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 **13**

16 Groupoid laws **15**

17 Functorial action and homotopies **17**

18 Functorial reduction lemmas **21**

19 Whiskering and cancellation of paths **23**

20 The self-action of a homotopy to the identity **25**

21 Contractibility and fibers **28**

22 Bi-invertible maps and equivalences **29**

23 The identity equivalence **30**

24 Composition of equivalences **31**

25 Extracting contraction data **36**

26 Contractible fibers yield a bi-inverse **37**

27 Equivalences preserve contractibility **40**

28 From a bi-inverse to a single quasi-inverse **42**

29 Coherent (half-adjoint) inverse data **44**

30 Contractibility of the fibers of a bi-invertible map **54**

31 Bi-invertible maps are contractible maps **62**

32 The bidirectional characterization **63**

33 Bi-inverses from quasi-inverse data **64**

34 The selected inverse of a bi-invertible map **65**

35 Inverting an equivalence **65**

36 Contractibility transfers along an equivalence **66**

37 Type equivalence is an equivalence relation **66**

38 The canonical map from function paths to homotopies **67**

39 Function extensionality **68**

40 The canonical map from identity to equivalence **69**

41 Univalence **70**

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)
  ⟨*proof*⟩

lemma *level-asym*:
  assumes *ij*: $i \prec_l j$
    and *ji*: $j \prec_l i$
  shows *PROP W*
  ⟨*proof*⟩

lemma *level-succ-not-below*:
  assumes *level-succ* $i \prec_l i$
  shows *PROP W*
  ⟨*proof*⟩

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

assumes *level-succ* (*level-succ i*) $\prec_l$ *i*
shows *PROP W*
⟨*proof*⟩

lemma *universe-cumul-two*:
assumes *A* $::_h$ *universe i*
shows *A* $::_h$ *universe* (*level-succ* (*level-succ i*))
⟨*proof*⟩

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* ⟹
(⋀*x*. *x* $::_h$ *A* ⟹ *B x* $::_h$ *universe i*) ⟹
pi-type A B $::_h$ *universe i* **and**
pi-intro:
A $::_h$ *universe i* ⟹
(⋀*x*. *x* $::_h$ *A* ⟹ *b x* $::_h$ *B x*) ⟹
lambda A b $::_h$ *pi-type A B* **and**
pi-elim:
f $::_h$ *pi-type A B* ⟹ *a* $::_h$ *A* ⟹
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*
⟨*proof*⟩

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* $A\ B$
⟨*proof*⟩

lemma *arrow-elim*:
assumes f: $f ::_h$ *arrow* $A\ B$
and a: $a ::_h A$
shows *app* $f\ a ::_h B$
⟨*proof*⟩

lemma *product-form*:
assumes A: $A ::_h$ *universe* i
and B: $B ::_h$ *universe* i
shows *product* $A\ B ::_h$ *universe* i
⟨*proof*⟩

lemma *product-intro*:
assumes a: $a ::_h A$
and b: $b ::_h B$
shows *pair* $a\ b ::_h$ *product* $A\ B$
⟨*proof*⟩

lemma *product-first*:
assumes $p ::_h$ *product* $A\ B$
shows *first* $p ::_h A$
⟨*proof*⟩

lemma *product-second*:
assumes $p ::_h$ *product* $A\ B$
shows *second* $p ::_h B$
⟨*proof*⟩

lemma *identity-function-type*:
assumes A: $A ::_h$ *universe* i
shows *identity-function* $A ::_h$ *arrow* $A\ A$
⟨*proof*⟩

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* $A\ C$
⟨*proof*⟩

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

⟨*proof*⟩

lemma *apply-compose* [*simp*]:
app (*compose A B C g f*) *x* ≡ *app g* (*app f x*)
⟨*proof*⟩

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* ⟹ *x* ::$_h$ *A* ⟹ *y* ::$_h$ *A* ⟹
identity-type A x y ::$_h$ *universe i* **and**
identity-intro:
A ::$_h$ *universe i* ⟹ *x* ::$_h$ *A* ⟹
reflexivity A x ::$_h$ *identity-type A x x* **and**
identity-elim:
A ::$_h$ *universe i* ⟹ *x* ::$_h$ *A* ⟹
(⋀*y p*. *y* ::$_h$ *A* ⟹ *p* ::$_h$ *identity-type A x y* ⟹
C y p ::$_h$ *universe j*) ⟹
d ::$_h$ *C x* (*reflexivity A x*) ⟹
y ::$_h$ *A* ⟹ *p* ::$_h$ *identity-type A x y* ⟹
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*) ≡ *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 ≡
identity-ind A x (λ*y* -. *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* $A\ P\ x\ y\ p\ u ::_h P\ y$
⟨*proof*⟩

lemma *transport-refl* [*simp*]:
transport $A\ P\ x\ x$ (*reflexivity* $A\ x$) $u \equiv u$
⟨*proof*⟩

definition *path-inverse* ::
expr ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr*
where
path-inverse $A\ x\ y\ p \equiv$
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* $A\ y\ x$
⟨*proof*⟩

lemma *path-inverse-refl* [*simp*]:
path-inverse $A\ x\ x$ (*reflexivity* $A\ x$) ≡ *reflexivity* $A\ x$
⟨*proof*⟩

definition *path-concat* ::
expr ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒
expr ⇒ *expr* ⇒ *expr*
where
path-concat $A\ x\ y\ z\ p\ q \equiv$
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* $A\ x\ z$
⟨*proof*⟩

lemma *path-concat-refl-right* [*simp*]:
path-concat A x y y p (*reflexivity A y*) ≡ *p*
⟨*proof*⟩

16 Groupoid laws

definition *path-right-unit* ::
expr ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr*
where
path-right-unit A x y p ≡
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*⟩

definition *path-left-unit* ::
expr ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr*
where
path-left-unit A x y p ≡
identity-ind A x
(λ*y 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*
⟨*proof*⟩

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
(λ*z 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))
  ⟨proof⟩

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)
  ⟨proof⟩

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* $A\ y$)
⟨*proof*⟩

17 Functorial action and homotopies

definition *ap* ::
expr ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒
expr ⇒ *expr*
where
ap $A\ B\ f\ x\ y\ p \equiv$
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* $f\ y$)
⟨*proof*⟩

lemma *ap-refl* [*simp*]:
ap $A\ B\ f\ x\ x$ (*reflexivity* $A\ x$) ≡ *reflexivity* B (*app* $f\ x$)
⟨*proof*⟩

definition *ap-concat* ::
expr ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒
expr ⇒ *expr* ⇒ *expr* ⇒ *expr*
where
ap-concat $A\ B\ f\ x\ y\ z\ p\ q \equiv$
identity-ind $A\ y$
(λ$z\ q$. *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))
  ⟨proof⟩

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))
  ⟨proof⟩
```

definition *pair-second-map* ::
  $expr \Rightarrow (expr \Rightarrow expr) \Rightarrow expr \Rightarrow expr$
where
  $pair\text{-}second\text{-}map\ A\ B\ x \equiv lambda\ (B\ x)\ (\lambda u.\ pair\ x\ u)$

lemma *pair-second-map-type*:
  assumes A: $A ::_h universe\ i$
    and B: $\bigwedge x.\ x ::_h A \Longrightarrow B\ x ::_h universe\ i$
    and x: $x ::_h A$
  shows $pair\text{-}second\text{-}map\ A\ B\ x ::_h arrow\ (B\ x)\ (sigma\text{-}type\ A\ B)$
  ⟨*proof*⟩

definition *sigma-path* ::
  $expr \Rightarrow (expr \Rightarrow expr) \Rightarrow expr \Rightarrow expr \Rightarrow$
    $expr \Rightarrow expr \Rightarrow expr \Rightarrow expr \Rightarrow expr$
where
  $sigma\text{-}path\ A\ B\ x\ y\ u\ v\ p\ q \equiv$
    app
      (app
        ($identity\text{-}ind\ A\ x$
          ($\lambda y\ p.\ pi\text{-}type\ (B\ y)\ (\lambda v.$
            $arrow$
              ($identity\text{-}type\ (B\ y)\ (transport\ A\ B\ x\ y\ p\ u)\ v$)
              ($identity\text{-}type\ (sigma\text{-}type\ A\ B)\ (pair\ x\ u)\ (pair\ y\ v)$))))
          ($lambda\ (B\ x)\ (\lambda v.$
            $lambda\ (identity\text{-}type\ (B\ x)\ u\ v)\ (\lambda r.$
              $ap\ (B\ x)\ (sigma\text{-}type\ A\ B)\ (pair\text{-}second\text{-}map\ A\ B\ x)$
                $u\ v\ r$)))
          $y\ p$)
        v)
      q

lemma *sigma-path-type*:
  assumes A: $A ::_h universe\ i$
    and B: $\bigwedge x.\ x ::_h A \Longrightarrow B\ x ::_h universe\ i$
    and x: $x ::_h A$
    and y: $y ::_h A$
    and u: $u ::_h B\ x$
    and v: $v ::_h B\ y$
    and p: $p ::_h identity\text{-}type\ A\ x\ y$
    and q: $q ::_h$
      $identity\text{-}type\ (B\ y)\ (transport\ A\ B\ x\ y\ p\ u)\ v$
  shows $sigma\text{-}path\ A\ B\ x\ y\ u\ v\ p\ q ::_h$
    $identity\text{-}type\ (sigma\text{-}type\ A\ B)\ (pair\ x\ u)\ (pair\ y\ v)$
⟨*proof*⟩

definition *homotopy* ::
  $expr \Rightarrow (expr \Rightarrow expr) \Rightarrow expr \Rightarrow expr \Rightarrow expr$
where

homotopy A B f g ≡
pi-type A (λ*x*. *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*
⟨*proof*⟩

definition *homotopy-refl* ::
expr ⇒ (*expr* ⇒ *expr*) ⇒ *expr* ⇒ *expr*
where
homotopy-refl A B f ≡
lambda A (λ*x*. *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*
⟨*proof*⟩

definition *homotopy-concat* ::
expr ⇒ (*expr* ⇒ *expr*) ⇒ *expr* ⇒ *expr* ⇒
expr ⇒ *expr* ⇒ *expr* ⇒ *expr*
where
homotopy-concat A B f g h H K ≡
lambda A (λ*x*.
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*
⟨*proof*⟩

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*: $\bigwedge x.\ x ::_h A \Longrightarrow B\ x ::_h$ *universe i*
and *f*: $f ::_h$ *pi-type A B*
and *g*: $g ::_h$ *pi-type A B*
and *H*: $H ::_h$ *homotopy A B f g*
shows *homotopy-inverse A B f g H* $::_h$ *homotopy A B g f*
⟨*proof*⟩

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
(λ*y 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*))
⟨*proof*⟩

18 Functorial reduction lemmas

definition *ap-identity* ::
expr ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr*
where

ap-identity A x y p ≡
identity-ind A x
(λ*y 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*
⟨*proof*⟩

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
(λ*y 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*))
⟨*proof*⟩

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)
  ⟨proof⟩
```

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*
⟨*proof*⟩

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*
(λ*q′ 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′*)
⟨*proof*⟩

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*
(λ*z 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*⟩

20 The self-action of a homotopy to the identity

definition *homotopy-self* ::
expr ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr*
where
homotopy-self $A\ u\ H\ a \equiv$
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*⟩

definition *ap-congruence* ::
  $expr \Rightarrow expr \Rightarrow expr \Rightarrow expr \Rightarrow expr \Rightarrow$
    $expr \Rightarrow expr \Rightarrow expr \Rightarrow expr$
where
  ap-congruence $A\ B\ f\ x\ y\ q\ q'\ m \equiv$
    ap (*identity-type* $A\ x\ y$) (*identity-type* B (*app* $f\ x$) (*app* $f\ y$))
      (*lambda* (*identity-type* $A\ x\ y$) ($\lambda t.$ *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*⟩

definition *path-whisker-right* ::
  $expr \Rightarrow expr \Rightarrow expr \Rightarrow expr \Rightarrow expr \Rightarrow$
    $expr \Rightarrow expr \Rightarrow expr \Rightarrow expr$
where
  path-whisker-right $A\ x\ y\ z\ p\ p'\ r\ m \equiv$
    identity-ind (*identity-type* $A\ x\ y$) p
      ($\lambda p'\ m.$ *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* $A\ x\ y\ z\ p'\ r$)
  ⟨*proof*⟩

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⟩

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⟩

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
  ⟨proof⟩

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
```

⟨*proof*⟩

definition *contractible-map* ::
expr ⇒ *expr* ⇒ *expr* ⇒ *expr*
where
contractible-map A B f ≡
pi-type B (λ*b*. *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*
⟨*proof*⟩

22 Bi-invertible maps and equivalences

definition *left-inverse* ::
expr ⇒ *expr* ⇒ *expr* ⇒ *expr*
where
left-inverse A B f ≡
sigma-type (*arrow B A*)
(λ*g*. *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*)
(λ*g*. *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*) (λ*f*. *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*⟩

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*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *equivalence-form*:
assumes A: $A ::_h$ *universe i*
and B: $B ::_h$ *universe i*
shows *equivalence A B* $::_h$ *universe i*
⟨*proof*⟩

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*
⟨*proof*⟩

23 The identity equivalence

lemma *compose-identity-self* [*simp*]:
compose A A A (*identity-function A*) (*identity-function A*) ≡
identity-function A
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*

⟨*proof*⟩

lemma *right-inverse-component-type*:
assumes *w*: $w ::_h$ *biinvertible A B f*
shows *second w* $::_h$ *right-inverse A B f*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

definition *composite-left-homotopy* ::
  $expr \Rightarrow expr \Rightarrow expr \Rightarrow expr \Rightarrow expr \Rightarrow$
    $expr \Rightarrow expr \Rightarrow expr$
where
  composite-left-homotopy A B C f g wf wg $\equiv$
    lambda A ($\lambda x.$
      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*)
  ⟨*proof*⟩

definition *composite-left-inverse* ::
  $expr \Rightarrow expr \Rightarrow expr \Rightarrow expr \Rightarrow expr \Rightarrow$
    $expr \Rightarrow expr \Rightarrow expr$
where
  composite-left-inverse A B C f g wf wg $\equiv$
    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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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 (λ*z*.
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)
⟨*proof*⟩

definition *composite-right-inverse* ::
expr ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒
expr ⇒ *expr* ⇒ *expr*
where
composite-right-inverse A B C f g wf $wg \equiv$
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)
⟨*proof*⟩

definition *compose-biinvertible* ::
expr ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒
expr ⇒ *expr* ⇒ *expr*
where
compose-biinvertible A B C f g wf $wg \equiv$
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)
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *contraction-paths-type*:
assumes *c*: *c* ::$_h$ *contractible A*
shows *contraction-paths c* ::$_h$
pi-type A (*λx. identity-type A* (*contraction-center c*) *x*)
⟨*proof*⟩

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* ::
  $expr \Rightarrow expr \Rightarrow expr \Rightarrow expr \Rightarrow expr \Rightarrow expr$
where
  $fiber\text{-}contraction\ A\ B\ f\ c\ b \equiv contraction\text{-}paths\ (app\ c\ b)$

lemma *contractible-map-component-type*:
  assumes c: $c ::_h contractible\text{-}map\ A\ B\ f$
    and b: $b ::_h B$
  shows $app\ c\ b ::_h contractible\ (fiber\ A\ B\ f\ b)$
  ⟨*proof*⟩

lemma *fiber-center-type*:
  assumes c: $c ::_h contractible\text{-}map\ A\ B\ f$
    and b: $b ::_h B$
  shows $fiber\text{-}center\ A\ B\ f\ c\ b ::_h fiber\ A\ B\ f\ b$
  ⟨*proof*⟩

lemma *fiber-contraction-type*:
  assumes c: $c ::_h contractible\text{-}map\ A\ B\ f$
    and b: $b ::_h B$
  shows $fiber\text{-}contraction\ A\ B\ f\ c\ b ::_h$
    $pi\text{-}type\ (fiber\ A\ B\ f\ b)$
      $(\lambda z.\ identity\text{-}type\ (fiber\ A\ B\ f\ b)$
        $(fiber\text{-}center\ A\ B\ f\ c\ b)\ z)$
  ⟨*proof*⟩

definition *fiber-first-map* ::
  $expr \Rightarrow expr \Rightarrow expr \Rightarrow expr \Rightarrow expr$
where
  $fiber\text{-}first\text{-}map\ A\ B\ f\ b \equiv$
    $lambda\ (fiber\ A\ B\ f\ b)\ (\lambda z.\ first\ z)$

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 $fiber\text{-}first\text{-}map\ A\ B\ f\ b ::_h arrow\ (fiber\ A\ B\ f\ b)\ A$
⟨*proof*⟩

26 Contractible fibers yield a bi-inverse

definition *contractible-inverse* ::
  $expr \Rightarrow expr \Rightarrow expr \Rightarrow expr \Rightarrow expr$
where
  $contractible\text{-}inverse\ A\ B\ f\ c \equiv$
    $lambda\ B\ (\lambda b.\ first\ (fiber\text{-}center\ A\ B\ f\ c\ b))$

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*
  ⟨*proof*⟩

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*)
  ⟨*proof*⟩

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*⟩

definition *contractible-left-homotopy* ::
  expr ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr*
where
  contractible-left-homotopy A B f c ≡
    lambda A (λ*a*.
      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)
  ⟨proof⟩

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
  ⟨proof⟩

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
  ⟨proof⟩

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
  ⟨proof⟩

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)
      (λc. 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)
  ⟨proof⟩
```

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
  ⟨proof⟩

lemma equivalence-biinvertible-type:
  assumes e: e ::_h equivalence A B
  shows equivalence-biinvertible e ::_h
    biinvertible A B (equivalence-function e)
  ⟨proof⟩
```

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 (λ*b*.
      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 (λ*b*.
      identity-type B (*equivalence-contractible-center A B e c*) *b*)
  ⟨*proof*⟩

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⟩
```

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*⟩

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*⟩

definition *quasi-rho* ::
  expr ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr*
where
  quasi-rho A B f w ≡
    lambda B (λ*b*. *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*)
  ⟨*proof*⟩

lemma *quasi-rho-beta* [*simp*]:
  app (*quasi-rho A B f w*) *b* ≡ *quasi-rho-point A B f w b*
  ⟨*proof*⟩

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* ::
  $expr \Rightarrow expr \Rightarrow expr \Rightarrow expr \Rightarrow expr \Rightarrow expr$
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*⟩

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* ::
  $expr \Rightarrow expr \Rightarrow expr \Rightarrow expr \Rightarrow expr \Rightarrow expr$
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⟩
```

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⟩
```

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
```

$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))))$
$(app\ f\ a))$
$(path\text{-}concat\ B$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))))$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))$
$(app\ f\ a)$
$(ap\ A\ B\ f$
$(app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a))))$
$(app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a))$
$(app\ (left\text{-}inverse\text{-}homotopy\ A\ B\ f\ w)$
$(app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a))))$
$(quasi\text{-}rho\text{-}point\ A\ B\ f\ w\ (app\ f\ a)))$
$(path\text{-}concat\ B$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))))$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))$
$(app\ f\ a)$
$(quasi\text{-}rho\text{-}point\ A\ B\ f\ w$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a))))$
$(ap\ A\ B\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a))\ a$
$(app\ (left\text{-}inverse\text{-}homotopy\ A\ B\ f\ w)\ a)))$
$\langle proof \rangle$

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

definition *coherent-tau* ::
$expr \Rightarrow expr \Rightarrow expr \Rightarrow expr \Rightarrow expr \Rightarrow expr$
where
$coherent\text{-}tau\ A\ B\ f\ w\ a \equiv$
$path\text{-}inverse$
$(identity\text{-}type\ B\ (app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))$
$(app\ f\ a))$
$(path\text{-}concat\ B$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))))$
$(app\ f\ a)$
$(path\text{-}inverse\ B$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))))$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))$
$(quasi\text{-}rho\text{-}point\ A\ B\ f\ w$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))))$
$(path\text{-}concat\ B$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))))$
$(app\ f\ (app\ (left\text{-}inverse\text{-}function\ A\ B\ f\ w)\ (app\ f\ a)))$

```
    (app f a)
    (ap A B f
      (app (left-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*))
  ⟨*proof*⟩

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* (λ*b*. *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*)
  ⟨*proof*⟩

lemma *coherent-eps-beta* [*simp*]:
  app (*coherent-eps A B f w*) *b* ≡ *coherent-eps-point A B f w b*
  ⟨*proof*⟩

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⟩
```

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⟩
```

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⟩
```

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 (λa. 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 (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)
  ⟨proof⟩

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)
⟨*proof*⟩

31 Bi-invertible maps are contractible maps

definition *biinv-fiber-contractible* ::
expr ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *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*⟩

definition *biinvertible-to-contractible-map* ::
expr ⇒ *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr*
where
biinvertible-to-contractible-map A B f $w \equiv$
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
⟨*proof*⟩

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)
  ⟨proof⟩
```

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))
  ⟨proof⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

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*)
  ⟨*proof*⟩

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*)
  ⟨*proof*⟩

35 Inverting an equivalence

definition *equivalence-inverse-function* ::
  expr ⇒ *expr* ⇒ *expr* ⇒ *expr*
where
  equivalence-inverse-function A B e ≡

```
    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
  ⟨proof⟩

definition inverse-equivalence ::
  expr ⇒ expr ⇒ expr ⇒ expr
where
  inverse-equivalence A B e ≡
    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⟩
```

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
  ⟨proof⟩
```

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *happly-refl* [*simp*]:
happly A B f f (*reflexivity* (*pi-type A B*) *f*) ≡
homotopy-refl A B f
⟨*proof*⟩

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*: ⋀*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 *happly-map A B f g* ::$_h$
arrow (*identity-type* (*pi-type A B*) *f g*) (*homotopy A B f g*)
⟨*proof*⟩

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*: $\bigwedge x.\ x ::_h A \Longrightarrow B\ x ::_h$ *universe i*
and *f*: $f ::_h$ *pi-type A B*
and *g*: $g ::_h$ *pi-type A B*
and *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*⟩

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*
⟨*proof*⟩

lemma *idtoequiv-refl* [*simp*]:
idtoequiv i A A (*reflexivity* (*universe i*) *A*) ≡
identity-equivalence A
⟨*proof*⟩

definition *idtoequiv-map* ::
level ⇒ *expr* ⇒ *expr* ⇒ *expr*
where
idtoequiv-map i A B ≡
lambda (*identity-type* (*universe i*) *A B*)
(λ*p*. *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*⟩

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*)
⟨*proof*⟩

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*, number 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.