NOTES ON A PUSHOUT OF SETS

1. HoTT concepts

Definition 1. A **set** is a type $\mathtt{S}$ such that for any elements $x, y\colon \mathtt{S}$, and $p, q\colon x = y$, we have $p = q$.

Definition 2. Let $f\colon \mathtt{A} \to \mathtt{B}$. For any $x, y\colon \mathtt{A}$, we get a function

$$\mathtt{ap}_f\colon x =_{\mathtt{A}} y \to fx =_{\mathtt{B}} fy$$

On Identity types.

This can be interpreted in three ways:

(1) type morphisms preserve equality,
(2) functions of spaces are continuous,
(3) groupoid morphisms given functions on hom-sets.

Because $\mathtt{ap}$ preserve paths, all functions in HoTT are continuous. There are more results showing that f is functorial in that it preserves refl's and path concatenation.

Note: we can take the categorical notation and write $f(p)$ for a path $p\colon x = y$ instead of $\mathtt{ap}_f(p)$, but for now we stick with the latter.

Definition 3. Types can be defined by constructors. For example the circle type $\mathtt{S}^1$ is given by a 0-cell s and a 2-cell $p\colon s = s$.

Higher induction says that to define a map out of such a type, it suffices to define the map on the constructors. Hence a map

$$f\colon \mathtt{S}^1 \to \mathtt{A}$$

is given by $f(s)$ and $\mathtt{ap}_f(p)$.

Definition 4. Given a span

$$\begin{tikzcd} \mathtt{A} \arrow[r, "f"] \arrow[d, "g"'] & \mathtt{B} \\ \mathtt{C} & \end{tikzcd}$$

its pushout $\mathtt{B} =_{\mathtt{A}} \mathtt{C}$ is defined by

- a function $\mathtt{inl}\colon \mathtt{B} \to \mathtt{B} =_{\mathtt{A}} \mathtt{C}$
- a function $\mathtt{inr}\colon \mathtt{C} \to \mathtt{B} =_{\mathtt{A}} \mathtt{C}$
- for each $a \in \mathtt{A}$ and path $\mathtt{glue}(a)\colon fa = ga$

Hence, all functions $F\colon \mathtt{B} =_{\mathtt{A}} \mathtt{C} \to \mathtt{D}$ are given by higher induction:

- define $F(\mathtt{inl}(b))$ for all $b\colon \mathtt{B}$
- define $F(\mathtt{inr}(c))$ for all $c\colon \mathtt{C}$
- define $\mathtt{ap}_F(\mathtt{glue}(a))\colon F(\mathtt{inl}(fa)) = F(\mathtt{inr}(ga))$ for all $a\colon \mathtt{A}$

2. THE SETUP

The idea is that we have types A, B, and C, all of which are sets. The question: is the pushout given by the square

$$\begin{tikzcd}
\mathtt{A} \arrow[r, hook, "f"] \arrow[d, "g"'] & \mathtt{B} \arrow[d, "\mathtt{inl}"] \\
\mathtt{C} \arrow[r, "\mathtt{inr}"] & \mathtt{B} =_{\mathtt{A}} \mathtt{C}
\end{tikzcd}$$

also a set when f is a monomorphism?

Thus to determine whether $\mathtt{B} =_{\mathtt{A}} \mathtt{C}$ is a set, we need to access its identity types. We do this with an *encode-decode* style proof.

Roughly, a proof of this sort begins by guessing what the identity types are. That is, for each x and y in $\mathtt{B} =_{\mathtt{A}} \mathtt{C}$, we define a type

$$\mathtt{code} \colon \mathtt{B} =_{\mathtt{A}} \mathtt{C} \to \mathtt{B} =_{\mathtt{A}} \mathtt{C} \to \mathtt{Type}$$

so that $\mathtt{code}(x, y)$ serves as our guess as to what $x =_{\mathtt{B}=_{\mathtt{A}}\mathtt{C}} y$ actually is. Then we define functions

$$\mathtt{encode}_{x,y} \colon (x = y) \to \mathtt{code}(x, y) \text{ and } \mathtt{decode}_{x,y} \colon \mathtt{code}(x, y) \to (x = y)$$

for each x and y in $\mathtt{B} =_{\mathtt{A}} \mathtt{C}$. Hopefully, these are mutually inverse.

3. DEFINING `code`

Let's try to define

$$\mathtt{code}\colon \mathtt{B} =_{\mathtt{A}} \mathtt{C} \to \mathtt{B} =_{\mathtt{A}} \mathtt{C} \to \mathtt{Type}.$$

Note that `code` is a map from a pushout, so we define it using induction of higher types, as in Definition 4. Hence we need three types schemes:

$$\begin{aligned}\mathtt{code}(\mathtt{inl}(b))&\colon \mathtt{B} =_{\mathtt{A}} \mathtt{C} \to \mathtt{Type}\\ \mathtt{code}(\mathtt{inr}(c))&\colon \mathtt{B} =_{\mathtt{A}} \mathtt{C} \to \mathtt{Type}\\ \mathtt{code}(\mathtt{ap}_{\mathtt{glue}}(a))&\colon \mathtt{B} =_{\mathtt{A}} \mathtt{C} \to \mathtt{Type}\end{aligned}$$

These schemes run through $a\colon A$, $b\colon B$, and $c\colon C$. They are also functions on the same coproduct! To define $\mathtt{code}(\mathtt{inl}(b))$, we use higher induction which gives the type schemes:

$$\mathtt{code}(\mathtt{inl}(b),\mathtt{inl}(b')),\ \ \mathtt{code}(\mathtt{inl}(b),\mathtt{inr}(c')),\ \ \mathtt{code}(\mathtt{inl}(b),\mathtt{ap}_{\mathtt{glue}}(a')).$$

Similarly, we define $\mathtt{code}(\mathtt{inr}(c))$ by

$$\mathtt{code}(\mathtt{inr}(c),\mathtt{inl}(b')),\ \ \mathtt{code}(\mathtt{inr}(c),\mathtt{inr}(c')),\ \ \mathtt{code}(\mathtt{inr}(c),\mathtt{ap}_{\mathtt{glue}}(a')).$$

and $\mathtt{code}(\mathtt{glue}(a))$ by

$$\mathtt{code}(\mathtt{ap}_{\mathtt{glue}}(a),\mathtt{inl}(b')),\ \ \mathtt{code}(\mathtt{ap}_{\mathtt{glue}}(a),\mathtt{inr}(c')),\ \ \mathtt{code}(\mathtt{ap}_{\mathtt{glue}}(a),\mathtt{ap}_{\mathtt{glue}}(a')).$$

The `code`'s that have no $\mathtt{ap}_{\mathtt{glue}}$'s in the arguments correspond to our guesses for the identity types. The `code`'s that have one $\mathtt{ap}_{\mathtt{glue}}$ in the arguments give a pre- or post-composition of paths. The `code`'s that have two $\mathtt{ap}_{\mathtt{glue}}$'s in the arguments ensure that this pre- and post-composition action is coherent. This fits together in a nice little diagram:

$$\begin{tikzcd}
\mathtt{code}(\mathtt{inl}(b),\mathtt{inl}(b')) \arrow[r, "{\mathtt{code}(\mathtt{inl}(b),\mathtt{ap}_{\mathtt{glue}}(a')}"] \arrow[d, "{\mathtt{code}(\mathtt{ap}_{\mathtt{glue}}(a),\mathtt{inl}(b')}"'] \arrow[dr, phantom, "{\simeq \mathtt{code}(\mathtt{ap}_{\mathtt{glue}}(a),\mathtt{ap}_{\mathtt{glue}}(a'))}"] & \mathtt{code}(\mathtt{inl}(b),\mathtt{inr}(c')) \arrow[d, "{\mathtt{code}(\mathtt{ap}_{\mathtt{glue}}(a),\mathtt{inr}(c')}"] \\
\mathtt{code}(\mathtt{inr}(c),\mathtt{inl}(b')) \arrow[r, "{\mathtt{code}(\mathtt{inr}(c),\mathtt{ap}_{\mathtt{glue}}(a')}"'] & \mathtt{code}(\mathtt{inr}(c),\mathtt{inr}(c'))
\end{tikzcd}$$

code(b,b'). $\texttt{code}(\texttt{inl}(b), \texttt{inl}(b'))$ is the most complicated. In order to incorporate $\texttt{refl}_b$ when b is not in the image of f, we define this type to be the pushout of the span

$$\begin{tikzcd}
\sum_{a,a':A}(b =_B f(a)) \times (b' =_B f(a')) \times (b =_B b') \arrow[r, "\alpha"] \arrow[d, "\beta"] & (b =_B b') \\
\sum_{a,a':A}(b =_B f(a)) \times (b' =_B f(a')) \times (g(a) =_C g(a')) &
\end{tikzcd}$$

Here, α is a projection. Also, β is a projection of the first two factors and places $\texttt{ap}_g(p)$ in the third factor. This uses the injectivity of f to get a $p\colon a = a'$ if the upper left is populated.

This is a proposition. Indeed, the span feet are propositions and the only way for both to be populated is if the apex is also populated. But this would identify the left and right included elements with a glue.

But this can be simplified via some case analysis. If (p, q, r) is in the upper left, then since we know that we get a path $qrp^{-1} : fa =_{\texttt{B}} fa'$ which gives a $\ell : a =_{\texttt{A}} a'$ by injectivity of f. Thus we can β-reduce fa' to fa which gives us $(b =_{\texttt{B}} fa) \times (b' =_{\texttt{B}} fa) \times (b =_{\texttt{B}} b')$. Any witness to that has form (p, q', r) where $q^{-1}p : b =_{\texttt{B}} b'$ so since $\texttt{B}$ is a set, $q^{-1}p = r$ so this $(p, q', r) = (p, q', q^{-1}p)$. Since the last factor depends on the first two, we can ignore it so we reduce the upper left further to $(b =_{\texttt{B}} fa) \times (b' =_{\texttt{B}} fa)$. But maybe a nicer thing to work with is $(b =_{\texttt{B}} fa) \times (b' =_{\texttt{B}} fa') \times (a =_{\texttt{A}} a')$. Then α assembles those maps into one of form $b =_{\texttt{B}} b'$ and β applies g to the last factor. That is, we can work instead with the pushout

$$\begin{tikzcd}
\sum_{a,a':A}(b =_B fa) \times (b' =_{\texttt{B}} fa') \times (a =_{\texttt{A}} a') \arrow[r, "\alpha"] \arrow[d, "\beta"] & (b =_{\texttt{B}} b') \\
\sum_{a,a':A}(b =_{\texttt{B}} fa) \times (b' =_{\texttt{B}} fa') \times (ga =_C ga') &
\end{tikzcd}$$

We can simplify this further due to the injectivity of f. The apex of the span can be boiled down to

$$(b =_{\texttt{B}} fa) \times (b' =_{\texttt{B}} fa') \times (a =_{\texttt{A}} a')$$

is equivalent to

$$(b =_{\texttt{B}} fa) \times (b' =_{\texttt{B}} fa)$$

because if all three factors are populated, then we can β-reduce a' to a and $a =_{\texttt{A}} a$ contains only $\texttt{refl}$ because $\texttt{A}$ is a set. Thus, our pushout becomes

$$\begin{tikzcd}
\sum_{a:\texttt{A}}(b =_{\texttt{B}} fa) \times (b' =_{\texttt{B}} fa) \arrow[r, "\alpha"] \arrow[d, "\beta"] & (b =_{\texttt{B}} b') \\
\sum_{a,a':A}(b =_{\texttt{B}} fa) \times (b' =_{\texttt{B}} fa') \times (ga =_C ga') &
\end{tikzcd}$$

Later, when defining $\texttt{decode}$, we'll need to know what constructors this pushout has.

- $\sum_{a,a':A}(b =_B fa) \times (b' =_{\texttt{B}} fa') \times (a =_{\texttt{A}} a')$. Any witnesses (p, q, ℓ) assembles into a path $b =_{\texttt{B}} b'$ so α is injective. Since $\texttt{B}$ is a set, this is a proposition.

- $\sum_{a,a':A}(b =_B fa) \times (b' =_{\mathtt{B}} fa') \times (a =_{\mathtt{A}} a')$. This can only contain one element. Indeed, suppose that $(p,q,r) : (b =_B fa) \times (b' =_{\mathtt{B}} fa') \times (a =_{\mathtt{A}} a')$ and $(p',q',r') : (b =_B fa'') \times (b' =_{\mathtt{B}} fa''') \times (a'' =_{\mathtt{A}} a''')$ witness the lower left. Then p and p' assemble to witness $fa =_{\mathtt{B}} fa''$. Similarly, q and q' assemble to witness $fa' =_{\mathtt{B}} fa'''$. By injectivity of f, we get than $a =_{\mathtt{A}} a''$ and $a' =_{\mathtt{A}} a'''$. So $(b =_B fa'') \times (b' =_{\mathtt{B}} fa''') \times (a'' =_{\mathtt{A}} a''')$ beta-reduces to $(b =_B fa) \times (b' =_{\mathtt{B}} fa'') \times (a' =_{\mathtt{A}} a')$. This reduction identifies (p,q,r) and (p',q',r') because each factor is a set. So the lower left is a proposition.

From this, it follows that both α and $\beta$are injections and so the pushout is a set by some older result (ask mike). Anyway, depending on a few things we have different cases.

Remark 1.
- If $p : b =_{\mathtt{B}} b'$ neither b, b' in the image of f, then the only constructor of the pushout is $\mathtt{inl} p$.
- If $p : b =_{\mathtt{B}} b'$ and b is in the image of f, then so is b'. Then the upper left is populated, which implies the lower left is and so the constructor are the elements included from the upper right and lower left which are then glued together.
- If $b =_{\mathtt{B}} b'$ is empty is either b or b' are not in the image of f then both other corners are empty too, so no constructors.
- If $b =_{\mathtt{B}} b'$ is empty, $p : b =_{\mathtt{B}} fa$ and $q : b' =_{\mathtt{B}} fa'$, then $a =_{\mathtt{A}} a'$ must be empty, else we can prove $b =_{\mathtt{B}} b'$. Hence upper right and upper left are empty. So either $ga =_{\mathtt{C}} ga'$ is empty or not. If not, we get a constructor included from the lower left, else the pushout is empty.

code (b,c). $\mathtt{code}\,(\mathtt{inl}(b), \mathtt{inr}(c')) :\equiv \sum_{a:A}(b =_B f(a)) \times (c' =_C g(a))$ This is a proposition. Indeed, if there does not exist an $a : A$ such that $b =_B f(a)$ and $c' =_C g(a)$ are both populated, then $\mathtt{code}\,(\mathtt{inl}(b), \mathtt{inr}(c'))$ is empty. If there exists a single $a : A$ such that $b =_B f(a)$ and $c' =_C g(a)$ are both populated, then because they are each equivalent to $\mathtt{1}$, $\mathtt{code}\,(\mathtt{inl}(b), \mathtt{inr}(c'))$ is also equivalent to $\mathtt{1}$. If there is $a, a' : A$ such that $b =_B f(a)$ and $c' =_C g(a)$, and also $b =_B f(a')$ and $c' =_C g(a')$, then the injectivity of f and $f(a) =_B b =_B f(a')$ implies that $a =_A a'$ which also gives us that $\mathtt{code}\,(\mathtt{inl}(b), \mathtt{inr}(c'))$ is equivalent to $\mathtt{1}$.

code (c,b). $\texttt{code}\,(\texttt{inr}(c), \texttt{inl}(b')) :\equiv \sum_{a:A}(c =_C g(a)) \times (b' =_B f(a))$ This is a proposition by the same sort of argument from above.

code (c,c'). $\texttt{code}\,(\texttt{inr}(c), \texttt{inr}(c')) :\equiv \sum_{a,a':A}(c =_C g(a)) \times (c' =_C g(a')) \times (f(a) =_B f(a'))$ The injectivity of f gives us that $f(a) =_B f(a')$ imples that $a =_A a'$ which in turn implies that $g(a) =_C g(a')$, hence $c =_C c'$. Therefore, $\texttt{code}\,(\texttt{inr}(c), \texttt{inr}(c')) = (c =_C c')$. Hence $\texttt{code}\,(\texttt{inr}(c), \texttt{inr}(c'))$ is a proposition.

code (b, glue a). These are all equivalences, hence by univalence we define them as identity types. To show this, we show each is populated.

$\mathtt{code}\left(\mathtt{inl}(b), \mathtt{ap}_{\mathtt{glue}}(a')\right) : (\mathtt{code}(\mathtt{inl}(b), \mathtt{inl}(f(a'))) = \mathtt{code}(\mathtt{inl}(b), \mathtt{inr}(g(a'))))$ Because both sides of the identity type are propositions, to show that this equivalence holds it suffices to show that either $\mathtt{code}(\mathtt{inl}(b), \mathtt{inl}(f(a')))$ and $\mathtt{code}(\mathtt{inl}(b), \mathtt{inr}(g(a')))$ are both empty or both populated. This follows from post-composition with $\mathtt{glue(a')}$ or its inverse.

code (glue a, b). $\texttt{code}\left(\texttt{ap}_{\texttt{glue}}(a), \texttt{inl}(b')\right) : \left(\texttt{code}(\texttt{inl}(f(a)), \texttt{inl}(b')) = \texttt{code}(\texttt{inr}(g(a)), \texttt{inl}(b'))\right)$
This follows from a similar argument to that above, with post-composition replaced with pre-composition.

code (c, glue a). $\texttt{code}\left(\texttt{inr}(c), \texttt{ap}_{\texttt{glue}}(a')\right) :\equiv (\texttt{code}(\texttt{inr}(c), \texttt{inl}(f(a'))) = \texttt{code}(\texttt{inr}(c), \texttt{inr}(g(a'))))$
This follows from a similar argument.

code (glue a , c). $\texttt{code}\left(\texttt{ap}_{\texttt{glue}}(a), \texttt{inr}(c')\right) :\equiv (\texttt{code}(\texttt{inl}(f(a)), \texttt{inr}(c')) = \texttt{code}(\texttt{inr}(g(a)), \texttt{inr}(c')))$
This follows from a similar argument.

code (glue a, glue a'). $\texttt{code}(\texttt{ap}_{\texttt{glue}}(a), \texttt{ap}_{\texttt{glue}}(a'))$ is uniquely determined because everything involved is a proposition. Because we have that the 1-cells in the square are equalities, there is only a single way to commute. This single way is how we define our 2-cell.

4. DEFINING `encode`

Now that `code` is defined, we define maps between it and the identity types inside of $B +_A C$. The first map we consider is encode, which is of type

$$\mathtt{encode} : \prod_{x:\mathtt{B=_AC}} \prod_{y:\mathtt{B=_AC}} (x =_{\mathtt{B=_AC}} y) \to \mathtt{code}(x, y).$$

What are the non-empty identity types in $B +_A C$ that `encode` must map from?

- for $b, b' : B$ and $p : b =_B b'$, a path
$$\mathtt{ap}_{\mathtt{inl}}(p) : \mathtt{inl}(b) =_{\mathtt{B=_AC}} \mathtt{inl}(b')$$
- for $c, c' : C$ and $q : c =_C c'$, a path
$$\mathtt{ap}_{\mathtt{inr}}(q) : \mathtt{inr}(c) =_{\mathtt{B=_AC}} \mathtt{inl}(c')$$
- for $a : A$, a path
$$\mathtt{glue}(a) : \mathtt{inl}(fa) =_{\mathtt{B=_AC}} \mathtt{inr}(ga)$$

Thus it suffices to define the value of `encode` for $\mathtt{ap}_{\mathtt{inl}}(p)$, $\mathtt{ap}_{\mathtt{inr}}(q)$, and $\mathtt{glue}(a)$. Also, since we are mapping out of identity types, we can use path induction.

- Define
$$\mathtt{encode}\colon (\mathtt{inl}(b) =_{\mathtt{B=_AC}} \mathtt{inl}(b)) \to \mathtt{code}(\mathtt{inl}(b), \mathtt{inl}(b))$$
by $\mathtt{refl}_b \mapsto \mathtt{ap}_{\mathtt{inl}}(\mathtt{refl}_\mathtt{b})$. Note that this `inl` is coming from the span used to define $\mathtt{code}(\mathtt{inl}(b), \mathtt{inl}(b))$
- Define
$$\mathtt{encode}\colon (\mathtt{inr}(c) =_{\mathtt{B=_AC}} \mathtt{inr}(c)) \to \mathtt{code}(\mathtt{inr}(c), \mathtt{inr}(c))$$
by $\mathtt{refl}_c \mapsto \mathtt{ap}_{\mathtt{inr}}(\mathtt{refl}_c)$. Note that this `inr` is coming from the span used to define $\mathtt{code}(\mathtt{inr}(c), \mathtt{inr}(c))$.
- Define
$$\mathtt{encode}\colon (\mathtt{inl}(fa) =_{\mathtt{B=_AC}} \mathtt{inr}(ga)) \to \mathtt{code}(\mathtt{inl}(fa), \mathtt{inr}(ga))$$
by $\mathtt{glue}(a) \mapsto (\mathtt{refl}_{fa}, \mathtt{refl}_{ga})$

5. DEFINING $\mathtt{decode}$

Define

$$\mathtt{decode} : \prod_{x:\mathtt{B}=_{\mathtt{A}}\mathtt{C}} \prod_{y:\mathtt{B}=_{\mathtt{A}}\mathtt{C}} \mathtt{code}(x,y) \to (x =_{\mathtt{B}=_{\mathtt{A}}\mathtt{C}} y).$$

We can't use path induction because we are not mapping out of an identity type. The general strategy will be to give values for $\mathtt{decode}$ when feeding it the four different types of inputs— b/b', c/c', b/c, and c/b—as well as higher paths coming from $\mathtt{glue}$. Of the former four, b/b' is the most difficult because of the complicated definition for $\mathtt{code}(\mathtt{inl}b, \mathtt{inl}b')$. When dealing with $\mathtt{glue}$, we must ensure naturality, meaning that there will be commuting diagrams to check.

- The type
$$\mathtt{decode}(\mathtt{inr}c, \mathtt{inr}c')\colon \mathtt{code}(\mathtt{inr}c, \mathtt{inr}c') \to (c =_{\mathtt{B}+_{\mathtt{A}}\mathtt{C}})c')$$
is given by $p \mapsto \mathtt{ap}_{\mathtt{inl}}p$
- The type
$$\mathtt{decode}(\mathtt{inl}b, \mathtt{inr}c)\colon \mathtt{code}(\mathtt{inl}b, \mathtt{inr}c) \to (b =_{\mathtt{B}+_{\mathtt{A}}\mathtt{C}})c)$$
is trivial unless $b = fa$ and $c = ga$ hold. Define
$$\mathtt{decode}(\mathtt{inl}b, \mathtt{inr}c)\colon \mathtt{code}(\mathtt{inl}fa, \mathtt{inr}ga) \to (fa =_{\mathtt{B}+_{\mathtt{A}}\mathtt{C}})ga)$$
given by $(p,q) \mapsto (\mathtt{ap}_{\mathtt{inr}}q^{-1})(\mathtt{glue}a)(\mathtt{ap}_{\mathtt{inl}}p)$.
- The type
$$\mathtt{decode}(\mathtt{inr}c, \mathtt{inl}b)\colon \mathtt{code}(\mathtt{inr}c, \mathtt{inl}b) \to (c =_{\mathtt{B}+_{\mathtt{A}}\mathtt{C}})b)$$
is trivial unless $b = fa$ and $c = ga$ hold. Define
$$\mathtt{decode}(\mathtt{inr}c, \mathtt{inl}b)\colon \mathtt{code}(\mathtt{inr}ga, \mathtt{inl}fa) \to (ga =_{\mathtt{B}+_{\mathtt{A}}\mathtt{C}})fa)$$
given by $(q,p) \mapsto (\mathtt{ap}_{\mathtt{inl}}p^{-1})(\mathtt{glue}a)(\mathtt{ap}_{\mathtt{inr}}q)$.
- The type
$$\mathtt{decode}(\mathtt{inl}b, \mathtt{inl}b')\colon \mathtt{code}(\mathtt{inl}b, \mathtt{inl}b') \to (b =_{\mathtt{B}+_{\mathtt{A}}\mathtt{C}})b')$$
is more involved because $\mathtt{code}(\mathtt{inl}b, \mathtt{inl}b')$ is a pushout. To define a map out of a pushout, define it on the constructors. Hence, to define $\mathtt{decode}(\mathtt{inl}b, \mathtt{inl}b')$ we need to produce values for
  - $\mathtt{decode}(\mathtt{inl}b, \mathtt{inl}b')(\mathtt{inl}p)$ for $p : b =_{\mathtt{B}} b'$
  - for each $a, a' : \mathtt{A}$, define $\mathtt{decode}(\mathtt{inl}b, \mathtt{inl}b')(\mathtt{inr}(p,q,r))$ for $p : (b +_{\mathtt{B}} fa)$, $q : (b' =_{\mathtt{B}} fa')$, and $r : (ga =_{\mathtt{C}} ga')$, and
  - $\mathtt{ap}_{\mathtt{decode}(\mathtt{inl}b,\mathtt{inl}b')}(\mathtt{glue}a)$ for $a : \mathtt{A}$.

Now let's make the definitions depending on the four cases laid out in Remark 1

- Case 1. For $p : b =_{\mathtt{B}} b'$, then define $\mathtt{decode}(\mathtt{inl}b, \mathtt{inl}b')(\mathtt{inl}p) = \mathtt{ap}_{\mathtt{inl}}\mathtt{inl}p$
- Case 2. For $\alpha(p,q,r) = p(\mathtt{ap}_f r)q^{-1} : b =_{\mathtt{B}} b'$ and $\beta(p,q,r) = (p,q,\mathtt{ap}_g r) : \sum_{a,a'}(b = b') \times (b' = fa') \times (ga = ga')$, then define $\mathtt{decode}(\mathtt{inl}b, \mathtt{inl}b')(\mathtt{inl}p(\mathtt{ap}_f r)q^{-1})$ to be $\mathtt{inl}p(\mathtt{ap}_f r)q^{-1}$. Define $\mathtt{decode}(\mathtt{inl}b, \mathtt{inl}b')(\mathtt{inr}(p,q,\mathtt{ap}_g r))$ to be $(\mathtt{inr}q'^{-1})(\mathtt{glue}a')^{-1}(\mathtt{ap}_g r)(\mathtt{glue}a)$ This is well defined because there is a $\mathtt{glue}(p,q,r)$ connecting $\mathtt{inl}(p(\mathtt{ap}_f r)q^{-1})$ to $\mathtt{inr}(p,q,\mathtt{ap}_g r)$ in the $\mathtt{code}(\mathtt{inl}b, \mathtt{inl}b')$ definition which we apply $\mathtt{decode}$ to, identifying their images.
- Case 3. Nothing to do, no constructors.

- Case 4. There is only one constructor, $\texttt{inr}(p, q, \texttt{ap}_g r)$. Define $\texttt{decode}(\texttt{inl}b, \texttt{inl}b')(\texttt{inr}(p, q, \texttt{ap}_g r))$ to be the path $(\texttt{inr}q'^{-1})(\texttt{glue}a')^{-1}(\texttt{ap}_g r)(\texttt{glue}a)(\texttt{inr}p)$.

Showing $\texttt{decode}$ is well defined, we must have that

$$\texttt{decode}(\texttt{code}(\texttt{inl}fa, \texttt{inl}b)) = \texttt{decode}(\texttt{code}(\texttt{inr}ga, \texttt{inl}b)).$$

Recall,

$$\texttt{code}(\texttt{inr}ga, \texttt{inl}b) :\equiv \Sigma_{x:\texttt{A}}(ga =_{\texttt{C}} gx) \times (b =_{\texttt{B}} fx)$$

and

$$\texttt{code}(\texttt{inl}fa, \texttt{inl}b)$$

is the pushout of

$$\begin{tikzcd}
\sum_{x,x':\texttt{A}}(fa =_{\texttt{B}} fx) \times (b =_{\texttt{B}} fx') \times (fa =_{\texttt{B}} b) \arrow[r, "\alpha"] \arrow[d, "\beta"'] & (fa =_{\texttt{B}} b) \\
\sum_{x,x':\texttt{A}}(fa =_{\texttt{B}} fx) \times (b =_{\texttt{B}} fx') \times (gx =_{\texttt{C}} gx') &
\end{tikzcd}$$

Clearly, $\texttt{code}(\texttt{inr}ga, \texttt{inl}b)$ it empty unless $b =_{\texttt{B}} fa'$. The same can be said for $\texttt{code}(\texttt{inl}fa, \texttt{inl}b)$ because the constructors of the pushout require either a populated $fa =_{\texttt{B}} b$ or a populated $b =_{\texttt{B}} fx'$ for any $x' : \texttt{A}$. So, we might as well assume that $b =_{\texttt{B}} fa$. Why not $b =_{\texttt{B}} fa'$ for some other $a' : \texttt{A}$? I should think about this later.

Now, we have that

$$\texttt{code}(\texttt{inr}ga, \texttt{inl}fa) :\equiv \Sigma_{x:\texttt{A}}(ga =_{\texttt{C}} gx) \times (fa =_{\texttt{B}} fx)$$

and

$$\texttt{code}(\texttt{inl}fa, \texttt{inl}fa)$$

is the pushout of

$$\begin{tikzcd}
\sum_{x,x':\texttt{A}}(fa =_{\texttt{B}} fx) \times (fa =_{\texttt{B}} fx') \times (fa =_{\texttt{B}} fa) \arrow[r, "\alpha"] \arrow[d, "\beta"'] & (fa =_{\texttt{B}} fa) \\
\sum_{x,x':\texttt{A}}(fa =_{\texttt{B}} fx) \times (fa =_{\texttt{B}} fx') \times (gx =_{\texttt{C}} gx') &
\end{tikzcd}$$

which should just be $\{\texttt{refl}_{fa}\}$ since $\texttt{B}$ is a set. It follows that $\texttt{decode}(\texttt{code}(\texttt{inl}fa, \texttt{inl}fa))$ maps $\texttt{refl}_{fa}$ to $(\texttt{refl}_{fa})$ as required. We must now show that $\texttt{decode}(\texttt{code}(\texttt{inr}ga, \texttt{inl}fa))$ is also $\texttt{refl}_{fa}$. But this follows from

$$\texttt{code}(\texttt{inr}ga, \texttt{inl}fa) = (ga =_{\texttt{C}} ga) \times (fa_{\texttt{B}} fa) = \{(\texttt{refl}_{ga}, \texttt{refl}_{fa})$$

which is mapped via $\texttt{decode}$ to $(\texttt{ap}_{\texttt{inl}}\texttt{refl}_{fa})(\texttt{glue}a)(\texttt{ap}_{\texttt{inr}}\texttt{refl}_{ga}$, which is equal to $\texttt{glue}a$. But since $\texttt{glue}a$ is contractible, it is homotopic to $\texttt{refl}_{fa}$.