisker-right-type*[*OF B fgb fx b apfg epsb-pinv p APFG*])

have *EB*: *path-concat* (*identity-type B*
(*app f* (*app* (*left-inverse-function A B f w*) *b*)) *b*)
(*coherent-eps-point A B f w b*)
(*path-concat B* (*app f* (*app* (*left-inverse-function A B f w*) *b*))
(*app f x*) *b*
(*path-concat B* (*app f* (*app* (*left-inverse-function A B f w*) *b*)) *b*
(*app f x*)
(*coherent-eps-point A B f w b*)
(*path-inverse B* (*app f x*) *b p*))
p)
(*path-concat B* (*app f* (*app* (*left-inverse-function A B f w*) *b*))
(*app f x*) *b*
(*ap A B f* (*app* (*left-inverse-function A B f w*) *b*) *x*
(*fiber-gamma A B f w b x p*))
p)
(*path-inverse* (*identity-type B*
(*app f* (*app* (*left-inverse-function A B f w*) *b*)) *b*)
(*path-concat B* (*app f* (*app* (*left-inverse-function A B f w*) *b*))
(*app f x*) *b*
(*path-concat B* (*app f* (*app* (*left-inverse-function A B f w*) *b*)) *b*
(*app f x*)
(*coherent-eps-point A B f w b*)
(*path-inverse B* (*app f x*) *b p*))
p)
(*coherent-eps-point A B f w b*)
(*path-concat-inverse-right B*
(*app f* (*app* (*left-inverse-function A B f w*) *b*)) (*app f x*) *b*
(*coherent-eps-point A B f w b*) *p*))
(*path-inverse* (*identity-type B*
(*app f* (*app* (*left-inverse-function A B f w*) *b*)) *b*)
(*path-concat B* (*app f* (*app* (*left-inverse-function A B f w*) *b*))
(*app f x*) *b*
(*ap A B f* (*app* (*left-inverse-function A B f w*) *b*) *x*
(*fiber-gamma A B f w b x p*))
p)
(*path-concat B* (*app f* (*app* (*left-inverse-function A B f w*) *b*))
(*app f x*) *b*
(*path-concat B* (*app f* (*app* (*left-inverse-function A B f w*) *b*)) *b*
(*app f x*)
(*coherent-eps-point A B f w b*)
(*path-inverse B* (*app f x*) *b p*))
p)
(*path-whisker-right B*
(*app f* (*app* (*left-inverse-function A B f w*) *b*)) (*app f x*) *b*
(*ap A B f* (*app* (*left-inverse-function A B f w*) *b*) *x*
(*fiber-gamma A B f w b x p*))
(*path-concat B* (*app f* (*app* (*left-inverse-function A B f w*) *b*)) *b*
(*app f x*)
(*coherent-eps-point A B f w b*)

```
      (path-inverse B (app f x) b p))
    p
    (fiber-apf-gamma A B f w b x p))) ::_h
  identity-type
   (identity-type B (app f (app (left-inverse-function A B f w) b)) b)
   (coherent-eps-point A B f w b)
   (path-concat B (app f (app (left-inverse-function A B f w) b))
    (app f x) b
    (ap A B f (app (left-inverse-function A B f w) b) x
     (fiber-gamma A B f w b x p))
    p)
 by (rule path-concat-type[OF SPACE2 epsb epsb-pinv-p apfg-p
     path-inverse-type[OF SPACE2 epsb-pinv-p epsb PCIR]
     path-inverse-type[OF SPACE2 apfg-p epsb-pinv-p WRAPFG]])
have WHISK-EB: path-whisker-left B (app f x)
  (app f (app (left-inverse-function A B f w) b)) b
  (path-inverse B (app f (app (left-inverse-function A B f w) b))
   (app f x)
   (ap A B f (app (left-inverse-function A B f w) b) x
    (fiber-gamma A B f w b x p)))
  (coherent-eps-point A B f w b)
  (path-concat B (app f (app (left-inverse-function A B f w) b))
   (app f x) b
   (ap A B f (app (left-inverse-function A B f w) b) x
    (fiber-gamma A B f w b x p))
   p)
  (path-concat (identity-type B
     (app f (app (left-inverse-function A B f w) b)) b)
   (coherent-eps-point A B f w b)
   (path-concat B (app f (app (left-inverse-function A B f w) b))
    (app f x) b
    (path-concat B (app f (app (left-inverse-function A B f w) b)) b
     (app f x)
     (coherent-eps-point A B f w b)
     (path-inverse B (app f x) b p))
    p)
   (path-concat B (app f (app (left-inverse-function A B f w) b))
    (app f x) b
    (ap A B f (app (left-inverse-function A B f w) b) x
     (fiber-gamma A B f w b x p))
    p)
   (path-inverse (identity-type B
     (app f (app (left-inverse-function A B f w) b)) b)
    (path-concat B (app f (app (left-inverse-function A B f w) b))
     (app f x) b
     (path-concat B (app f (app (left-inverse-function A B f w) b)) b
      (app f x)
      (coherent-eps-point A B f w b)
      (path-inverse B (app f x) b p))
```

```
      p)
    (coherent-eps-point A B f w b)
    (path-concat-inverse-right B
      (app f (app (left-inverse-function A B f w) b)) (app f x) b
      (coherent-eps-point A B f w b) p))
  (path-inverse (identity-type B
      (app f (app (left-inverse-function A B f w) b)) b)
    (path-concat B (app f (app (left-inverse-function A B f w) b))
      (app f x) b
      (ap A B f (app (left-inverse-function A B f w) b) x
        (fiber-gamma A B f w b x p))
      p)
    (path-concat B (app f (app (left-inverse-function A B f w) b))
      (app f x) b
      (path-concat B (app f (app (left-inverse-function A B f w) b)) b
        (app f x)
        (coherent-eps-point A B f w b)
        (path-inverse B (app f x) b p))
      p)
    (path-whisker-right B
      (app f (app (left-inverse-function A B f w) b)) (app f x) b
      (ap A B f (app (left-inverse-function A B f w) b) x
        (fiber-gamma A B f w b x p))
      (path-concat B (app f (app (left-inverse-function A B f w) b)) b
        (app f x)
        (coherent-eps-point A B f w b)
        (path-inverse B (app f x) b p))
      p
      (fiber-apf-gamma A B f w b x p)))) ::_h
identity-type (identity-type B (app f x) b)
  (path-concat B (app f x)
    (app f (app (left-inverse-function A B f w) b)) b
    (path-inverse B (app f (app (left-inverse-function A B f w) b))
      (app f x)
      (ap A B f (app (left-inverse-function A B f w) b) x
        (fiber-gamma A B f w b x p)))
    (coherent-eps-point A B f w b))
  (path-concat B (app f x)
    (app f (app (left-inverse-function A B f w) b)) b
    (path-inverse B (app f (app (left-inverse-function A B f w) b))
      (app f x)
      (ap A B f (app (left-inverse-function A B f w) b) x
        (fiber-gamma A B f w b x p)))
    (path-concat B (app f (app (left-inverse-function A B f w) b))
      (app f x) b
      (ap A B f (app (left-inverse-function A B f w) b) x
        (fiber-gamma A B f w b x p))
      p))
by (rule path-whisker-left-type[OF B fx fgb b apfg-inv epsb apfg-p EB])
```

have *PICL*: *path-inverse-concat-left B*
(*app f* (*app* (*left-inverse-function A B f w*) *b*)) (*app f x*) *b*
(*ap A B f* (*app* (*left-inverse-function A B f w*) *b*) *x*
(*fiber-gamma A B f w b x p*))
p ::$_h$
identity-type (*identity-type B* (*app f x*) *b*)
(*path-concat B* (*app f x*)
(*app f* (*app* (*left-inverse-function A B f w*) *b*)) *b*
(*path-inverse B* (*app f* (*app* (*left-inverse-function A B f w*) *b*))
(*app f x*)
(*ap A B f* (*app* (*left-inverse-function A B f w*) *b*) *x*
(*fiber-gamma A B f w b x p*)))
(*path-concat B* (*app f* (*app* (*left-inverse-function A B f w*) *b*))
(*app f x*) *b*
(*ap A B f* (*app* (*left-inverse-function A B f w*) *b*) *x*
(*fiber-gamma A B f w b x p*))
p))
p
by (*rule path-inverse-concat-left-type*[*OF B fgb fx b apfg p*])
have *apfg-inv-apfg-p*: *path-concat B* (*app f x*)
(*app f* (*app* (*left-inverse-function A B f w*) *b*)) *b*
(*path-inverse B* (*app f* (*app* (*left-inverse-function A B f w*) *b*))
(*app f x*)
(*ap A B f* (*app* (*left-inverse-function A B f w*) *b*) *x*
(*fiber-gamma A B f w b x p*)))
(*path-concat B* (*app f* (*app* (*left-inverse-function A B f w*) *b*))
(*app f x*) *b*
(*ap A B f* (*app* (*left-inverse-function A B f w*) *b*) *x*
(*fiber-gamma A B f w b x p*))
p) ::$_h$
identity-type B (*app f x*) *b*
by (*rule path-concat-type*[*OF B fx fgb b apfg-inv apfg-p*])
note *BIGCHAIN* = *path-concat-type*[*OF SPACEq TILrhs apfg-inv-apfg-p p WHISK-EB PICL*]
show *fiber-q A B f w b x p* ::$_h$
identity-type (*identity-type B* (*app f x*) *b*)
(*transport A* (λ*a*. *identity-type B* (*app f a*) *b*)
(*app* (*left-inverse-function A B f w*) *b*) *x*
(*fiber-gamma A B f w b x p*)
(*coherent-eps-point A B f w b*))
p
unfolding *fiber-q-def*
by (*rule path-concat-type*[*OF SPACEq transp TILrhs p TIL BIGCHAIN*])
qed

Assembling γ and q with *sigma-path* yields the path from the centre ($g\ b$, $\varepsilon\ b$) to an arbitrary fibre point (x, p).

definition *fiber-path* ::
expr ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr*

where
fiber-path A B f w b x p ≡
sigma-path A (λ*a*. *identity-type B* (*app f a*) *b*)
(*app* (*left-inverse-function A B f w*) *b*) *x*
(*coherent-eps-point A B f w b*) *p*
(*fiber-gamma A B f w b x p*)
(*fiber-q A B f w b x p*)

lemma *fiber-path-type*:
assumes *A*: *A* ::$_h$ *universe i*
and *B*: *B* ::$_h$ *universe i*
and *f*: *f* ::$_h$ *arrow A B*
and *w*: *w* ::$_h$ *biinvertible A B f*
and *b*: *b* ::$_h$ *B*
and *x*: *x* ::$_h$ *A*
and *p*: *p* ::$_h$ *identity-type B* (*app f x*) *b*
shows *fiber-path A B f w b x p* ::$_h$
identity-type (*sigma-type A* (λ*a*. *identity-type B* (*app f a*) *b*))
(*pair* (*app* (*left-inverse-function A B f w*) *b*)
(*coherent-eps-point A B f w b*))
(*pair x p*)
unfolding *fiber-path-def*
proof (*rule sigma-path-type*[*OF A*])
fix *a*
assume *a*: *a* ::$_h$ *A*
have *fa*: *app f a* ::$_h$ *B* **by** (*rule arrow-elim*[*OF f a*])
show *identity-type B* (*app f a*) *b* ::$_h$ *universe i*
by (*rule identity-form*[*OF B fa b*])
next
have *g*: *left-inverse-function A B f w* ::$_h$ *arrow B A*
by (*rule left-inverse-function-type*[*OF w*])
show *app* (*left-inverse-function A B f w*) *b* ::$_h$ *A*
by (*rule arrow-elim*[*OF g b*])
next
show *x* ::$_h$ *A* **by** (*rule x*)
next
show *coherent-eps-point A B f w b* ::$_h$
identity-type B (*app f* (*app* (*left-inverse-function A B f w*) *b*)) *b*
by (*rule coherent-eps-point-type*[*OF A B f w b*])
next
show *p* ::$_h$ *identity-type B* (*app f x*) *b* **by** (*rule p*)
next
show *fiber-gamma A B f w b x p* ::$_h$
identity-type A (*app* (*left-inverse-function A B f w*) *b*) *x*
by (*rule fiber-gamma-type*[*OF A B f w b x p*])
next
show *fiber-q A B f w b x p* ::$_h$
identity-type (*identity-type B* (*app f x*) *b*)
(*transport A* (λ*a*. *identity-type B* (*app f a*) *b*)

```
      (app (left-inverse-function A B f w) b) x
      (fiber-gamma A B f w b x p)
      (coherent-eps-point A B f w b))
    p
  by (rule fiber-q-type[OF A B f w b x p])
qed

definition biinv-fiber-contraction ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr
where
  biinv-fiber-contraction A B f w b ≡
    lambda (fiber A B f b)
      (λz. sigma-ind A (λa. identity-type B (app f a) b)
        (λz. identity-type (fiber A B f b)
          (pair (app (left-inverse-function A B f w) b)
            (coherent-eps-point A B f w b))
          z)
        (λx p. fiber-path A B f w b x p)
        z)

lemma biinv-fiber-contraction-type:
  assumes A: A ::h universe i
    and B: B ::h universe i
    and f: f ::h arrow A B
    and w: w ::h biinvertible A B f
    and b: b ::h B
  shows biinv-fiber-contraction A B f w b ::h
    pi-type (fiber A B f b)
      (λz. identity-type (fiber A B f b)
        (pair (app (left-inverse-function A B f w) b)
          (coherent-eps-point A B f w b))
        z)
  unfolding biinv-fiber-contraction-def
proof (rule pi-intro[OF fiber-form[OF A B f b]])
  fix z
  assume z: z ::h fiber A B f b
  have g: left-inverse-function A B f w ::h arrow B A
    by (rule left-inverse-function-type[OF w])
  have gb: app (left-inverse-function A B f w) b ::h A
    by (rule arrow-elim[OF g b])
  have epsb: coherent-eps-point A B f w b ::h
      identity-type B (app f (app (left-inverse-function A B f w) b)) b
    by (rule coherent-eps-point-type[OF A B f w b])
  have fibU: fiber A B f b ::h universe i
    by (rule fiber-form[OF A B f b])
  have center: pair (app (left-inverse-function A B f w) b)
      (coherent-eps-point A B f w b) ::h fiber A B f b
    unfolding fiber-def
    by (rule sigma-intro[where B=λa. identity-type B (app f a) b,
```

```
        OF gb epsb])
  show sigma-ind A (λa. identity-type B (app f a) b)
     (λz. identity-type (fiber A B f b)
       (pair (app (left-inverse-function A B f w) b)
         (coherent-eps-point A B f w b))
       z)
     (λx p. fiber-path A B f w b x p) z ::h
     identity-type (fiber A B f b)
       (pair (app (left-inverse-function A B f w) b)
         (coherent-eps-point A B f w b))
       z
  proof (rule sigma-elim[OF A])
    fix a
    assume a: a ::h A
    have fa: app f a ::h B by (rule arrow-elim[OF f a])
    show identity-type B (app f a) b ::h universe i
      by (rule identity-form[OF B fa b])
  next
    fix q
    assume q: q ::h sigma-type A (λa. identity-type B (app f a) b)
    have qfib: q ::h fiber A B f b using q unfolding fiber-def .
    show identity-type (fiber A B f b)
       (pair (app (left-inverse-function A B f w) b)
         (coherent-eps-point A B f w b))
       q ::h universe i
      by (rule identity-form[OF fibU center qfib])
  next
    fix x p
    assume x: x ::h A
    assume pp: p ::h identity-type B (app f x) b
    have fp: fiber-path A B f w b x p ::h
       identity-type (sigma-type A (λa. identity-type B (app f a) b))
         (pair (app (left-inverse-function A B f w) b)
           (coherent-eps-point A B f w b))
         (pair x p)
      by (rule fiber-path-type[OF A B f w b x pp])
    show fiber-path A B f w b x p ::h
       identity-type (fiber A B f b)
         (pair (app (left-inverse-function A B f w) b)
           (coherent-eps-point A B f w b))
         (pair x p)
      using fp unfolding fiber-def .
  next
    show z ::h sigma-type A (λa. identity-type B (app f a) b)
      using z unfolding fiber-def .
  qed
qed
```

31 Bi-invertible maps are contractible maps

definition *biinv-fiber-contractible* ::
  $expr \Rightarrow expr \Rightarrow expr \Rightarrow expr \Rightarrow expr \Rightarrow expr$
where
  biinv-fiber-contractible A B f w b $\equiv$
    pair
      (*pair* (*app* (*left-inverse-function A B f w*) *b*)
        (*coherent-eps-point A B f w b*))
      (*biinv-fiber-contraction A B f w b*)

lemma *biinv-fiber-contractible-type*:
  assumes *A*: $A ::_h$ *universe i*
    and *B*: $B ::_h$ *universe i*
    and *f*: $f ::_h$ *arrow A B*
    and *w*: $w ::_h$ *biinvertible A B f*
    and *b*: $b ::_h B$
  shows *biinv-fiber-contractible A B f w b* $::_h$ *contractible* (*fiber A B f b*)
proof −
  have *g*: *left-inverse-function A B f w* $::_h$ *arrow B A*
    by (*rule left-inverse-function-type*[*OF w*])
  have *gb*: *app* (*left-inverse-function A B f w*) *b* $::_h$ *A*
    by (*rule arrow-elim*[*OF g b*])
  have *epsb*: *coherent-eps-point A B f w b* $::_h$
      identity-type B (*app f* (*app* (*left-inverse-function A B f w*) *b*)) *b*
    by (*rule coherent-eps-point-type*[*OF A B f w b*])
  have *center*: *pair* (*app* (*left-inverse-function A B f w*) *b*)
      (*coherent-eps-point A B f w b*) $::_h$ *fiber A B f b*
    unfolding *fiber-def*
    by (*rule sigma-intro*[**where** $B{=}\lambda a.$ *identity-type B* (*app f a*) *b*,
        OF gb epsb])
  have *contr*: *biinv-fiber-contraction A B f w b* $::_h$
      pi-type (*fiber A B f b*)
        ($\lambda z.$ *identity-type* (*fiber A B f b*)
          (*pair* (*app* (*left-inverse-function A B f w*) *b*)
            (*coherent-eps-point A B f w b*))
          z)
    by (*rule biinv-fiber-contraction-type*[*OF A B f w b*])
  show *biinv-fiber-contractible A B f w b* $::_h$ *contractible* (*fiber A B f b*)
    unfolding *biinv-fiber-contractible-def contractible-def*
    by (*rule sigma-intro*[**where** $B{=}\lambda center.$
        pi-type (*fiber A B f b*)
          ($\lambda x.$ *identity-type* (*fiber A B f b*) *center x*),
        OF center contr])
qed

definition *biinvertible-to-contractible-map* ::
  $expr \Rightarrow expr \Rightarrow expr \Rightarrow expr \Rightarrow expr$
where

```
  biinvertible-to-contractible-map A B f w ≡
    lambda B (λb. biinv-fiber-contractible A B f w b)

theorem biinvertible-to-contractible-map-type:
  assumes A: A ::_h universe i
    and B: B ::_h universe i
    and f: f ::_h arrow A B
    and w: w ::_h biinvertible A B f
  shows biinvertible-to-contractible-map A B f w ::_h
    contractible-map A B f
  unfolding biinvertible-to-contractible-map-def contractible-map-def
proof (rule pi-intro[OF B])
  fix b
  assume b: b ::_h B
  show biinv-fiber-contractible A B f w b ::_h contractible (fiber A B f b)
    by (rule biinv-fiber-contractible-type[OF A B f w b])
qed

definition biinvertible-to-contractible-map-map ::
  expr ⇒ expr ⇒ expr ⇒ expr
where
  biinvertible-to-contractible-map-map A B f ≡
    lambda (biinvertible A B f)
      (λw. biinvertible-to-contractible-map A B f w)

corollary biinvertible-to-contractible-map-map-type:
  assumes A: A ::_h universe i
    and B: B ::_h universe i
    and f: f ::_h arrow A B
  shows biinvertible-to-contractible-map-map A B f ::_h
    arrow (biinvertible A B f) (contractible-map A B f)
  unfolding biinvertible-to-contractible-map-map-def
proof (rule arrow-intro[OF biinvertible-form[OF A B f]])
  fix w
  assume w: w ::_h biinvertible A B f
  show biinvertible-to-contractible-map A B f w ::_h contractible-map A B f
    by (rule biinvertible-to-contractible-map-type[OF A B f w])
qed
```

32 The bidirectional characterization

A map $f : A \to B$ is a contractible map if and only if it is bi-invertible. Both directions are witnessed by object-language functions:

- *contractible-map-to-biinvertible-map* turns a proof that every fiber is contractible into a bi-invertibility witness;
- *biinvertible-to-contractible-map-map* turns a bi-invertibility witness into a proof that every fiber is contractible.

The packaged witness is a pair of object-language implications, not an inhabitant of *equivalence*; it is named a correspondence to keep that distinction visible.

```
definition contractible-map-biinvertible-correspondence ::
  expr ⇒ expr ⇒ expr ⇒ expr
where
  contractible-map-biinvertible-correspondence A B f ≡
    pair
      (contractible-map-to-biinvertible-map A B f)
      (biinvertible-to-contractible-map-map A B f)

theorem contractible-map-biinvertible-criterion:
  assumes A: A ::h universe i
    and B: B ::h universe i
    and f: f ::h arrow A B
  shows contractible-map-biinvertible-correspondence A B f ::h
    product
      (arrow (contractible-map A B f) (biinvertible A B f))
      (arrow (biinvertible A B f) (contractible-map A B f))
  unfolding contractible-map-biinvertible-correspondence-def
  by (rule product-intro[OF
        contractible-map-to-biinvertible-map-type[OF A B f]
        biinvertible-to-contractible-map-map-type[OF A B f]])

end
theory HoTT-Equivalence-Inverse
  imports HoTT-Equivalence-Characterization
begin
```

33 Bi-inverses from quasi-inverse data

Bi-invertibility is a product of independent left and right inverse data. A quasi-inverse supplies both components at once: one map g is used twice, once with a retraction homotopy $g \circ f \sim id$ and once with a section homotopy $f \circ g \sim id$. This is the basic construction lemma for equivalence witnesses, dual to the extraction lemmas *left-inverse-function* and *right-inverse-function*.

```
definition quasi-inverse-biinvertible ::
  expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr
where
  quasi-inverse-biinvertible A B f g eta rho ≡
    pair (pair g eta) (pair g rho)

lemma quasi-inverse-biinvertible-type:
  assumes g: g ::h arrow B A
    and eta: eta ::h homotopy A (λ-. A)
```

```
    (compose A B A g f) (identity-function A)
  and rho: rho ::_h homotopy B (λ-. B)
    (compose B A B f g) (identity-function B)
 shows quasi-inverse-biinvertible A B f g eta rho ::_h
  biinvertible A B f
 unfolding quasi-inverse-biinvertible-def biinvertible-def
proof (rule product-intro)
 show pair g eta ::_h left-inverse A B f
  unfolding left-inverse-def
 proof (rule sigma-intro)
  show g ::_h arrow B A by (rule g)
 next
  show eta ::_h homotopy A (λ-. A)
      (compose A B A g f) (identity-function A)
    by (rule eta)
 qed
next
 show pair g rho ::_h right-inverse A B f
  unfolding right-inverse-def
 proof (rule sigma-intro)
  show g ::_h arrow B A by (rule g)
 next
  show rho ::_h homotopy B (λ-. B)
      (compose B A B f g) (identity-function B)
    by (rule rho)
 qed
qed

definition quasi-inverse-equivalence ::
 expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr ⇒ expr
where
 quasi-inverse-equivalence A B f g eta rho ≡
  pair f (quasi-inverse-biinvertible A B f g eta rho)

theorem quasi-inverse-equivalence-type:
 assumes f: f ::_h arrow A B
  and g: g ::_h arrow B A
  and eta: eta ::_h homotopy A (λ-. A)
    (compose A B A g f) (identity-function A)
  and rho: rho ::_h homotopy B (λ-. B)
    (compose B A B f g) (identity-function B)
 shows quasi-inverse-equivalence A B f g eta rho ::_h
  equivalence A B
 unfolding quasi-inverse-equivalence-def
 by (rule equivalence-intro[OF f
      quasi-inverse-biinvertible-type[OF g eta rho]])
```

34 The selected inverse of a bi-invertible map

Let $f : A \to B$ be bi-invertible with witness w. Its selected left inverse g already carries the retraction homotopy $g \circ f \sim id_A$, and *quasi-rho* manufactures the section homotopy $f \circ g \sim id_B$ from the independent right inverse. Reading that pair from the other side exhibits f as both a left and a right inverse of g, so g is bi-invertible.

definition *inverse-biinvertible* ::
  expr ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr*
where
  inverse-biinvertible A B f w ≡
    quasi-inverse-biinvertible B A
      (*left-inverse-function A B f w*) *f*
      (*quasi-rho A B f w*)
      (*left-inverse-homotopy A B f w*)

theorem *inverse-biinvertible-type*:
  assumes *A*: *A* ::$_h$ *universe i*
    and *B*: *B* ::$_h$ *universe i*
    and *f*: *f* ::$_h$ *arrow A B*
    and *w*: *w* ::$_h$ *biinvertible A B f*
  shows *inverse-biinvertible A B f w* ::$_h$
    biinvertible B A (*left-inverse-function A B f w*)
  unfolding *inverse-biinvertible-def*
proof (*rule quasi-inverse-biinvertible-type*)
  show *f* ::$_h$ *arrow A B* **by** (*rule f*)
next
  show *quasi-rho A B f w* ::$_h$ *homotopy B* (λ-. *B*)
      (*compose B A B f* (*left-inverse-function A B f w*))
      (*identity-function B*)
    by (*rule quasi-rho-type*[*OF A B f w*])
next
  show *left-inverse-homotopy A B f w* ::$_h$ *homotopy A* (λ-. *A*)
      (*compose A B A* (*left-inverse-function A B f w*) *f*)
      (*identity-function A*)
    by (*rule left-inverse-homotopy-type*[*OF w*])
qed

definition *inverse-contractible-map* ::
  expr ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr*
where
  inverse-contractible-map A B f w ≡
    biinvertible-to-contractible-map B A
      (*left-inverse-function A B f w*)
      (*inverse-biinvertible A B f w*)

corollary *inverse-contractible-map-type*:
  assumes *A*: *A* ::$_h$ *universe i*

and B: $B ::_h$ *universe* i
and f: $f ::_h$ *arrow* A B
and w: $w ::_h$ *biinvertible* A B f
shows *inverse-contractible-map* A B f $w ::_h$
contractible-map B A (*left-inverse-function* A B f w)
unfolding *inverse-contractible-map-def*
by (*rule biinvertible-to-contractible-map-type*[*OF* B A
left-inverse-function-type[*OF* w]
inverse-biinvertible-type[*OF* A B f w]])

35 Inverting an equivalence

definition *equivalence-inverse-function* ::
expr ⇒ *expr* ⇒ *expr* ⇒ *expr*
where
equivalence-inverse-function A B $e \equiv$
left-inverse-function A B
(*equivalence-function* e) (*equivalence-biinvertible* e)

lemma *equivalence-inverse-function-type*:
assumes e: $e ::_h$ *equivalence* A B
shows *equivalence-inverse-function* A B $e ::_h$ *arrow* B A
unfolding *equivalence-inverse-function-def*
by (*rule left-inverse-function-type*[*OF equivalence-biinvertible-type*[*OF* e]])

definition *inverse-equivalence* ::
expr ⇒ *expr* ⇒ *expr* ⇒ *expr*
where
inverse-equivalence A B $e \equiv$
pair (*equivalence-inverse-function* A B e)
(*inverse-biinvertible* A B
(*equivalence-function* e) (*equivalence-biinvertible* e))

theorem *inverse-equivalence-type*:
assumes A: $A ::_h$ *universe* i
and B: $B ::_h$ *universe* i
and e: $e ::_h$ *equivalence* A B
shows *inverse-equivalence* A B $e ::_h$ *equivalence* B A
proof −
have f: *equivalence-function* $e ::_h$ *arrow* A B
by (*rule equivalence-function-type*[*OF* e])
have w: *equivalence-biinvertible* $e ::_h$
biinvertible A B (*equivalence-function* e)
by (*rule equivalence-biinvertible-type*[*OF* e])
have g: *equivalence-inverse-function* A B $e ::_h$ *arrow* B A
by (*rule equivalence-inverse-function-type*[*OF* e])
have *witness*: *inverse-biinvertible* A B
(*equivalence-function* e) (*equivalence-biinvertible* e) $::_h$
biinvertible B A (*equivalence-inverse-function* A B e)

unfolding *equivalence-inverse-function-def*
by (*rule inverse-biinvertible-type*[*OF A B f w*])
show *inverse-equivalence A B e* ::$_h$ *equivalence B A*
unfolding *inverse-equivalence-def*
by (*rule equivalence-intro*[*OF g witness*])
qed

36 Contractibility transfers along an equivalence

Combined with *equivalence-preserves-contractible-type*, the inverse equivalence transfers contractibility backwards, so contractibility is invariant in both directions.

definition *equivalence-reflects-contractible* ::
expr ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr*
where
equivalence-reflects-contractible A B e c ≡
equivalence-preserves-contractible B A (*inverse-equivalence A B e*) *c*

theorem *equivalence-reflects-contractible-type*:
assumes *A*: *A* ::$_h$ *universe i*
and *B*: *B* ::$_h$ *universe i*
and *e*: *e* ::$_h$ *equivalence A B*
and *c*: *c* ::$_h$ *contractible B*
shows *equivalence-reflects-contractible A B e c* ::$_h$ *contractible A*
unfolding *equivalence-reflects-contractible-def*
by (*rule equivalence-preserves-contractible-type*[*OF B A*
inverse-equivalence-type[*OF A B e*] *c*])

37 Type equivalence is an equivalence relation

The identity equivalence, the inverse equivalence, and the composite equivalence make *equivalence* reflexive, symmetric, and transitive on the types of a universe. Every witness is an explicit object-language term, so the three closure properties are inhabitation results rather than meta-level assertions.

corollary *equivalence-relation-refl*:
assumes *A*: *A* ::$_h$ *universe i*
shows *identity-equivalence A* ::$_h$ *equivalence A A*
by (*rule identity-equivalence-type*[*OF A*])

corollary *equivalence-relation-symm*:
assumes *A*: *A* ::$_h$ *universe i*
and *B*: *B* ::$_h$ *universe i*
and *e*: *e* ::$_h$ *equivalence A B*
shows *inverse-equivalence A B e* ::$_h$ *equivalence B A*
by (*rule inverse-equivalence-type*[*OF A B e*])

corollary *equivalence-relation-trans*:
assumes *A*: $A ::_h$ *universe i*
and *B*: $B ::_h$ *universe i*
and *C*: $C ::_h$ *universe i*
and *e*: $e ::_h$ *equivalence A B*
and *d*: $d ::_h$ *equivalence B C*
shows *compose-equivalence A B C e d* $::_h$ *equivalence A C*
by (*rule compose-equivalence-type*[*OF A B C e d*])

end
theory *HoTT-Function-Extensionality*
imports *HoTT-Equivalence*
begin

38 The canonical map from function paths to homotopies

definition *happly* ::
expr ⇒ (*expr* ⇒ *expr*) ⇒ *expr* ⇒ *expr* ⇒
expr ⇒ *expr*
where
happly A B f g p ≡
identity-ind (*pi-type A B*) *f*
(λ*g* -. *homotopy A B f g*)
(*homotopy-refl A B f*) *g p*

lemma *happly-type*:
assumes *A*: $A ::_h$ *universe i*
and *B*: $\bigwedge x.\ x ::_h A \Longrightarrow B\ x ::_h$ *universe i*
and *f*: $f ::_h$ *pi-type A B*
and *g*: $g ::_h$ *pi-type A B*
and *p*: $p ::_h$ *identity-type* (*pi-type A B*) *f g*
shows *happly A B f g p* $::_h$ *homotopy A B f g*
unfolding *happly-def*
proof (*rule identity-elim*)
show *pi-type A B* $::_h$ *universe i*
by (*rule pi-form*[*OF A B*])
next
show *f* $::_h$ *pi-type A B* **by** (*rule f*)
next
fix *h q*
assume *h*: $h ::_h$ *pi-type A B*
assume *q* $::_h$ *identity-type* (*pi-type A B*) *f h*
show *homotopy A B f h* $::_h$ *universe i*
by (*rule homotopy-form*[*OF A B f h*])
next
show *homotopy-refl A B f* $::_h$ *homotopy A B f f*
by (*rule homotopy-refl-type*[*OF A B f*])

next
  show $g ::_h$ *pi-type A B* **by** (*rule g*)
next
  show $p ::_h$ *identity-type* (*pi-type A B*) *f g* **by** (*rule p*)
qed

lemma *happly-refl* [*simp*]:
  happly A B f f (*reflexivity* (*pi-type A B*) *f*) ≡
    homotopy-refl A B f
  unfolding *happly-def* **by** *simp*

definition *happly-map* ::
  expr ⇒ (*expr* ⇒ *expr*) ⇒ *expr* ⇒ *expr* ⇒ *expr*
where
  happly-map A B f g ≡
    lambda (*identity-type* (*pi-type A B*) *f g*)
      (λ*p*. *happly A B f g p*)

lemma *happly-map-type*:
  assumes *A*: $A ::_h$ *universe i*
    and *B*: $\bigwedge x.\ x ::_h A \Longrightarrow B\ x ::_h$ *universe i*
    and *f*: $f ::_h$ *pi-type A B*
    and *g*: $g ::_h$ *pi-type A B*
  shows *happly-map A B f g* $::_h$
    arrow (*identity-type* (*pi-type A B*) *f g*) (*homotopy A B f g*)
proof −
  have *functions*: *pi-type A B* $::_h$ *universe i*
    by (*rule pi-form*[*OF A B*])
  have *paths*: *identity-type* (*pi-type A B*) *f g* $::_h$ *universe i*
    by (*rule identity-form*[*OF functions f g*])
  show *happly-map A B f g* $::_h$
      arrow (*identity-type* (*pi-type A B*) *f g*) (*homotopy A B f g*)
    unfolding *happly-map-def*
  proof (*rule arrow-intro*[*OF paths*])
    fix *p*
    assume *p*: $p ::_h$ *identity-type* (*pi-type A B*) *f g*
    show *happly A B f g p* $::_h$ *homotopy A B f g*
      by (*rule happly-type*[*OF A B f g p*])
  qed
qed

39 Function extensionality

consts
  function-extensionality-witness ::
    expr ⇒ (*expr* ⇒ *expr*) ⇒ *expr* ⇒ *expr* ⇒ *expr*

axiomatization where
  function-extensionality:

```
    A ::_h universe i ⟹
      (⋀x. x ::_h A ⟹ B x ::_h universe i) ⟹
      f ::_h pi-type A B ⟹ g ::_h pi-type A B ⟹
      function-extensionality-witness A B f g ::_h
        biinvertible
          (identity-type (pi-type A B) f g)
          (homotopy A B f g)
          (happly-map A B f g)

definition funext ::
  expr ⇒ (expr ⇒ expr) ⇒ expr ⇒ expr ⇒
    expr ⇒ expr
where
  funext A B f g H ≡
    app
      (left-inverse-function
        (identity-type (pi-type A B) f g)
        (homotopy A B f g)
        (happly-map A B f g)
        (function-extensionality-witness A B f g))
      H

theorem funext-type:
  assumes A: A ::_h universe i
    and B: ⋀x. x ::_h A ⟹ B x ::_h universe i
    and f: f ::_h pi-type A B
    and g: g ::_h pi-type A B
    and H: H ::_h homotopy A B f g
  shows funext A B f g H ::_h
    identity-type (pi-type A B) f g
proof −
  have ext: function-extensionality-witness A B f g ::_h
      biinvertible
        (identity-type (pi-type A B) f g)
        (homotopy A B f g)
        (happly-map A B f g)
    by (rule function-extensionality[OF A B f g])
  have inverse: left-inverse-function
      (identity-type (pi-type A B) f g)
      (homotopy A B f g)
      (happly-map A B f g)
      (function-extensionality-witness A B f g)
      ::_h arrow (homotopy A B f g)
        (identity-type (pi-type A B) f g)
    by (rule left-inverse-function-type[OF ext])
  show funext A B f g H ::_h
      identity-type (pi-type A B) f g
    unfolding funext-def
    by (rule arrow-elim[OF inverse H])
```

qed

end
theory *HoTT-Univalence*
imports *HoTT-Equivalence*
begin

40 The canonical map from identity to equivalence

definition *idtoequiv* ::
level ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr*
where
idtoequiv i A B p ≡
identity-ind (*universe i*) *A*
(λ*B* -. *equivalence A B*)
(*identity-equivalence A*) *B p*

lemma *idtoequiv-type*:
assumes *A*: *A* ::$_h$ *universe i*
and *B*: *B* ::$_h$ *universe i*
and *p*: *p* ::$_h$ *identity-type* (*universe i*) *A B*
shows *idtoequiv i A B p* ::$_h$ *equivalence A B*
unfolding *idtoequiv-def*
proof (*rule identity-elim*)
show *universe i* ::$_h$ *universe* (*level-succ i*)
by (*rule universe-form*)
next
show *A* ::$_h$ *universe i* **by** (*rule A*)
next
fix *X q*
assume *X*: *X* ::$_h$ *universe i*
assume *q* ::$_h$ *identity-type* (*universe i*) *A X*
show *equivalence A X* ::$_h$ *universe i*
by (*rule equivalence-form*[*OF A X*])
next
show *identity-equivalence A* ::$_h$ *equivalence A A*
by (*rule identity-equivalence-type*[*OF A*])
next
show *B* ::$_h$ *universe i* **by** (*rule B*)
next
show *p* ::$_h$ *identity-type* (*universe i*) *A B* **by** (*rule p*)
qed

lemma *idtoequiv-refl* [*simp*]:
idtoequiv i A A (*reflexivity* (*universe i*) *A*) ≡
identity-equivalence A
unfolding *idtoequiv-def* **by** *simp*

definition *idtoequiv-map* ::

```
  level ⇒ expr ⇒ expr ⇒ expr
where
  idtoequiv-map i A B ≡
    lambda (identity-type (universe i) A B)
      (λp. idtoequiv i A B p)

lemma idtoequiv-map-type:
  assumes A: A ::h universe i
    and B: B ::h universe i
  shows idtoequiv-map i A B ::h
    arrow (identity-type (universe i) A B) (equivalence A B)
proof −
  have paths: identity-type (universe i) A B
      ::h universe (level-succ i)
    by (rule identity-form[OF universe-form A B])
  show idtoequiv-map i A B ::h
      arrow (identity-type (universe i) A B) (equivalence A B)
    unfolding idtoequiv-map-def
  proof (rule arrow-intro[OF paths])
    fix p
    assume p: p ::h identity-type (universe i) A B
    show idtoequiv i A B p ::h equivalence A B
      by (rule idtoequiv-type[OF A B p])
  qed
qed
```

41 Univalence

```
consts
  univalence-witness :: level ⇒ expr ⇒ expr ⇒ expr

axiomatization where
  univalence:
    A ::h universe i ⟹ B ::h universe i ⟹
      univalence-witness i A B ::h
        biinvertible
          (identity-type (universe i) A B)
          (equivalence A B)
          (idtoequiv-map i A B)

definition univalence-equivalence ::
  level ⇒ expr ⇒ expr ⇒ expr
where
  univalence-equivalence i A B ≡
    pair (idtoequiv-map i A B) (univalence-witness i A B)

theorem univalence-equivalence-type:
  assumes A: A ::h universe i
    and B: B ::h universe i
```

```
  shows univalence-equivalence i A B ::_h
    equivalence (identity-type (universe i) A B) (equivalence A B)
  unfolding univalence-equivalence-def
proof (rule equivalence-intro)
  show idtoequiv-map i A B ::_h
      arrow (identity-type (universe i) A B) (equivalence A B)
    by (rule idtoequiv-map-type[OF A B])
next
  show univalence-witness i A B ::_h
      biinvertible
        (identity-type (universe i) A B)
        (equivalence A B)
        (idtoequiv-map i A B)
    by (rule univalence[OF A B])
qed

end
```

References

[1] A. Bauer, J. Gross, P. L. Lumsdaine, M. Shulman, M. Sozeau, and B. Spitters. The HoTT library: A formalization of homotopy type theory in Coq. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP 2017)*, pages 164–172, New York, NY, USA, 2017. ACM. DOI: `https://doi.org/10.1145/3018610.3018615`.

[2] M. Bezem, T. Coquand, and S. Huber. A model of type theory in cubical sets. In *19th International Conference on Types for Proofs and Programs (TYPES 2013)*, volume 26 of *Leibniz International Proceedings in Informatics*, pages 107–128, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. DOI: `https://doi.org/10.4230/LIPIcs.TYPES.2013.107`.

[3] J. Chen. Homotopy type theory in Isabelle. In L. Cohen and C. Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving (ITP 2021)*, volume 193 of *Leibniz International Proceedings in Informatics*, pages 12:1–12:8, Dagstuhl, Germany, 2021. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. DOI: `https://doi.org/10.4230/LIPIcs.ITP.2021.12`.

[4] R. Harper, F. Honsell, and G. D. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993. DOI: `https://doi.org/10.1145/138027.138060`.

[5] M. Hofmann and T. Streicher. The groupoid interpretation of type theory. In *Twenty-Five Years of Constructive Type Theory*, num-

ber 36 in Oxford Logic Guides, pages 83–111. Oxford University Press, Oxford, 1998. DOI: https://doi.org/10.1093/oso/9780198501275.003.0008.

[6] K. Kapulkin and P. L. Lumsdaine. The simplicial model of univalent foundations (after Voevodsky). *Journal of the European Mathematical Society*, 23(6):2071–2126, 2021. DOI: https://doi.org/10.4171/JEMS/1050.

[7] P. Martin-Löf. *Intuitionistic Type Theory*. Number 1 in Studies in Proof Theory. Lecture Notes. Bibliopolis, Naples, 1984. Notes by Giovanni Sambin. ISBN: 88-7088-105-9.

[8] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*. Number 7 in International Series of Monographs on Computer Science. Oxford University Press, Oxford, 1990. ISBN: 0-19-853814-6.

[9] L. C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–397, 1989. DOI: https://doi.org/10.1007/BF00248324.

[10] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 199–208, Atlanta, Georgia, USA, 1988. ACM. DOI: https://doi.org/10.1145/53990.54010.

[11] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, Princeton, NJ, 2013. https://homotopytypetheory.org/book/.

[12] V. Voevodsky. An experimental library of formalized mathematics based on the univalent foundations. *Mathematical Structures in Computer Science*, 25(5):1278–1294, 2015. DOI: https://doi.org/10.1017/S0960129514000577.

[13] M. Wenzel, L. C. Paulson, and T. Nipkow. The Isabelle framework. In *Theorem Proving in Higher Order Logics (TPHOLs 2008)*, volume 5170 of *Lecture Notes in Computer Science*, pages 33–38, Berlin, Heidelberg, 2008. Springer. DOI: https://doi.org/10.1007/978-3-540-71067-7_7.