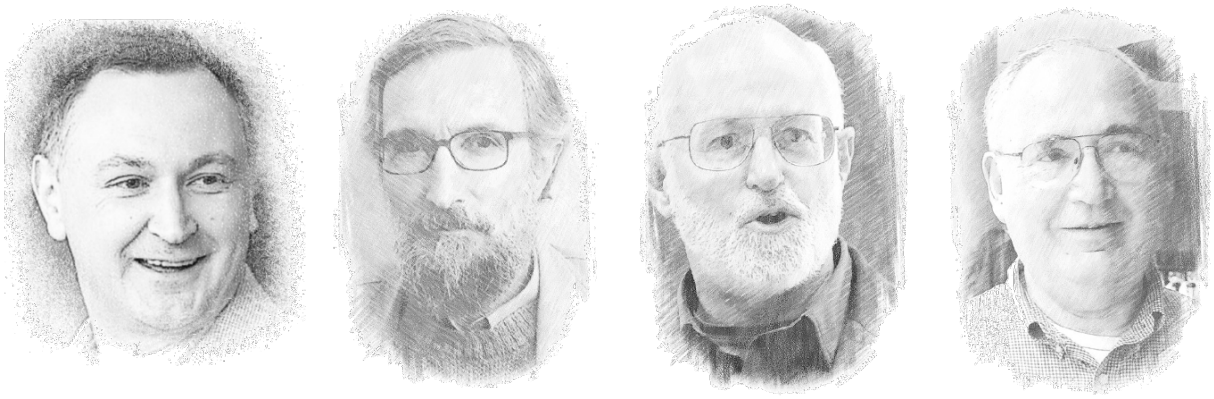
RedPRL Documentation

The RedPRL Development Team

Oct 10, 2018

Contents



Red**PRL** is an experimental proof assistant based on cubical computational type theory, which extends the Nuprl semantics by higher-dimensional features inspired by homotopy type theory. Red**PRL** is created and maintained by the RedPRL Development Team.

Red**PRL** is written in Standard ML, and is available for download on GitHub.

CHAPTER 1

Features

- computational canonicity and extraction
- univalence as a theorem
- strict (exact) equality types
- coequalizer and pushout types
- functional extensionality
- equality reflection
- proof tactics

CHAPTER 2

Papers & Talks

- Angiuli, Favonia, Harper. Cartesian Cubical Computational Type Theory: Constructive Reasoning with Paths and Equalities. CSL 2018.
- Angiuli, Cavallo, Favonia, Harper, Sterling. The RedPRL Proof Assistant. LFMTP 2018 (Invited Paper).
- Favonia. Cubical Computational Type Theory & RedPRL. 2018.
- Harper, Angiuli. Computational (Higher) Type Theory. ACM POPL Tutorial Session 2018.
- Sterling, Harper. Algebraic Foundations of Proof Refinement. Draft, 2016.

CHAPTER 3

RedPRL User Guide

3.1 Tutorial

We will walk through parts of `examples/tutorial.prl`, which was the live demo of Red**PRL** in our POPL 2018 tutorial on Computational (Higher) Type Theory. For further guidance, we recommend that new users consult the many other proofs in the `examples/` subdirectory.

Red**PRL** is a program logic for a functional programming language extended with constructs for higher-dimensional reasoning. A proof in Red**PRL** is a tactic script that constructs a program (or *extract*) and demonstrates that it inhabits the specified type.

3.1.1 Getting started

Let's start by defining the function that negates a boolean:

```
theorem Not :
  (-> bool bool)
by {
  lam b =>
  if b then `ff else `tt
}.
```

The `lam b =>` tactic introduces a variable `b : bool` in the context, `if then else` performs a case split, and each branch is resolved by a boolean literal (`` `ff `` and `` `tt ``). We can inspect the proof state at any point using a *hole*. Replace `` `ff `` with `?the-tt-case`, and run Red**PRL** again:

```
?the-tt-case.
Goal #0.
  b : bool
  ---------
  bool
```

That is, the current subgoal has type `bool`, and `b : bool` is in scope. Replace `?the-tt-case` with `` `ff `` once again, and follow this theorem by:

```
print Not.
```

The `print` command displays the theorem statement and its extract. In this case, we can prove `Not` with the extract directly:

```
theorem NotDirect :
  (-> bool bool)
by {
  `(lam [b] (if [_] bool b ff tt))
}.
```

In general, we might not have a particular extract in mind, or establishing the type of that extract may require non-trivial reasoning, so we typically choose (or are forced) to use interactive tactics rather than specifying extracts.

Red**PRL** has a notion of exact, extensional equality of programs, written =. For example, applying `Not` twice is equal to the identity function. (Function application is written `$`.)

```
theorem NotNot :
  (->
   [b : bool]
   (= bool ($ Not ($ Not b)) b))
by {
  lam b => auto
}.
```

This instance of `auto` cases on `b`, and in each case simplifies the left-hand side and supplies a reflexive equality. (For example, the subgoal `(= bool tt tt)` is handled by the *refinement rule* `refine bool/eq/tt`.)

Families of types respect equality of indices. For example, suppose we have a boolean-indexed family of types `family`. By virtue of the equation we just proved, an element of the type `($ family b)` is also an element of the type `($ family ($ Not ($ Not b)))`:

```
theorem RespectEquality :
  (->
   [family : (-> [b : bool] (U 0))]
   [b : bool]
   ($ family b)
   ($ family ($ Not ($ Not b))))
by {
  lam family b pf =>
  rewrite ($ NotNot b);        // equation to rewrite along
  [ with b' => `($ family b') // what to rewrite (i.e., b')
  , use pf
  ];
  auto
}.
```

Here, `(U 0)` is a universe (the "type of small types"). The `rewrite` tactic rewrites the argument to `family` along the equality `(= bool ($ Not ($ Not b)) b)` given by `($ NotNot b)`, taking `pf : ($ family b)` to a proof of `($ family ($ Not ($ Not b)))`.

Surprisingly, the extract is just a constant function: `(lam [x0 x1 x2] x2)`! The reason is that at runtime, for any particular `b`, the types `($ family b)` and `($ family ($ Not ($ Not b)))` will be exactly the same, so there's no need for a coercion.

3.1.2 Cubical reasoning

Red**PRL** also includes a notion of *path* similar to the *identity type* of homotopy type theory. Like equations, paths are respected by the constructs of type theory. However, while respect for equations is silent, respect for paths affects the runtime behavior of programs.

In Red**PRL**, paths are mediated by *dimension variables* abstractly representing how the path varies over an interval. Nested paths of paths are indexed by multiple dimension variables, and therefore trace out squares, cubes, hypercubes, etc., hence the name *cubical type theory*. A reflexive path depends degenerately on a dimension variable:

```
theorem Refl :
  (->
   [ty : (U 0)]
   [a : ty]
   (path [_] ty a a))
by {
  lam ty a =>
  abs _ => `a
}.
```

The `abs _ =>` tactic is analogous to `lam a =>` but introduces dimension variables rather than ordinary variables.

Paths form a groupoid: they can be composed and reversed; composition is associative (up to a path) and has `Refl` as unit (up to a path); etc. These operations all follow from a single operation, *homogeneous Kan composition* (`hcom`), which produces the fourth side of a square given the other three, or the sixth side of a cube given the other five, etc. The details of this operation are beyond the scope of this tutorial, but the following illustration demonstrates how to compose paths `p` and `q` using `hcom`:

```
 --- x          p
|           ---------
y          |         |
         a |         | q
           |         |
           a.........c
```

That is, if `p` goes from `a` to `b`, and `q` goes from `b` to `c`, then we can form a square with `p` on top, `q` on the right, and the constantly-`a` path on the left; the bottom must therefore be a path from `a` to `c`. The concrete notation is given below (where `(@ p x)` applies the path `p` to the dimension variable `x` as argument).

```
theorem PathConcat :
  (->
   [ty : (U 0 kan)]
   [a b c : ty]
   [p : (path [_] ty a b)]
   [q : (path [_] ty b c)]
   (path [_] ty a c))
by {
  lam ty a b c p q =>
  abs x =>
  `(hcom 0~>1 ty (@ p x) [x=0 [_] a] [x=1 [y] (@ q y)])
}.
```

Another source of paths is Voevodsky's *univalence principle*, stating that any equivalence (isomorphism-up-to-paths) between types gives rise to a path between those types. We apply this principle to the isomorphism between `(-> bool ty)` and `(* ty ty)` sending a function to the pair `({ , })` of its output on `tt` and `ff`.

```
theorem FunToPair :
  (->
   [ty : (U 0 kan)]
   (-> bool ty)
   (* ty ty))
by {
  lam ty fun =>
  {`($ fun tt), `($ fun ff)}
}.

theorem PathFunToPair :
  (->
   [ty : (U 0 kan)]
   (path [_] (U 0 kan) (-> bool ty) (* ty ty)))
by {
  lam ty => abs x =>
  // see tutorial.prl for omitted proofs
}.
```

Respect for paths follows from an explicit *coercion* operation (`coe`). We can coerce along the path `($ PathFunToPair ty)` from left to right (`0~>1`), taking an element of `(bool -> ty)` to `(ty * ty)`.

```
theorem RespectPaths :
  (->
   [ty : (U 0 kan)]
   (-> bool ty)
   (* ty ty))
by {
  lam ty fun =>
  `(coe 0~>1 [x] (@ ($ PathFunToPair ty) x) fun)
}.
```

Unlike `rewrite`, uses of `coe` are reflected in the extract, because they affect computation. Indeed, an element of `(bool -> ty)` is not literally an element of `(ty * ty)`, and there is more than one isomorphism between these types! A major benefit of cubical type theory over homotopy type theory is that coercions actually *compute*: if we apply `RespectPaths` to the identity function, we get exactly the pair `{`tt,`ff}`.

```
theorem ComputeCoercion :
  (=
   (* bool bool)
   ($ RespectPaths bool (lam [b] b))
   (tuple [proj1 tt] [proj2 ff]))
by {
  auto
}.
```

Experts: though paths in Red**PRL** are defined by dimension variables rather than the `refl` and `J` operators of homotopy type theory, `J` is definable using coercion and homogeneous Kan composition (but will not compute to `d` on `refl`).

```
theorem J(#l:lvl) : // parametrized over any universe level #l
  (->
   [ty : (U #l kan)]
   [a : ty]
   [fam : (-> [x : ty] (path [_] ty a x) (U #l kan))]
   [d : ($ fam a (abs [_] a))]
   [x : ty]
```

(continues on next page)

(continued from previous page)

```
   [p : (path [_] ty a x)]
   ($ fam x p))
by {
  lam ty a fam d x p =>
  `(coe 0~>1
    [i] ($ fam
           (hcom 0~>1 ty a [i=0 [_] a] [i=1 [j] (@ p j)])
           (abs [j] (hcom 0~>j ty a [i=0 [_] a] [i=1 [j] (@ p j)])))) d)
}.
```

3.2 Language reference

RedPRL documents contain expressions written in multiple languages: the *top-level vernacular*, the *object language*, and the *tactic language*.

3.2.1 Top-level vernacular

The top-level vernacular is a very simple language of commands that interact with the *signature*: this language is for declaring *new theorems*, *definitional extensions* and *tactics*; the top-level vernacular can also be used to print out an object from the signature. This is the language that one writes in a `.prl` file.

Defining theorems

A *theorem* in RedPRL is given by a type (an object language expression) together with a tactic script which establishes that the given type is inhabited; when a theorem is declared, the tactic script is executed against the goal, and if the result is successful, the generated evidence is added to the signature.

```
theorem OpName(#p : ...) :
  // goal here (object language expression)
by {
  // script here (tactic expression)
}.
```

Most definitions in a RedPRL signature will take the form of theorems; but other forms of definition may be preferable, *depending on circumstances*.

Defining new operators

The most primitive way to define a new operator in RedPRL is to use the `define` command. A definition is specified by giving an operator name (which must be capitalized), together with a (possibly empty) sequence of parameters together with their valences, and an object-language term which shall be the definiens:

```
define OpName(#p : [dim].exp, ...) : exp =
  // object language expression here
.
```

A parameter is referenced using a *metavariable* (which is distinguished syntactically using the `#` sigil); the valence of a parameter specifies binding structure, with `[tau1,tau2].tau` being the valence of a binder of sort `tau` that binds a variable of sort `tau1` and a variable of sort `tau2`.

A simple definition of sort `exp` without parameters can be abbreviated as follows:

```
define OpName =
  // object language expression here
.
```

Definitions of this kind are not subject to any typing conditions in CHTT; instead, if you use a primitive definition within a proof, you will have to prove that it is well-typed.

Defining tactics

A tactic can be defined using the special `tactic` command:

```
tactic OpName(#p : ...) =
  // tactic expression here
.
```

This desugars to an instance of the `define` command, and differs only in that the body of the definiens is here parsed using the grammar of tactic expressions.

Printing objects

To print a previously-defined object from the signature, one can write the following command:

```
print OpName.
```

When to use theorems or definitions?

As a rule of thumb, in most cases it is simpler to interactively construct an element of a type using a `theorem` declaration than it is to define a code for an element, and then prove that it has the intended type. This is why theorems are usually preferred to definitions in Red**PRL**.

However, definitions may be preferable in some cases; consider the definition of an abbreviation for the type family `(lam [ty] (-> nat ty))` of sequences. As a theorem, this definition must take a universe level as a parameter

```
define Sequence(#l : lvl) :
  (-> [ty : (U #l)] (U #l))
by {
  // apply function introduction rule in the tactic language
  lam ty =>
    // explicitly give the body of the function in the object language
    `(-> nat ty)
}.
```

Later, when using this definition, one would have to explicitly provide the universe level, even though it does not play a part in the actual defined object: for instance, `(Sequence #lvl{0})`. The parameter was present only in order to express the type of the type family. On the other hand, with a definition, we can write the following:

```
define Sequence =
  (lam [ty] (-> nat ty))
.
```

One advantage of theorems over definitions is that Red**PRL** knows their type intrinsically; whereas definitions must be unfolded and proved to be well-typed at each use-site.

3.2.2 Object language

Red**PRL**'s object language and tactic language share a common syntactic framework based on multi-sorted second-order abstract syntax, which provides a uniform treatment of binding with syntactic sorts. Red**PRL** has three main sorts: `exp` (the sort of expressions), `dim` (the sort of dimension expressions) and `tac` (the sort of tactic expressions).

The object language is written in a variant of s-expression notation, with binding operators written systematically in the style of `(lam [x] x)`. An expression in the object language is an *untyped program* or *realizer* in the language of Computational Higher Type Theory (CHTT).

These expressions include ordinary programming constructs like lambda abstraction and application, records, projection, etc., as well as cubical programming constructs inspired by cubical sets. Below are summarized common forms overlapping with other calculi.

Ordinary Operation	Expression
dependent function type	`(-> [x y ... : ty] ... ty)`
lambda abstraction	`(lam [x y ...] e)`
function application	`($ f e1 e2 ...)`
dependent record type	`(record [lbl ... : ty] ..)`
tuple (record element)	`(tuple [lbl e] ...)`
record projection	`(! lbl e)`

The cubical extension is characterized by a new sort of expressions, *dimension expressions* along with many new operations. A dimension expression can be a dimension variable `i`, representing an interval, or a dimension constant `0` or `1`, representing one of its end point.

Cubical Operation	Expression
coercion	`(coe r~>s [i] ty e)`
homogeneous composition	`(hcom r~>s ty cap [i=0 [j] tube0] ...)`
path type	`(path [i] ty e0 e1)`
line type	`(-> [i : dim] ... ty)`
path/line abstraction	`(abs [i j ...] e)`
path/line application	`(@ e r1 r2 ...)`
univalence	`(V a b e)`

Todo: Finish summary of object language terms.

3.2.3 Tactic language

Todo: Summarize tactic language

3.3 Atomic judgments

Red**PRL** currently has five forms of atomic (non-hypothetical) judgments that may appear in subgoals.

1. *Truth* asserts that a type is inhabited.
2. *Type equality* asserts an equality between two types.

3. *Subtyping* asserts a subtyping relation.
4. *Subkinding* asserts that some type is actually a universe in which all types has a particular kind.
5. *Term* lets the user give an expression.

Note that these judgment forms differ from our semantic presentations in papers.

3.3.1 Truth

A *truth* judgment

```
a true
```

or simply

```
a
```

means `a` is an inhabited type. Any inhabitant can realize this judgment. For example, the expression `1` realizes

```
int
```

because `1` is in the type `int`. This is commonly used to state a theorem or specify the type of the program to be implemented. In fact, all top-level theorems (see *Defining theorems*) must be in this judgmental form.

3.3.2 Type equality

A *type equality* judgment

```
a = b type
```

means `a` are `b` are equal types (without regard to universe level), and its realizer must be `ax`, the same as the realizer of equality types. For example, we have

```
int = int type
```

realized by `ax`. Multiverses are supported through kind markers such as `kan` or `discrete`:

```
a = b discrete type
a = b kan type
a = b coe type
a = b hcom type
a = b pre type
```

where `a = b kan type` means `a` and `b` are equal Kan types. (The judgment `a = b type` is really an abbreviation of `a = b pre type` because `pre` is the default kind.) Following the PRL family of proof assistants which use partial equivalence relations, well-typedness is defined as the equality of the type and itself; to save some typing, `a type` stands for `a = a type` and `a kan type` stands for `a = a kan type`.

In the presence of universes and equality types, one might wonder why we still have a dedicated judgmental form for type equality. That is, one may intuitively treat the judgment

```
a = b type
```

as `(= (U l) a b) true` for some unknown universe level `l`. It turns out to be very convenient to state type equality without specifying the universe levels; with this, we survived without a universe level synthesizer as the one in Nuprl, which was created to alleviate the burden of guessing universe levels.

3.3.3 Subtyping

A *subtype* judgment

```
a <= b type
```

states that `a` is a subtype of `b`. More precisely, the partial equivalence relation associated with `a` is a subrelation of the one associated with `b`. The realizer must be `ax`. There is no support of kind markers because the subtyping relation never takes additional structures into consideration.

This is currently used whenever we only need a subtyping relationship rather than type equality. For example, if a function `f` is in type `(-> a b)`, the rule to determine whether the function application `($ f x)` is in type `b'` will only demand `b <= b' type` rather than `b = b' type`. That said, the only non-trivial subtyping relation one can prove in Red**PRL** now is the cumulativity of universes. One instance would be

```
(U 0 discrete) <= (U 1 kan)
```

realized by `ax`.

3.3.4 Subkinding

The following are *subkind* judgments:

```
a <= discrete universe
a <= kan universe
a <= coe universe
a <= hcom universe
a <= pre universe
```

They assert that `a` is a subuniverse of the universe of the specified kind at the omega level. Intuitively, `a <= k universe` would be the *subtyping judgment* `a <= (U omega k) type` if we could internalize universes at the omega level. The realizer must be `ax`. These judgments play the same role as *subtyping judgments* except that they handle the cases where the right hand side is some omega-level universe. Suppose a function `f` is in type `(-> a b)`. The rule to determine whether the function application `($ f x)` is a type will demand `b <= pre universe` rather than `b = (U omega) type` (or `b = (U l) type` for some universe level `l`).

3.3.5 Term

A *term* judgment is displayed in the sort of the expression it is asking for, for example:

```
dim
exp
```

The realizer is the received term from the user. This is used to obtain motives or dimension expressions. For example, the `rewrite` tactic requires users to specify the parts to be rewritten by fulfilling *term* subgoals.

3.4 Multiverses

Todo: To Infinity… and Beyond!

3.5 Refinement rules

3.5.1 Booleans

bool/eqtype

```
H >> bool = bool in (U #l #k)
```

bool/eq/tt

```
H >> tt = tt in bool
```

bool/eq/ff

```
H >> ff = ff in bool
```

bool/eq/if

```
H >> (if [x] (#c0 x) #m0 #t0 #f0) = (if [x] (#c1 x) #m1 #t1 #f1) in #ty
where H >> #m0 = #m1 synth ~> bool, psi
| H >> #t0 = #t1 in (#c0 tt)
| H >> #f0 = #f1 in (#c0 ff)
| H, x:bool >> (#c0 x) = (#c1 x) type
| psi
| H >> (#c0 #m0) <= #ty type
```

3.5.2 Natural numbers and integers

nat/eqtype

```
H >> nat = nat in (U #l #k)
```

nat/eq/zero

```
H >> (nat 0) = (nat 0) in nat
```

nat/eq/succ

```
H >> (succ #n) = (succ #m) in nat
| H >> #n = #m in nat
```

nat/eq/nat-rec

```
H >> (nat-rec [x] (#c0 x) #m0 #n0 [a b] (#p0 a b)) = (nat-rec [x] (#c1 x) #m1 #n1 [a␣
↪b] (#p1 a b)) in #ty
| H >> #m0 = #m1 in nat
| H >> #n0 = #n1 in (#c0 (nat 0))
| H, a:nat, b:(#c0 a) >> #p0 a b = #p1 a b in (#c0 (succ a))
| H, x:nat >> (#c0 x) = (#c1 x) type
| H >> (#c0 #m0) <= #ty type
```

int/eqtype

```
H >> int = int in (U #l #k)
```

int/eq/pos

```
H >> (pos #m) = (pos #n) in int
| H >> #m = #n in nat
```

int/eq/negsucc

```
H >> (negsucc #m) = (negsucc #n) in int
| H >> #m = #n in nat
```

int/eq/int-rec

```
H >> (int-rec [x] (#e0 x) #m0 [a] (#n0 a) [b] (#p0 b)) = (int-rec [x] (#e1 x) #m1 [a]␣
↪(#n1 a) [b] (#p1 b)) in #ty
| H >> #m0 = #m1 in int
| H, b:nat >> (#p0 b) = (#p1 b) in #e0 (pos b)
| H, a:nat >> (#n0 a) = (#n1 a) in #e0 (negsucc a)
| H, x:int >> (#e0 x) = (#e1 x) type
| H >> (#e0 m0) <= #ty type
```

3.5.3 Void

void/eqtype

```
H >> void = void in (U #l #k)
```

3.5.4 Circle

s1/eqtype

```
H >> S1 = S1 in (U #l #k)
where kan <= #k universe
```

s1/eq/base

```
H >> base = base in S1
```

s1/eq/loop

```
H >> loop #r = loop #r in S1
```

s1/eq/fcom

```
H >> (fcom #i~>#j #cap0 [#r/0=#s/0 [k] (#t0/0 k)] ... [#r/n=#s/n [k] (#t0/n k)])
     = (fcom #i~>#j #cap1 [#r/0=#s/0 [k] (#t1/0 k)] ... [#r/n=#s/n [k] (#t1/n k)]) in␣
↪S1
| H >> #cap0 = #cap1 in S1
| H, k:dim, #r/0=#s/0 >> (#t0/0 k) = (#t1/0 k) in S1
| ...
| H, k:dim, #r/n=#s/n >> (#t0/n k) = (#t1/n k) in S1
| H, k:dim, #r/0=#s/0, #r/1=#s/1 >> (#t0/0 k) = (#t1/1 k) in S1
| H, k:dim, #r/0=#s/0, #r/2=#s/2 >> (#t0/0 k) = (#t1/2 k) in S1
| ...
| H, k:dim, #r/n-1=#s/n-1, #r/n=#s/n >> (#t0/n-1 k) = (#t1/n k) in S1
| H, #r/0=#s/0 >> #cap0 = (#t0/0 #i) in S1
| ...
| H, #r/n=#s/n >> #cap0 = (#t0/n #i) in S1
```

s1/eq/s1-rec

```
H >> (S1-rec [x] (#c0 x) #m0 #b0 [u] #l0) = (S1-rec [x] (#c1 x) #m1 #b1 [u] #l1) in
↪#ty
| H >> #m0 = #m1 in S1
| H >> #b0 = #b1 in (#c0 base)
| H, u:dim >> (#l0 u) = (#l1 u) in (#c0 (loop u))
| H >> (#l0 0) = #b0 in (#c0 base)
| H >> (#l0 1) = #b0 in (#c0 base)
| H, x:S1 >> (#c0 x) = (#c1 x) kan type
| H >> (#c0 #m0) <= #ty type
```

s1/beta/loop

```
H >> (S1-rec [x] (#c x) (loop #r) #b [u] (#l u)) = #m in #ty
| H >> (#l #r) = #m in #ty
| H, #r=0 >> #b = #m in #ty
| H, #r=1 >> #b = #m in #ty
```

3.5.5 Dependent functions

fun/eqtype

```
H >> (-> [x : #a0] (#b0 x)) = (-> [x : #a1] (#b1 x)) in (U #l #k)
where
  (#k/dom, #k/cod) <-
    (discrete, discrete) if #k == discrete
    (coe, kan) if #k == kan
    (pre, hcom) if #k == hcom
    (coe, coe) if #k == coe
    (pre, pre) if #k == pre
| H >> #a0 = #a1 in (U #l #k/dom)
| H, x:#a0 >> (#b0 x) = (#b1 x) in (U #l #k/cod)
```

fun/eq/lam

```
H >> (lam [x] (#e0 x)) = (lam [x] (#e1 x)) in (-> [x : #a] (#b x))
| H, x:#a >> (#e0 x) = (#e1 x) in (#b x)
| H >> #a type
```

fun/intro

```
H >> (-> [x : #a] (#b x)) ext (lam [x] (#e x))
| H, x:#a >> (#b x) ext (#e x)
| H >> #a type
```

fun/eq/eta

```
H >> #e = #f in (-> [x : #a] (#b x))
| H >> (lam [x] ($ #e x)) = #f in (-> [x : #a] (#b x))
| H >> #e = #e in (-> [x : #a] (#b x))
```

fun/eq/app

```
H >> ($ #f0 #e0) = ($ #f1 #e1) in #ty
where H >> #f0 = #f1 synth ~> (-> [x : #a] (#b x)), psi
| H >> #e0 = #e1 in #a
| psi
| H >> (#cod #e0) <= #ty type
```

3.5.6 Records

record/eqtype

```
H >> (record [lbl/a : #a0] ... [lbl/b : (#b0 lbl/a ...)])
     = (record [lbl/a : #a1] ... [lbl/b : (#b1 lbl/a ...)])
     in (U #l #k)
where
  (#k/hd, #kltl) <-
    (discrete, discrete) if #k == discrete
    (kan, kan) if #k == kan
    (hcom, kan) if #k == hcom
    (coe, coe) if #k == coe
    (pre, pre) if #k == pre
| H >> #a0 = #a1 in (U #l #k/hd)
| ...
| H, x : #a0, ... >> (#b0 x ...) = (#b1 x ...) in (U #l #k/tl)
```

Todo: The choice of kinds `#k/hd` and `#k/tl` looks a little fishy; is this exactly what would be generated if a record were encoded as an iterated sigma type?

record/eq/tuple

```
H >> (tuple [lbl/a #p0] ... [lbl/b #q0])
     = (tuple [lbl/a #p1] ... [lbl/b #q1])
     in (record [lbl/a : #a] ... [lbl/b : (#b lbl/a ...)])
| H >> #p0 = #p1 in #a
| ...
| H >> #q0 = #q1 in (#b #p0 ...)
| ...
| H, x:#a, ... >> (#b x ...) type
```

record/eq/eta

```
H >> #e0 = #e1 in (record [lbl/a : #a] ... [lbl/b : (#b lbl/a ...)])
| H >> (tuple [lbl/a (! lbl/a #e0)] ... [lbl/b (! lbl/b #e0)])
|       = #e1 in (record [lbl/a : #a] ... [lbl/b : (#b lbl/a ...)])
| H >> #e0 in (record [lbl/a : #a] ... [lbl/b : (#b lbl/a ...)])
```

record/eq/proj

```
H >> (! lbl #e0) = (! lbl #e1) in #ty
where H >> #e0 = #e1 synth ~> (record [lbl0 : #a0] ... [lbl : (#a ...)] ...), psi
| psi
| H >> (#a (! lbl0  #e0) ...) <= #ty type
```

record/intro

```
H >> (record [lbl/a : #a] ... [lbl/b : (#b lbl/a ...)])
     ext (tuple [lbl/a #p/a] ... [lbl/b #p/b])
| H >> #a ext #p/a
| ...
```

(continues on next page)

(continued from previous page)

```
| H >> (#b #p/a ...) ext #p/b
| ...
| H, x:#a, ... >> (#b x ...) type
```

3.5.7 Paths

path/eqtype

```
H >> (path [u] (#a0 u) #m0 #n0) = (path [u] (#a1 u) #m1 #n1) in (U #l #k)
where
  #ka <-
    discrete if #k == discrete
    kan if #k == kan
    hcom if #k == hcom
    kan if #k == coe
    pre if #k == pre
| H, u:dim >> (#a0 u) = (#a1 u) in (U #l #ka)
| H >> #m0 = #m1 in (#a0 0)
| H >> #n0 = #n1 in (#a0 1)
```

path/eq/abs

```
H >> (abs [v] (#m0 v)) = (abs [v] (#m1 v)) in (path [v] (#a v) #p0 #p1)
| H, v:dim >> #m0 v = #m1 v in (#a v)
| H >> (#m0 0) = #p0 in (#a 0)
| H >> (#m0 1) = #p1 in (#a 1)
```

path/intro

```
H >> (path [u] (#a u) #p0 #p1) ext (abs [u] (#m u))
| H, u:dim >> (#a u) ext (#m u)
| H >> (#m 0) = #p0 in (#a 0)
| H >> (#m 1) = #p1 in (#a 1)
```

path/eq/eta

```
H >> #m = #n in (path [u] (#a u) #p0 #p1)
| H >> (abs [u] (#m u)) = #n in (path [u] (#a u) #p0 #p1)
| H >> #m = #m in (path [u] (#a u) #p0 #p1)
```

path/eq/app

```
H >> (@ #m0 #r) = (@ #m1 #r) in #ty
where H >> #m0 = #m1 synth ~> (path [u] (#a u) #p0 #p1), psi
| psi
| H >> (#a #r) = #ty type
```

path/eq/app/const

```
H >> (@ #m #r) = #p in #a
where
  H >> #m = #m synth ~> (path [x] (#ty x) #p0 #p1), psi
  #pr <-
    #p0 if #r == 0
    #p1 if #r == 1
| H >> #pr = #p in #a
| psi
| H >> #ty #r <= #a type
```

path/eq/from-line

```
H >> #m0 = #m1 in (path [x] (#ty x) #n0 #n1)
| H >> #m0 = #m1 in (line [x] (#ty x))
| H >> #n0 = (@ #m0 0) in (#ty 0)
| H >> #n1 = (@ #m1 1) in (#ty 1)
```

3.5.8 Lines

line/eqtype

```
H >> (line [u] (#a0 u)) = (line [u] (#a1 u)) in (U #l #k)
where
  #ka <-
    discrete if #k == discrete
    kan if #k == kan
    hcom if #k == hcom
    kan if #k == coe
    pre if #k == pre
| H, u:dim >> (#a0 u) = (#a1 u) in (U #l #ka)
```

line/eq/abs

```
H >> (abs [u] (#m0 u)) = (abs [u] (#m1 u)) in (line [u] (#a u))
| H, u:dim >> #m0 u = #m1 u in (#a u)
```

line/intro

```
H >> (line [u] (#a u)) ext (abs [u] (#m u))
| H, u:dim >> (#a u) ext (#m u)
```

line/eq/eta

```
H >> #m = #n in (line [u] (#a u))
| H >> #m in (line [u] (#a u))
| H >> (abs [u] (@ m u)) = #n in (line [u] (#a u))
```

line/eq/app

```
H >> (@ #m0 #r) = (@ #m0 #r) in #ty
where H >> #m0 = #m1 synth ~> (line [u] (#a u)), psi
| psi
| H >> (#a #r) <= #ty type
```

3.5.9 Pushouts

pushout/eqtype

```
H >> (pushout #a0 #b0 #c0 [x] (#f0 x) [x] (#g0 x)) = (pushout #a1 #b1 #c1 [x] (#f1 x)␣
↪[x] (#g1 x)) in (U #l #k)
where
  (#k/end, #k/apex) <-
    (coe, coe) if #k == kan
    (coe, coe) if #k == coe
    (pre, pre) if #k == hcom
    (pre, pre) if #k == pre
| H, x:#c0 >> (#f0 x) = (#f1 x) in #a0
| H, x:#c0 >> (#g0 x) = (#g1 x) in #b0
| H >> #a0 = #a1 in (U #l #k/end)
| H >> #b0 = #b1 in (U #l #k/end)
| H >> #c0 = #c1 in (U #l #k/apex)
```

pushout/eq/left

```
H >> (left #m0) = (left #m1) in (pushout #a #b #c [x] (#f x) [x] (#g x))
| H >> #m0 = #m1 in #a
| H, x:#c >> (#f x) in #a
| H, x:#c >> (#g x) in #b
| H >> #b type
| H >> #c type
```

pushout/eq/right

```
H >> (right #m0) = (right #m1) in (pushout #a #b #c [x] (#f x) [x] (#g x))
| H >> #m0 = #m1 in #b
| H, x:#c >> (#f x) in #a
| H, x:#c >> (#g x) in #b
| H >> #a type
| H >> #c type
```

pushout/eq/glue

```
H >> (glue #r #m0 #fm0 #gm0) = (glue #r #m1 #fm1 #gm1) in (pushout #a #b #c [x] (#f␣
↪x) [x] (#g x))
| H >> #m0 = #m1 in #c
| H >> #fm0 = #fm1 in #a
```

(continues on next page)

(continued from previous page)

```
| H >> #gm0 = #gm1 in #b
| H >> (#f #m0) = #fm0 in #a
| H >> (#g #m0) = #gm0 in #b
| H, x:#c >> (#f x) in #a
| H, x:#c >> (#g x) in #b
```

pushout/eq/fcom

```
H >> (fcom #i~>#j #cap0 [#r/0=#s/0 [k] (#t0/0 k)] ... [#r/n=#s/n [k] (#t0/n k)])
     = (fcom #i~>#j #cap1 [#r/0=#s/0 [k] (#t1/0 k)] ... [#r/n=#s/n [k] (#t1/n k)])
     in (pushout #a #b #c [x] (#f x) [x] (#g x))
where
  #ty <- (pushout #a #b #c [x] (#f x) [x] (#g x))
| H, x:#c >> (#f x) in #a
| H, x:#c >> (#g x) in #b
| H >> #a type
| H >> #b type
| H >> #c type
| H >> #cap0 = #cap1 in #ty
| H, k:dim, #r/0=#s/0 >> (#t0/0 k) = (#t1/0 k) in #ty
| ...
| H, k:dim, #r/n=#s/n >> (#t0/n k) = (#t1/n k) in #ty
| H, k:dim, #r/0=#s/0, #r/1=#s/1 >> (#t0/0 k) = (#t1/1 k) in #ty
| H, k:dim, #r/0=#s/0, #r/2=#s/2 >> (#t0/0 k) = (#t1/2 k) in #ty
| ...
| H, k:dim, #r/n-1=#s/n-1, #r/n=#s/n >> (#t0/n-1 k) = (#t1/n k) in #ty
| H, #r/0=#s/0 >> #cap0 = (#t0/0 #i) in #ty
| ...
| H, #r/n=#s/n >> #cap0 = (#t0/n #i) in #ty
```

pushout/eq/pushout-rec

```
H >> (pushout-rec [p] (#d0 p) #m0 [a] (#n0 a) [b] (#p0 b) [v x] (#q0 v x))
     = (pushout-rec [x] (#d1 x) #m1 [a] (#n1 a) [b] (#p1 b) [v x] (#q1 v x)) in #ty
where H >> #m0 = #m1 synth ~> (pushout #a #b #c [x] (#f x) [x] (#g x)), psi
| H, a:#a >> (#n0 a) = (#n1 a) in (#d0 (left a))
| H, b:#b >> (#p0 b) = (#p1 b) in (#d1 (right b))
| H, v:dim, x:#c >> (#q0 v x) = (#q1 v x) in (#d0 (glue v x (#f x) (#g x)))
| H, x:#c >> (#q0 0 x) = (#n0 (#f x)) in (#d0 (left (#f x)))
| H, x:#c >> (#q0 1 x) = (#p0 (#g x)) in (#d0 (right (#g x)))
| H, p:(pushout #a #b #c [x] (#f x) [x] (#g x)) >> (#d0 p) = (#d1 p) kan type
| psi
| H >> (#d0 #m0) <= #ty type
```

pushout/beta/glue

```
H >> (pushout-rec [p] (#d p) (glue #r #t #ft #gt) [a] (#n a) [b] (#p b) [v x] (#q v␣
↪x)) = #s in #ty
| H >> (#q #r #t) = #s in #ty
| H, #r=0 >> (#n #ft) = #s in #ty
| H, #r=1 >> (#p #gt) = #s in #ty
```

3.5.10 Coequalizers

coeq/eqtype

```
H >> (coeq #a0 #b0 [x] (#f0 x) [x] (#g0 x)) = (coeq #a1 #b1 [x] (#f1 x) [x] (#g1 x))␣
↪in (U #l #k)
where
  (#k/cod, #k/dom) <-
    (coe, coe) if #k == kan
    (coe, coe) if #k == coe
    (pre, pre) if #k == hcom
    (pre, pre) if #k == pre
| H, x:#a0 >> (#f0 x) = (#f1 x) in #b0
| H, x:#a0 >> (#g0 x) = (#g1 x) in #b0
| H >> #a0 = #a1 in (U #l #k/dom)
| H >> #b0 = #b1 in (U #l #k/cod)
```

coeq/eq/cod

```
H >> (cecod #m0) = (cecod #m1) in (coeq #a #b [x] (#f x) [x] (#g x))
| H >> #m0 = #m1 in #b
| H, x:#a >> (#f x) in #b
| H, x:#a >> (#g x) in #b
| H >> #a type
```

coeq/eq/dom

```
H >> (cedom #r #m0 #fm0 #gm0) = (cedom #r #m0 #fm0 #gm0) in (coeq #a #b [x] (#f x)␣
↪[x] (#g x))
| H >> #m0 = #m1 in #a
| H >> #fm0 = #fm1 in #b
| H >> #gm0 = #gm1 in #b
| H >> (#f #m0) = #fm0 in #b
| H >> (#g #m0) = #gm0 in #b
| H, x:#a >> (#f x) in #b
| H, x:#a >> (#g x) in #b
```

coeq/eq/fcom

```
H >> (fcom #i~>#j #cap0 [#r/0=#s/0 [k] (#t0/0 k)] ... [#r/n=#s/n [k] (#t0/n k)])
     = (fcom #i~>#j #cap1 [#r/0=#s/0 [k] (#t1/0 k)] ... [#r/n=#s/n [k] (#t1/n k)])
     in (coeq #a #b [x] (#f x) [x] [x] (#g x))
where
  #ty <- (coeq #a #b [x] (#f x) [x] [x] (#g x))
| H, x:#a >> (#f x) in #b
| H, x:#a >> (#g x) in #b
| H >> #a type
| H >> #b type
| H >> #cap0 = #cap1 in #ty
| H, k:dim, #r/0=#s/0 >> (#t0/0 k) = (#t1/0 k) in #ty
| ...
```

(continues on next page)

(continued from previous page)

```
| H, k:dim, #r/n=#s/n >> (#t0/n k) = (#t1/n k) in #ty
| H, k:dim, #r/0=#s/0, #r/1=#s/1 >> (#t0/0 k) = (#t1/1 k) in #ty
| H, k:dim, #r/0=#s/0, #r/2=#s/2 >> (#t0/0 k) = (#t1/2 k) in #ty
| ...
| H, k:dim, #r/n-1=#s/n-1, #r/n=#s/n >> (#t0/n-1 k) = (#t1/n k) in #ty
| H, #r/0=#s/0 >> #cap0 = (#t0/0 #i) in #ty
| ...
| H, #r/n=#s/n >> #cap0 = (#t0/n #i) in #ty
```

coeq/beta/dom

```
H >> (coeq-rec [c] (#p c) (cedom #r #t #ft #gt) [b] (#n b) [v a] (#q v a)) = #s in #ty
| H >> (#q #r #t) = #s in #ty
| H, #r=0 >> (#n #ft) = #s in #ty
| H, #r=1 >> (#n #gt) = #s in #ty
```

coeq/eq/coeq-rec

```
H >> (coeq-rec [c] (#p0 c) #m0 [b] (#n0 b) [v a] (#q0 v a))
     = (coeq-rec [c] (#p1 c) #m1 [b] (#n1 b) [v a] (#q1 v a)) in #ty
where H >> #m0 = #m1 synth (coeq #a #b [x] (#f x) [x] (#g x)), psi
| H, b:#b >> (#n0 b) = (#n1 b) in (#p0 (cecod b))
| H, v:dim, a:#a >> (#q0 v a) = (#q1 v a) in (#p0 (cedom v a (#f a) (#g a))
| H, a:#a >> (#q0 0 a) = (#n0 (#f a)) in (#p0 (cecod (#f a)))
| H, a:#a >> (#q0 1 a) = (#n0 (#g a)) in (#p0 (cecod (#g a)))
| H, c:(coeq #a #b [x] (#f x) [x] (#g x)) >> (#p0 c) = (#p1 c) kan type
| psi
| H >> (#p0 #m0) <= #ty type
```

3.5.11 Exact equalities

eq/eqtype

```
H >> (= #a0 #m0 #n0) = (= #a1 #m1 #n1) in (U #l #k)
where
  #ka <-
    discrete if #k == discrete
    discrete if #k == kan
    pre if #k == hcom
    discrete if #k == coe
    pre if #k == pre
| H >> #m0 = #m1 in #a0
| H >> #n0 = #n1 in #a0
| H >> #a0 = #a1 in (U #l #ka)
```

eq/eq/ax

```
H >> ax = ax in (= #a #m #n)
| H >> #m = #n in #a
```

eq/eta

```
H >> #x = #y in (= #a #m #n)
| H >> ax = #y in (= #a #m #n)
| H >> #x in (= #a #m #n)
```

3.5.12 Composite types

fcom/eqtype

```
H >> (fcom #i~>#j #Cap0 [#r/0=#s/0 [k] (#T0/0 k)] ... [#r/n=#s/n [k] (#T0/n k)])
     = (fcom #i~>#j #Cap1 [#r/0=#s/0 [k] (#T1/0 k)] ... [#r/n=#s/n [k] (#T1/n k)])
     in (U #l #k)
where
  (#k/cap, #k/tube) <-
    (kan, kan) if #k == kan
    (hcom, kan) if #k == hcom
    (kan, kan) if #k == coe
    (pre, coe) if #k == pre
| H >> #Cap0 = #Cap1 in (U #l #k/cap)
| H, k:dim, #r/0=#s/0 >> (#T0/0 k) = (#T1/0 k) in (U #l #k/tube)
| ...
| H, k:dim, #r/n=#s/n >> (#T0/n k) = (#T1/n k) in (U #l #k/tube)
| H, k:dim, #r/0=#s/0, #r/1=#s/1 >> (#T0/0 k) = (#T1/1 k) in (U #l #k/tube)
| H, k:dim, #r/0=#s/0, #r/2=#s/2 >> (#T0/0 k) = (#T1/2 k) in (U #l #k/tube)
| ...
| H, k:dim, #r/n-1=#s/n-1, #r/n=#s/n >> (#T0/n-1 k) = (#T1/n k) in (U #l #k/tube)
| H, #r/0=#s/0 >> #Cap0 = (#T0/0 #i) in (U #l #k/cap)
| ...
| H, #r/n=#s/n >> #Cap0 = (#T0/n #i) in (U #l #k/cap)
```

fcom/eq/box

```
H >> (box #i~>#j #cap0 [#r/0=#s/0 #b0/0] ... [#r/n=#s/n #b0/n])
     = (box #i~>#j #cap1 [#r/0=#s/0 #b1/0] ... [#r/n=#s/n #b1/n])
     in (fcom #i~>#j #Cap [#r/0=#s/0 [k] (#T/0 k)] ... [#r/n=#s/n [k] (#T/n k)])
| H >> #cap0 = #cap1 in #Cap
| H, #r/0=#s/0 >> #b0/0 = #b1/0 in (#T/0 #j)
| ...
| H, #r/n=#s/n >> #b0/n = #b1/n in (#T/n #j)
| H, #r/0=#s/0, #r/1=#s/1 >> #b0/0 = #b1/1 in (#T/0 #j)
| H, #r/0=#s/0, #r/2=#s/2 >> #b0/0 = #b1/2 in (#T/0 #j)
| ...
| H, #r/n-1=#s/n-1, #r/n=#s/n >> #b0/n-1 = #b1/n in (#T/n-1 #j)
| H, #r/0=#s/0 >> #cap0 = (coe #j~>#i #T/0 #b0/0) in #Cap
| ...
| H, #r/n=#s/n >> #cap0 = (coe #j~>#i #T/n #b0/n) in #Cap
| H, k:dim, #r/0=#s/0 >> (#T/0 k) coe type
| ...
| H, k:dim, #r/n=#s/n >> (#T/n k) coe type
| H, k:dim, #r/0=#s/0, #r/1=#s/1 >> (#T/0 k) = (#T/1 k) coe type
| H, k:dim, #r/0=#s/0, #r/2=#s/2 >> (#T/0 k) = (#T/2 k) coe type
| ...
```

(continues on next page)

(continued from previous page)

```
| H, k:dim, #r/n-1=#s/n-1, #r/n=#s/n >> (#T/n-1 k) = (#T/n k) coe type
| H, #r/0=#s/0 >> #Cap = (#T/0 #i) type
| ...
| H, #r/n=#s/n >> #Cap = (#T/n #i) type
```

fcom/intro

```
H >> (fcom #i~>#j #Cap [#r/0=#s/0 [k] (#T/0 k)] ... [#r/n=#s/n [k] (#T/n k)])
     ext (box #i~>#j #cap [#r/0=#s/0 #b/0] ... [#r/n=#s/n #b/n])
| H >> #Cap ext #cap
| H, #r/0=#s/0 >> (#T/0 #j) ext #b/0
| ...
| H, #r/n=#s/n >> (#T/n #j) ext #b/n
| H, #r/0=#s/0, #r/1=#s/1 >> #b/0 = #b/1 in (#T/0 #j)
| H, #r/0=#s/0, #r/2=#s/2 >> #b/0 = #b/2 in (#T/0 #j)
| ...
| H, #r/n-1=#s/n-1, #r/n=#s/n >> #b/n-1 = #b/n in (#T/n-1 #j)
| H, #r/0=#s/0 >> #cap = (coe #j~>#i #T/0 #b/0) in #Cap
| ...
| H, #r/n=#s/n >> #cap = (coe #j~>#i #T/n #b/n) in #Cap
| H, k:dim, #r/0=#s/0 >> (#T/0 k) coe type
| ...
| H, k:dim, #r/n=#s/n >> (#T/n k) coe type
| H, k:dim, #r/0=#s/0, #r/1=#s/1 >> (#T/0 k) = (#T/1 k) coe type
| H, k:dim, #r/0=#s/0, #r/2=#s/2 >> (#T/0 k) = (#T/2 k) coe type
| ...
| H, k:dim, #r/n-1=#s/n-1, #r/n=#s/n >> (#T/n-1 k) = (#T/n k) coe type
| H, #r/0=#s/0 >> #Cap = (#T/0 #i) type
| ...
| H, #r/n=#s/n >> #Cap = (#T/n #i) type
```

3.5.13 V types

v/eqtype

```
H >> (V #r #a0 #b0 #e0) = (V #r #a1 #b1 #e1) in (U #l #k)
where
  (#ka, #kb) <-
    (kan, kan) if #k == kan
    (hcom, hcom) if #k == hcom
    (coe, com) if #k == coe
    (pre, pre) if #k == pre
| H, #r=0 >> #e0 = #e1 in (Equiv #a0 #b0)
| H, #r=0 >> #a0 = #a1 in (U #l #ka)
| H >> #b0 = #b1 in (U #l #kb)
```

where `Equiv` is defined by

```
define HasAllPathsTo (#C,#c) = (-> [center : #C] (path [_] #C center #c)).
define IsContr (#C) = (* [c : #C] (HasAllPathsTo #C c)).
define Fiber (#A,#B,#f,#b) = (* [a : #A] (path [_] #B ($ #f a) #b)).
define IsEquiv (#A,#B,#f) = (-> [b : #B] (IsContr (Fiber #A #B #f b))).
define Equiv (#A,#B) = (* [f : (-> #A #B)] (IsEquiv #A #B f)).
```

v/eq/vin

```
H >> (vin #r #m0 #n0) = (vin #r #m1 #n1) in (V #r #a #b #e)
| H, #r=0 >> #m0 = #m1 in #a
| H >> #n0 = #n1 in #b
| H, #r=0 >> ($ (! proj1 #e) #m0) = #n0 in #b
| H, #r=0 >> #e in (Equiv #a #b)
```

v/intro

```
H >> (V #r #a #b #e) ext (vin #r #m #n)
| H, #r=0 >> #a ext #m
| H >> #b ext #n
| H, #r=0 >> ($ (! proj1 #e) #m) = #n in #b
| H, #r=0 >> #e in (Equiv #a #b)
```

v/eq/proj

```
H >> (vproj #r #m0 #f0) = (vproj #r #m1 #f1) in #ty
where
  #r /= 0 and #r /= 1
  H >> #m0 = #m1 synth ~> (v #r #a #b #e), psi
| H, #r=0 >> #f0 = #f1 in (-> #a #b)
| H, #r=0 >> #f0 = (! proj1 #e) in (-> #a #b)
| psi
| H >> #b <= #ty type
```

3.5.14 Kan operations

hcom/eq

```
H >> (hcom #i~>#j #ty0 #cap0 [#r/0=#s/0 [k] (#t0/0 k)] ... [#r/n=#s/n [k] (#t0/n k)])
     = (hcom #i~>#j #ty1 #cap1 [#r/0=#s/0 [k] (#t1/0 k)] ... [#r/n=#s/n [k] (#t1/n␣
↪k)]) in #ty
| H >> #cap0 = #cap1 in #ty0
| H, k:dim, #r/0=#s/0 >> (#t0/0 k) = (#t1/0 k) in #ty0
| ...
| H, k:dim, #r/n=#s/n >> (#t0/n k) = (#t1/n k) in #ty0
| H, k:dim, #r/0=#s/0, #r/1=#s/1 >> (#t0/0 k) = (#t1/1 k) in #ty0
| H, k:dim, #r/0=#s/0, #r/2=#s/2 >> (#t0/0 k) = (#t1/2 k) in #ty0
| ...
| H, k:dim, #r/n-1=#s/n-1, #r/n=#s/n >> (#t0/n-1 k) = (#t1/n k) in #ty0
| H, #r/0=#s/0 >> #cap0 = (#t0/0 #i) in #ty0
| ...
| H, #r/n=#s/n >> #cap0 = (#t0/n #i) in #ty0
| H >> #ty0 = #ty1 hcom type
| H >> #ty0 <= #ty type
```

hcom/eq/cap

```
H >> (hcom #i~>#i #ty' #cap [#r/0=#s/0 [k] (#t/0 k)] ... [#r/n=#s/n [k] (#t/n k)]) =
↪#m in #ty
| H >> #cap = #m in #ty
| H, k:dim, #r/0=#s/0 >> (#t/0 k) in #ty'
| ...
| H, k:dim, #r/n=#s/n >> (#t/n k) in #ty'
| H, k:dim, #r/0=#s/0, #r/1=#s/1 >> (#t0 k) = (#t1 k) in #ty'
| H, k:dim, #r/0=#s/0, #r/2=#s/2 >> (#t0 k) = (#t2 k) in #ty'
| ...
| H, k:dim, #r/n-1=#s/n-1, #r/n=#s/n >> (#t/n-1 k) = (#t/n k) in #ty'
| H, #r/0=#s/0 >> #cap = (#t/0 #i) in #ty'
| ...
| H, #r/n=#s/n >> #cap = (#t/n #i) in #ty'
| H >> #ty' hcom type
| H >> #ty' <= #ty type
```

hcom/eq/tube

```
H >> (hcom #i~>#j #ty' #cap [#r/0=#s/0 [k] (#t/0 k)] ... [#r/n=#s/n [k] (#t/n k)]) =
↪#m in #ty
where
  #r/0 /= #s/0, ..., #r/l-1 /= #s/l-1 and #r/l = #s/l
| H >> (#t/l #j) = #m in #ty'
| H, k:dim, #r/0=#s/0 >> (#t/0 k) in #ty'
| ...
| H, k:dim, #r/n=#s/n >> (#t/n k) in #ty'
| H, k:dim, #r/0=#s/0, #r/1=#s/1 >> (#t/0 k) = (#t/1 k) in #ty'
| H, k:dim, #r/0=#s/0, #r/2=#s/2 >> (#t/0 k) = (#t/2 k) in #ty'
| ...
| H, k:dim, #r/n-1=#s/n-1, #rn=#sn >> (#t/n-1 k) = (#tn k) in #ty'
| H, #r/0=#s/0 >> #cap = (#t/0 #i) in #ty'
| ...
| H, #r/n=#s/n >> #cap = (#t/n #i) in #ty'
| H >> #ty' hcom type
| H >> #ty' <= #ty type
```

coe/eq

```
H >> (coe #i~>#j [u] (#a0 u) #m0) = (coe #i~>#j [u] (#a1 u) #m1) in #ty
| H >> #m0 = #m1 in (#a0 #i)
| H, u:dim >> #a0 = #a1 coe type
| H >> (#a0 #j) <= #ty type
```

coe/eq/cap

```
H >> (coe #i~>#i [u] (#a u) #m) = #n in #ty
| H >> #m = #n in #ty
| H, u:dim >> (#a u) coe type
| H >> (#a #i) <= #ty type
```

3.5.15 Universes

subtype/eq

```
H >> #a <= #b type
| H >> #a = #b type
```

universe/eqtype

```
H >> (U #l #k) = (U #l #k) in (U #l' #k')
where
  #k/univ <-
    discrete if #k == discrete
    kan if #k == kan
    kan if #k == hcom
    coe if #k == coe
    kan if #k == pre
  #l < #l'
  #k/univ <= #k'
```

universe/subtype

```
H >> (U #l0 #k0) <= (U #l1 #k1) type
where
  #l0 <= #l1
  #k0 <= #k1
```

3.6 Indices

- genindex
- search

3.7 Acknowledgments

This research was sponsored by the Air Force Office of Scientific Research under grant number FA9550-15-1-0053 and the National Science Foundation under grant number DMS-1638352. We also thank the Isaac Newton Institute for Mathematical Sciences for its support and hospitality during the program "Big Proof" when part of this work was undertaken; the program was supported by the Engineering and Physical Sciences Research Council under grant number EP/K032208/1. The views and conclusions contained here are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, government or any other entity.

Index

S

U

V