Inductively generated formal topologies in Matita

This is a not so short introduction to Matita, based on the formalization of the paper

> Between formal topology and game theory: an explicit solution for the conditions for an inductive generation of formal topologies

by Stefano Berardi and Silvio Valentini.

The tutorial and the formalization are by Enrico Tassi.

The reader should be familiar with inductively generated formal topologies and have some basic knowledge of type theory and λ-calculus.

A considerable part of this tutorial is devoted to explain how to define notations that resemble the ones used in the original paper. We believe this is an important part of every formalization, not only from the aesthetic point of view, but also from the practical point of view. Being consistent allows to follow the paper in a pedantic way, and hopefully to make the formalization (at least the definitions and proved statements) readable to the author of the paper.

The formalization uses the "new generation" version of Matita (that will be named 1.x when finally released). Last stable release of the "old" system is named 0.5.7; the ng system is coexisting with the old one in all development release (named "nightly builds" in the download page of Matita) with a version strictly greater than 0.5.7.

To read this tutorial in HTML format, you need a decent browser equipped with a unicode capable font. Use the PDF format if some symbols are not displayed correctly.

Orienteering

The graphical interface of Matita is composed of three windows: the script window, on the left, is where you type; the sequent window on the top right is where the system shows you the ongoing proof; the error window, on the bottom right, is where the system complains. On the top of the script window five buttons drive the processing of the proof script. From left to right they request the system to:

- go back to the beginning of the script
- go back one step
- go to the current cursor position
- advance one step
- advance to the end of the script

When the system processes a command, it locks the part of the script corresponding to the command, such that you cannot edit it anymore (without going back). Locked parts are coloured in blue.

The sequent window is hyper textual, i.e. you can click on symbols to jump to their definition, or switch between different notations for the same expression (for example, equality has two notations, one of them makes the type of the arguments explicit).

Everywhere in the script you can use the `ncheck (term).` command to ask for the type a given term. If you do that in the middle of a proof, the term is assumed to live in the current proof context (i.e. can use variables introduced so far).

To ease the typing of mathematical symbols, the script window implements two unusual input facilities:

- some TeX symbols can be typed using their TeX names, and are automatically converted to UTF-8 characters. For a list of the supported TeX names, see the menu: View ▹ TeX/UTF-8 Table. Moreover some ASCII-art is understood as well, like `=>` and `->` to mean double or single arrows. Here we recall some of these "shortcuts":

    - ∀ can be typed with `\forall`
    - λ can be typed with `\lambda`
    - ≝ can be typed with `\def` or `:=`
    - → can be typed with `\to` or `->`
- some symbols have variants, like the ≤ relation and ≤, ≰, ≮. The user can cycle between variants typing one of them and then pressing ALT-L. Note that also letters do have variants, for example W has Ω, 𝕎 and **W**, L has Λ, 𝕃, and **L**, F has Φ, … Variants are listed in the aforementioned TeX/UTF-8 table.

The syntax of terms (and types) is the one of the λ-calculus CIC on which Matita is based. The main syntactical difference w.r.t. the usual mathematical notation is the function application, written `(f x y)` in place of `f(x,y)`.

Pressing `F1` opens the Matita manual.

CIC (as implemented in Matita) in a nutshell

CIC is a full and functional Pure Type System (all products do exist, and their sort is is determined by the target) with an impredicative sort Prop and a predicative sort Type. It features both dependent types and polymorphism like the Calculus of Constructions. Proofs and terms share the same syntax, and they can occur in types.

The environment used for in the typing judgement can be populated with well typed definitions or theorems, (co)inductive types validating positivity conditions and recursive functions provably total by simple syntactical analysis (recursive calls are allowed only on structurally smaller subterms). Co-recursive functions can be defined as well, and must satisfy the dual condition, i.e. performing the recursive call only after having generated a constructor (a piece of output).

The CIC λ-calculus is equipped with a pattern matching construct (match) on inductive types defined in the environment. This construct, together with the possibility to definable total recursive functions, allows to define eliminators (or constructors) for (co)inductive types.

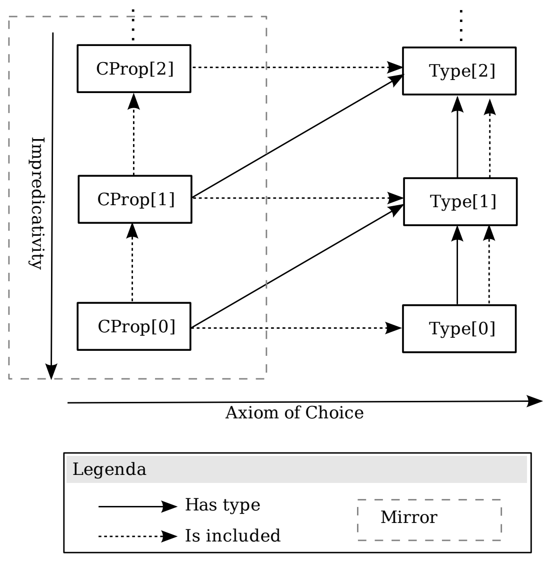
Types are compare up to conversion. Since types may depend on terms, conversion involves β-reduction, δ-reduction (definition unfolding), ζ-reduction (local definition unfolding), ι-reduction (pattern matching simplification), μ-reduction (recursive function computation) and ν-reduction (co-fixpoint computation).

Since we are going to formalize constructive and predicative mathematics in an intensional type theory like CIC, we try to establish some terminology. Type is the sort of sets equipped with the `Id` equality (i.e. an intensional, not quotiented set).

We write `Type[i]` to mention a Type in the predicative hierarchy of types. To ease the comprehension we will use `Type[0]` for sets, and `Type[1]` for classes. The index `i` is just a label: constraints among universes are declared by the user. The standard library defines

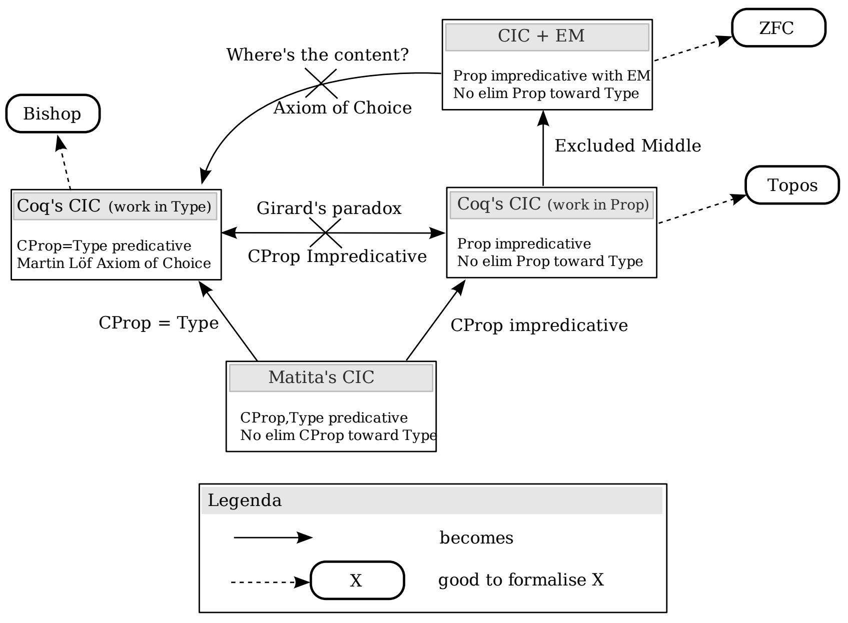
Type[0] < Type[1] < Type[2]

Matita implements a variant of CIC in which constructive and predicative proposition are distinguished from predicative data types.



For every `Type[i]` there is a corresponding level of predicative propositions `CProp[i]` (the C initial is due to historical reasons, and stands for constructive). A predicative proposition cannot be eliminated toward `Type[j]` unless it holds no computational content (i.e. it is an inductive proposition with 0 or 1 constructors with propositional arguments, like `Id` and `And` but not like `Or`).

The distinction between predicative propositions and predicative data types is a peculiarity of Matita (for example in CIC as implemented by Coq they are the same). The additional restriction of not allowing the elimination of a CProp toward a Type makes the theory of Matita minimal in the following sense:



Theorems proved in CIC as implemented in Matita can be reused in a classical and impredicative framework (i.e. forcing Matita to collapse the hierarchy of constructive propositions and assuming the excluded middle on them). Alternatively, one can decide to collapse predicative propositions and predicative data types recovering the Axiom of Choice in the sense of Martin Löf (i.e. ∃ really holds a witness and can be eliminated to inhabit a type).

This implementation of CIC is the result of the collaboration with Maietti M., Sambin G. and Valentini S. of the University of Padua.

Formalization choices

There are many different ways of formalizing the same piece of mathematics in CIC, depending on what our interests are. There is usually a trade-off between the possibility of reuse the formalization we did and its complexity.

In this work, our decisions mainly regarded the following two areas

- Axiom of Choice: controlled use or not
- Equality: Id or not

Axiom of Choice

In this paper it is clear that the author is interested in using the Axiom of Choice, thus choosing to identify ∃ and Σ (i.e. working in the leftmost box of the graph "Coq's CIC (work in CProp)") would be a safe decision (that is, the author of the paper would not complain we formalized something different from what he had in mind).

Anyway, we may benefit from the minimality of CIC as implemented in Matita, "asking" the type system to ensure we do no use the Axiom of Choice elsewhere in the proof (by mistake or as a shortcut). If we identify ∃ and Σ from the very beginning, the system will not complain if we use the Axiom of Choice at all. Moreover, the elimination of an inductive type (like ∃) is a so common operation that the syntax chosen for the elimination command is very compact and non informative, hard to spot for a human being (in fact it is just two characters long!).

We decided to formalize the whole paper without identifying CProp and Type and assuming the Axiom of Choice as a real axiom (i.e. a black hole with no computational content, a function with no body).

It is clear that this approach give us full control on when/where we really use the Axiom of Choice. But, what are we loosing? What happens to the computational content of the proofs if the Axiom of Choice gives no content back?

It really depends on when we actually look at the computational content of the proof and we "run" that program. We can extract the content and run it before or after informing the system that our propositions are actually code (i.e. identifying ∃ and Σ). If we run the program before, as soon as the computation reaches the Axiom of Choice it stops, giving no output. If we tell the system that CProp and Type are the same, we can exhibit a body for the Axiom of Choice (i.e. a projection) and the extracted code would compute an output.

Note that the computational content is there even if the Axiom of Choice is an axiom, the difference is just that we cannot use it (the typing rules inhibit the elimination of the existential). This is possible only thanks to the minimality of CIC as implemented in Matita.

Equality

What we have to decide here is which models we admit. The paper does not mention quotiented sets, thus using an intensional equality is enough to capture the intended content of the paper. Nevertheless, the formalization cannot be reused in a concrete example where the (families of) sets that will build the axiom set are quotiented.

Matita gives support for setoid rewriting under a context built with non dependent morphisms. As we will detail later, if we assume a generic equality over the carrier of our axiom set, a non trivial inductive construction over the ordinals has to be proved to respect extensionality (i.e. if the input is an extensional set, also the output is). The proof requires to rewrite under the context formed by the family of sets `I` and `D`, that have a dependent type. Declaring them as dependently typed morphisms is possible, but Matita does not provide an adequate support for them, and would thus need more effort than formalizing the whole paper.

Anyway, in a preliminary attempt of formalization, we tried the setoid approach, reaching the end of the formalization, but we had to assume the proof of the extensionality of the `U_x` construction (while there is no need to assume the same property for `F_x`!).

The current version of the formalization uses `Id`.

The standard library and the `include` command

Some basic notions, like subset, membership, intersection and union are part of the standard library of Matita.

These notions come with some standard notation attached to them:

- A ∪ B can be typed with `A \cup B`
- A ∩ B can be typed with `A \cap B`
- A ≬ B can be typed with `A \between B`
- x ∈ A can be typed with `x \in A`
- Ω^A, that is the type of the subsets of A, can be typed with `\Omega ^ A`

The `include` command tells Matita to load a part of the library, in particular the part that we will use can be loaded as follows:

```
include "sets/sets.ma".
```

Some basic results that we will use are also part of the sets library:

- subseteq_union_l: ∀A.∀U,V,W:Ω^A.U ⊆ W → V ⊆ W → U ∪ V ⊆ W
- subseteq_intersection_r: ∀A.∀U,V,W:Ω^A.W ⊆ U → W ⊆ V → W ⊆ U ∩ V

Defining Axiom set

A set of axioms is made of a set `S`, a family of sets `I` and a family `C` of subsets of `S` indexed by elements `a` of `S` and elements of `I(a)`.

It is desirable to state theorems like "for every set of axioms, ..." without explicitly mentioning S, I and C. To do that, the three components have to be grouped into a record (essentially a dependently typed tuple). The system is able to generate the projections of the record automatically, and they are named as the fields of the record. So, given an axiom set `A` we can obtain the set with `S A`, the family of sets with `I A` and the family of subsets with `C A`.

```
nrecord Ax : Type[1] ≝ {
  S :> Type[0];
  I :  S → Type[0];
  C :  ∀a:S. I a → Ω^S
}.
```

Forget for a moment the `:>` that will be detailed later, and focus on the record definition. It is made of a list of pairs: a name, followed by `:` and the its type. It is a dependently typed tuple, thus already defined names (fields) can be used in the types that follow.

Note that the field `S` was declared with `:>` instead of a simple `:`. This declares the `S` projection to be a coercion. A coercion is a "cast" function the system automatically inserts when it is needed. In that case, the projection `S` has type `Ax → setoid`, and whenever the expected type of a term is `setoid` while its type is `Ax`, the system inserts the coercion around it, to make the whole term well typed.

When formalizing an algebraic structure, declaring the carrier as a coercion is a common practice, since it allows to write statements like

```
∀G:Group.∀x:G.x * x^-1 = 1
```

The quantification over `x` of type `G` is ill-typed, since `G` is a term (of type `Group`) and thus not a type. Since the carrier projection `carr` is a coercion, that maps a `Group` into the type of its elements, the system automatically inserts `carr` around `G`, obtaining `…∀x: carr G.…`.

Coercions are hidden by the system when it displays a term. In this particular case, the coercion `S` allows to write (and read):

```
∀A:Ax.∀a:A.…
```

Since `A` is not a type, but it can be turned into a `setoid` by `S` and a `setoid` can be turned into a type by its `carr` projection, the composed coercion `carr ∘ S` is silently inserted.

Implicit arguments

Something that is not still satisfactory, is that the dependent type of `I` and `C` are abstracted over the Axiom set. To obtain the precise type of a term, you can use the `ncheck` command as follows.

```
(** ncheck I. *) (* shows: ∀A:Ax.A → Type[0] *)
(** ncheck C. *) (* shows: ∀A:Ax.∀a:A.A → I A a → Ω^A *)
```

One would like to write `I a` and not `I A a` under a context where `A` is an axiom set and `a` has type `S A` (or thanks to the coercion mechanism simply `A`). In Matita, a question mark represents an implicit argument, i.e. a missing piece of information the system is asked to infer. Matita performs Hindley-Milner-style type inference, thus writing `I ? a` is enough: since the second argument of `I` is typed by the first one, the first (omitted) argument can be inferred just computing the type of `a` (that is `A`).

```
(** ncheck (∀A:Ax.∀a:A.I ? a). *) (* shows: ∀A:Ax.∀a:A.I A a *)
```

This is still not completely satisfactory, since you have always to type `?`; to fix this minor issue we have to introduce the notational support built in Matita.

Notation for I and C

Matita is quipped with a quite complex notational support, allowing the user to define and use mathematical notations ([From Notation to Semantics: There and Back Again](#)).

Since notations are usually ambiguous (e.g. the frequent overloading of symbols) Matita distinguishes between the term level, the content level, and the presentation level, allowing multiple mappings between the content and the term level.

The mapping between the presentation level (i.e. what is typed on the keyboard and what is displayed in the sequent window) and the content level is defined with the `notation` command. When followed by `>`, it defines an input (only) notation.

```
notation > "𝐈 term 90 a" non associative with precedence 70 for @{ 'I $a }.
notation > "𝐂 term 90 a term 90 i" non associative with precedence 70 for @{ 'C $a $
```

The first notation defines the writing `𝐈 a` where `a` is a generic term of precedence 90, the maximum one. This high precedence forces parentheses around any term of a lower precedence. For example `𝐈 x` would be accepted, since identifiers have precedence 90, but `𝐈 f x` would be interpreted as `(𝐈 f) x`. In the latter case, parentheses have to be put around `f x`, thus the accepted writing would be `𝐈 (f x)`.

To obtain the `𝐈` is enough to type `I` and then cycle between its similar symbols with ALT-L. The same for `𝐂`. Notations cannot use regular letters or the round parentheses, thus their variants (like the bold ones) have to be used.

The first notation associates `I a` with `'I $a` where `'I` is a new content element to which a term `$a` is passed.

Content elements have to be interpreted, and possibly multiple, incompatible, interpretations can be defined.

```
interpretation "I" 'I a = (I ? a).
interpretation "C" 'C a i = (C ? a i).
```

The `interpretation` command allows to define the mapping between the content level and the terms level. Here we associate the `I` and `C` projections of the Axiom set record, where the Axiom set is an implicit argument `?` to be inferred by the system.

Interpretation are bi-directional, thus when displaying a term like `C _ a i`, the system looks for a presentation for the content element `'C a i`.

```
notation < "I  \sub( ❨a❩ )" non associative with precedence 70 for @{ 'I $a }.
notation < "C \sub( ❨a,\emsp i❩ )" non associative with precedence 70 for @{ 'C $a $
```

For output purposes we can define more complex notations, for example we can put bold parentheses around the arguments of **I** and **C**, decreasing the size of the arguments and lowering their baseline (i.e. putting them as subscript), separating them with a comma followed by a little space.

The first (technical) definition

Before defining the cover relation as an inductive predicate, one has to notice that the infinity rule uses, in its hypotheses, the cover relation between two subsets, while the inductive predicate we are going to define relates an element and a subset.

```
                  a ∈ U                   i ∈ I(a)    C(a,i) ◃ U
(reflexivity) ---------   (infinity) ------------------------
                  a ◃ U                        a ◃ U
```

An option would be to unfold the definition of cover between subsets, but we prefer to define the abstract notion of cover between subsets (so that we can attach a (ambiguous) notation to it).

Anyway, to ease the understanding of the definition of the cover relation between subsets, we first define the inductive predicate unfolding the definition, and we later refine it with.

```
ninductive xcover (A : Ax) (U : Ω^A) : A → CProp[0] ≝
| xcreflexivity : ∀a:A. a ∈ U → xcover A U a
| xcinfinity    : ∀a:A.∀i:I a. (∀y.y ∈ C a i → xcover A U y) → xcover A U a.
```

We defined the xcover (x will be removed in the final version of the definition) as an inductive predicate. The arity of the inductive predicate has to be carefully analyzed:

(A : Ax) (U : Ω^A) : A → CProp[0]

The syntax separates with `:` abstractions that are fixed for every constructor (introduction rule) and abstractions that can change. In that case the parameter `U` is abstracted once and for all in front of every constructor, and every occurrence of the inductive predicate is applied to `U` in a consistent way. Arguments abstracted on the right of `:` are not constant, for example the xcinfinity constructor introduces `a ◃ U`, but under the assumption that (for every y) `y ◃ U`. In that rule, the left had side of the predicate changes, thus it has to be abstracted (in the arity of the inductive predicate) on the right of `:`.

The intuition Valentini suggests is that we are defining the unary predicate "being covered by U" (i.e. `_ ◃ U`) and not "being covered" (i.e. `_ ◃ _`). Unluckily, the syntax of Matita forces us to abstract `U` first, and we will make it the second argument of the predicate using the notational support Matita offers.

```
(** ncheck xcreflexivity. *) (* shows: ∀A:Ax.∀U:Ω^A.∀a:A.a∈U → xcover A U a *)
```

We want now to abstract out `(∀y.y ∈ 𝐂 a i → xcover A U y)` and define a notion `cover_set` to which a notation `𝐂 a i ◃ U` can be attached.

This notion has to be abstracted over the cover relation (whose type is the arity of the inductive `xcover` predicate just defined).

Then it has to be abstracted over the arguments of that cover relation, i.e. the axiom set and the set `U`, and the subset (in that case `𝐂 a i`) sitting on the left hand side of `◃`.

```
ndefinition cover_set :
  ∀cover: ∀A:Ax.Ω^A → A → CProp[0]. ∀A:Ax.∀C,U:Ω^A. CProp[0]
≝
  λcover.                              λA,    C,U.     ∀y.y ∈ C → cover A U y.
```

The `ndefinition` command takes a name, a type and body (of that type). The type can be omitted, and in that case it is inferred by the system. If the type is given, the system uses it to infer implicit arguments of the body. In that case all types are left implicit in the body.

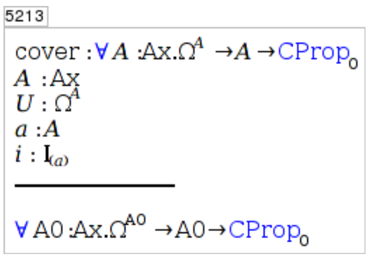
We now define the notation `a ◃ b`. Here the keywork `hvbox` and `break` tell the system how to wrap text when it does not fit the screen (they can be safely ignored for the scope of this tutorial). We also add an interpretation for that notation, where the (abstracted) cover relation is implicit. The system will not be able to infer it from the other arguments `C` and `U` and will thus prompt the user for it. This is also why we named this interpretation `covers set temp`: we will later define another interpretation in which the cover relation is the one we are going to define.

```
notation "hvbox(a break ◃ b)" non associative with precedence 45
for @{ 'covers $a $b }.

interpretation "covers set temp" 'covers C U = (cover_set ?? C U).
```

The cover relation

We can now define the cover relation using the `◃` notation for the premise of infinity.

```
ninductive cover (A : Ax) (U : Ω^A) : A → CProp[0] ≝
| creflexivity : ∀a. a ∈ U → cover A U a
| cinfinity    : ∀a. ∀i. 𝐂 a i ◃ U → cover A U a.
```


5213

cover : ∀A :Ax.Ω^A → A → CProp$_0$
A : Ax
U : Ω^A
a : A
i : $\mathbf{I}_{(a)}$

∀A0 :Ax.Ω^{A0} → A0 → CProp$_0$

```
napply cover;
nqed.
```

Note that the system accepts the definition but prompts the user for the relation the `cover_set` notion is abstracted on.

The horizontal line separates the hypotheses from the conclusion. The `napply cover` command tells the system that the relation it is looking for is exactly our first context entry (i.e. the inductive predicate we are defining, up to α-conversion); while the `nqed` command ends a definition or proof.

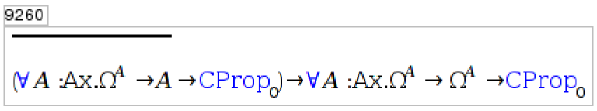
We can now define the interpretation for the cover relation between an element and a subset first, then between two subsets (but this time we fix the relation `cover_set` is abstracted on).

```
interpretation "covers" 'covers a U = (cover ? U a).
interpretation "covers set" 'covers a U = (cover_set cover ? a U).
```

We will proceed similarly for the fish relation, but before going on it is better to give a short introduction to the proof mode of Matita. We define again the `cover_set` term, but this time we build its body interactively. In the λ-calculus Matita is based on, CIC, proofs and terms share the same syntax, and it is thus possible to use the commands devoted to build proof term also to build regular definitions. A tentative semantics for the proof mode commands (called tactics) in terms of sequent calculus rules are given in the appendix.

```
ndefinition xcover_set :
  ∀c: ∀A:Ax.Ω^A → A → CProp[0]. ∀A:Ax.∀C,U:Ω^A. CProp[0].
```

9260



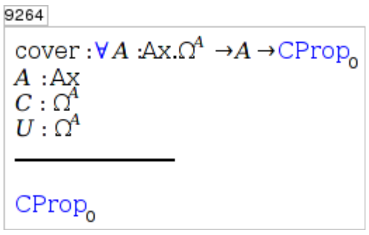
$(\forall A : Ax.\Omega^A \to A \to CProp_0) \to \forall A : Ax.\Omega^A \to \Omega^A \to CProp_0$

The system asks for a proof of the full statement, in an empty context.

The `#` command is the ∀-introduction rule, it gives a name to an assumption putting it in the context, and generates a λ-abstraction in the proof term.

```
#cover; #A; #C; #U;
```

9264



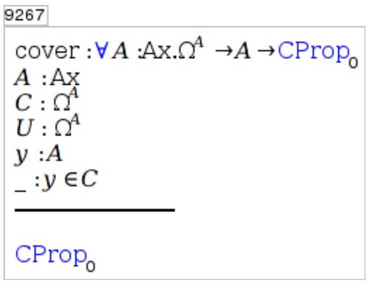
$cover : \forall A : Ax.\Omega^A \to A \to CProp_0$
$A : Ax$
$C : \Omega^A$
$U : \Omega^A$

$CProp_0$

We have now to provide a proposition, and we exhibit it. We left a part of it implicit; since the system cannot infer it it will ask for it later. Note that the type of `∀y:A.y ∈ C → ?` is a proposition whenever `?` is a proposition.

```
napply (∀y:A.y ∈ C → ?);
```

9267

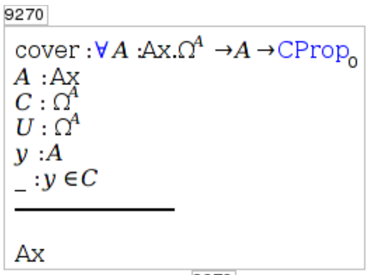


$cover : \forall A : Ax.\Omega^A \to A \to CProp_0$
$A : Ax$
$C : \Omega^A$
$U : \Omega^A$
$y : A$
$\_ : y \in C$

$CProp_0$

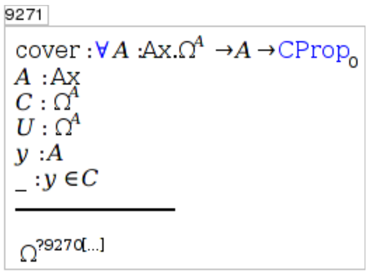
The proposition we want to provide is an application of the cover relation we have abstracted in the context. The command `napply`, if the given term has not the expected type (in that case it is a product versus a proposition) it applies it to as many implicit arguments as necessary (in that case `? ? ?`).

```
napply cover;
```


9270

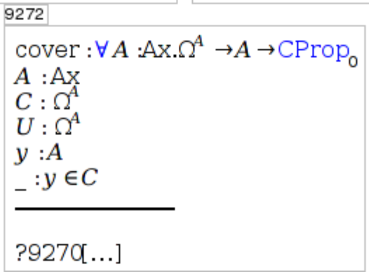
cover : $\forall A : Ax.\Omega^A \to A \to \mathrm{CProp}_0$
A : Ax
$C : \Omega^A$
$U : \Omega^A$
$y : A$
_ : $y \in C$

Ax


9271

cover : $\forall A : Ax.\Omega^A \to A \to \mathrm{CProp}_0$
A : Ax
$C : \Omega^A$
$U : \Omega^A$
$y : A$
_ : $y \in C$

$\Omega^{?9270[...]}$


9272

cover : $\forall A : Ax.\Omega^A \to A \to \mathrm{CProp}_0$
A : Ax
$C : \Omega^A$
$U : \Omega^A$
$y : A$
_ : $y \in C$

?9270[...]

The system will now ask in turn the three implicit arguments passed to cover. The syntax ##[allows to start a branching to tackle every sub proof individually, otherwise every command is applied to every subproof. The command ##| switches to the next subproof and ##] ends the branching.

```
##[ napply A;
##| napply U;
##| napply y;
##]
nqed.
```

The fish relation

The definition of fish works exactly the same way as for cover, except that it is defined as a coinductive proposition.

```
ndefinition fish_set ≝ λf:∀A:Ax.Ω^A → A → CProp[0].
 λA,U,V.
  ∃a.a ∈ V ∧ f A U a.

(* a \ltimes b *)
notation "hvbox(a break ⋉ b)" non associative with precedence 45
for @{ 'fish $a $b }.

interpretation "fish set temp" 'fish A U = (fish_set ?? U A).

ncoinductive fish (A : Ax) (F : Ω^A) : A → CProp[0] ≝
| cfish : ∀a. a ∈ F → (∀i:𝐈 a .𝐂  a i ⋉ F) → fish A F a.
napply fish;
nqed.

interpretation "fish set" 'fish A U = (fish_set fish ? U A).
interpretation "fish" 'fish a U = (fish ? U a).
```

Introduction rule for fish

Matita is able to generate elimination rules for inductive types

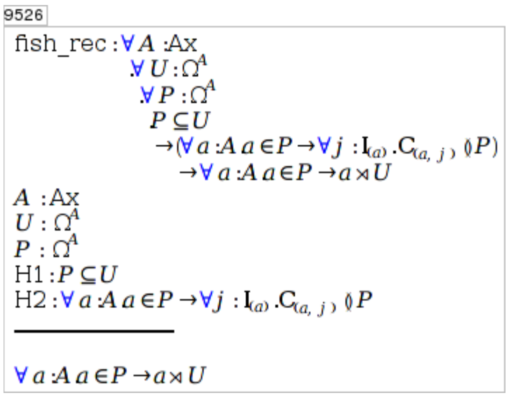
```
(** ncheck cover_rect_CProp0. *)
```

but not introduction rules for the coinductive case.

```
                   P ⊆ U   (∀x,j.x ∈ P → C(x,j) ≬ P)   a ∈ P
(fish intro) ----------------------------------------------
                                  a ⋉ U
```

We thus have to define the introduction rule for fish by co-recursion. Here we again use the proof mode of Matita to exhibit the body of the corecursive function.

```
nlet corec fish_rec (A:Ax) (U: Ω^A)
 (P: Ω^A) (H1: P ⊆ U)
  (H2: ∀a:A. a ∈ P → ∀j: 𝐈 a. 𝐂 a j ≬ P): ∀a:A. ∀p: a ∈ P. a ⋉ U ≝ ?.
```

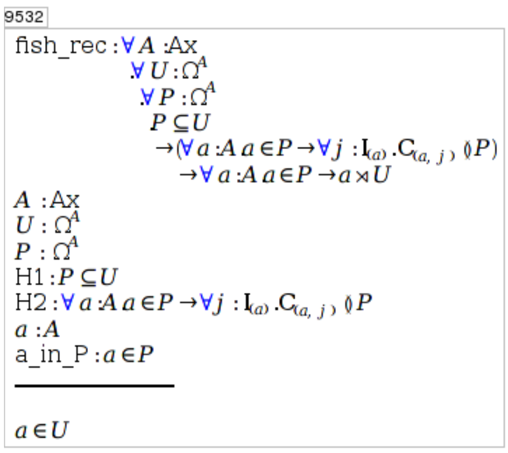


```
9526
fish_rec : ∀A :Ax
            ∀U :Ω^A
             ∀P :Ω^A
              P ⊆ U
               →(∀a :A a ∈P →∀j :𝐈(a) .𝐂(a, j) ≬P)
                 →∀a :A a ∈P →a ⋉U
A :Ax
U : Ω^A
P : Ω^A
H1 : P ⊆U
H2 : ∀a :A a ∈P →∀j :𝐈(a) .𝐂(a, j) ≬P
________________

∀a :A a ∈P →a ⋉U
```

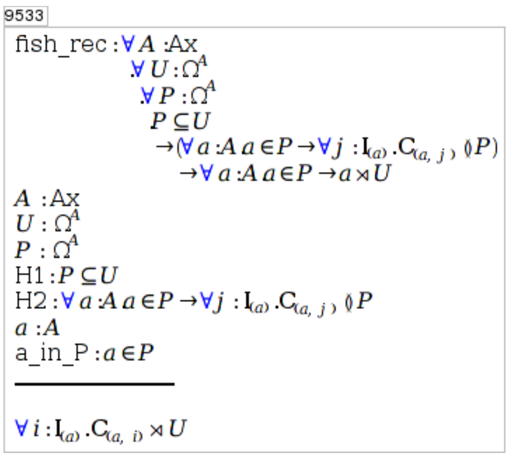
Note the first item of the context, it is the corecursive function we are defining. This item allows to perform the recursive call, but we will be allowed to do such call only after having generated a constructor of the fish coinductive type.

We introduce `a` and `p`, and then return the fish constructor `cfish`. Since the constructor accepts two arguments, the system asks for them.

```
#a; #a_in_P; napply cfish;
```



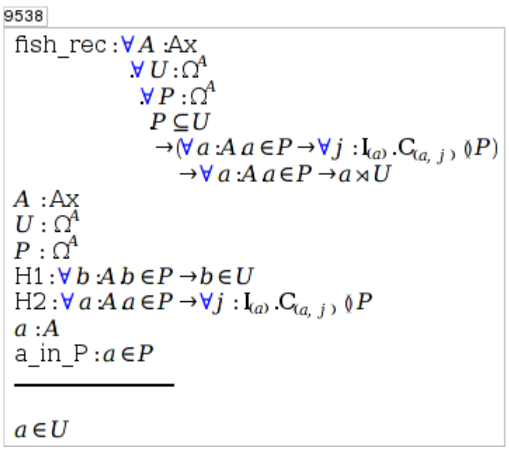
```
9532
fish_rec : ∀A :Ax
            ∀U :Ω^A
             ∀P :Ω^A
              P ⊆U
               →(∀a :A a ∈P →∀j :𝐈(a) .𝐂(a, j) ≬P)
                 →∀a :A a ∈P →a ⋉U
A :Ax
U : Ω^A
P : Ω^A
H1 : P ⊆U
H2 : ∀a :A a ∈P →∀j :𝐈(a) .𝐂(a, j) ≬P
a :A
a_in_P : a ∈P
________________

a ∈U
```



```
9533
fish_rec : ∀A :Ax
            ∀U :Ω^A
             ∀P :Ω^A
              P ⊆U
               →(∀a :A a ∈P →∀j :𝐈(a) .𝐂(a, j) ≬P)
                 →∀a :A a ∈P →a ⋉U
A :Ax
U : Ω^A
P : Ω^A
H1 : P ⊆U
H2 : ∀a :A a ∈P →∀j :𝐈(a) .𝐂(a, j) ≬P
a :A
a_in_P : a ∈P
________________

∀i :𝐈(a) .𝐂(a, i) ⋉U
```

The first one is a proof that `a ∈ U`. This can be proved using `H1` and `p`. With the `nchange` tactic we change `H1` into an equivalent form (this step can be skipped, since the system would be able to unfold the definition of inclusion by itself)

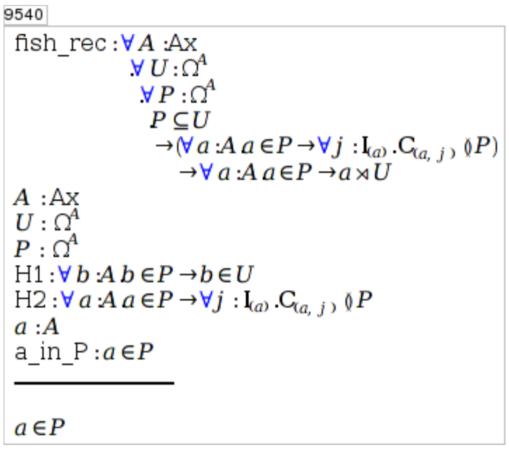
```
##[ nchange in H1 with (∀b.b∈P → b∈U);
```



```
9538
fish_rec : ∀A :Ax
            ∀U :Ω^A
             ∀P :Ω^A
              P ⊆U
               →(∀a :A a ∈P →∀j :𝐈(a) .𝐂(a, j) ≬P)
                 →∀a :A a ∈P →a ⋉U
A :Ax
U : Ω^A
P : Ω^A
H1 : ∀b :A b ∈P →b ∈U
H2 : ∀a :A a ∈P →∀j :𝐈(a) .𝐂(a, j) ≬P
a :A
a_in_P : a ∈P
________________

a ∈U
```

It is now clear that `H1` can be applied. Again `napply` adds two implicit arguments to `H1 ? ?`, obtaining a proof of `? ∈ U` given a proof that `? ∈ P`. Thanks to unification, the system understands that `?` is actually `a`, and it asks a proof that `a ∈ P`.

```
        napply H1;
```


9540

fish_rec : $\forall A:\mathrm{Ax}$
$\forall U:\Omega^A$
$\forall P:\Omega^A$
$P \subseteq U$
$\to (\forall a:A\ a \in P \to \forall j:\mathbb{I}_{\langle a\rangle}.\mathbb{C}_{\langle a,\ j\rangle} \between P)$
$\to \forall a:A\ a \in P \to a \ltimes U$

$A : \mathrm{Ax}$
$U : \Omega^A$
$P : \Omega^A$
H1 : $\forall b:A\ b \in P \to b \in U$
H2 : $\forall a:A\ a \in P \to \forall j:\mathbb{I}_{\langle a\rangle}.\mathbb{C}_{\langle a,\ j\rangle} \between P$
$a : A$
a_in_P : $a \in P$

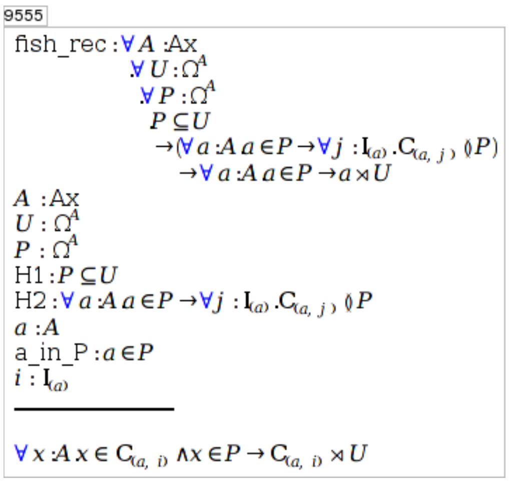
$a \in P$

The `nassumption` tactic looks for the required proof in the context, and in that cases finds it in the last context position.

We move now to the second branch of the proof, corresponding to the second argument of the `cfish` constructor.

We introduce `i` and then we destruct `H2 a p i`, that being a proof of an overlap predicate, give as an element and a proof that it is both in `C a i` and `P`.

```
        nassumption;
    ##| #i; ncases (H2 a a_in_P i);
```


9555

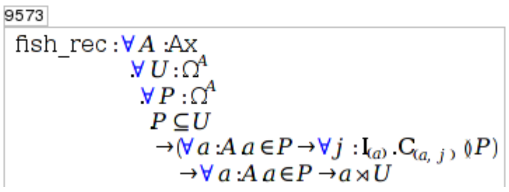
fish_rec : $\forall A:\mathrm{Ax}$
$\forall U:\Omega^A$
$\forall P:\Omega^A$
$P \subseteq U$
$\to (\forall a:A\ a \in P \to \forall j:\mathbb{I}_{\langle a\rangle}.\mathbb{C}_{\langle a,\ j\rangle} \between P)$
$\to \forall a:A\ a \in P \to a \ltimes U$

$A : \mathrm{Ax}$
$U : \Omega^A$
$P : \Omega^A$
H1 : $P \subseteq U$
H2 : $\forall a:A\ a \in P \to \forall j:\mathbb{I}_{\langle a\rangle}.\mathbb{C}_{\langle a,\ j\rangle} \between P$
$a : A$
a_in_P : $a \in P$
$i : \mathbb{I}_{\langle a\rangle}$

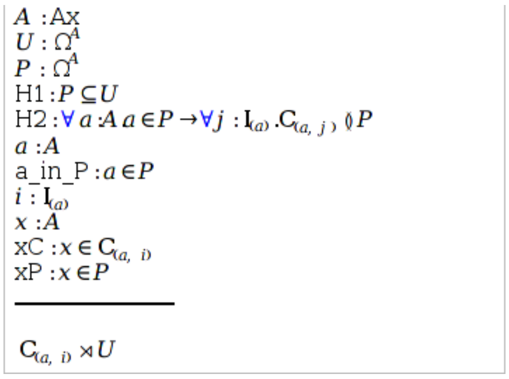
$\forall x:A\ x \in \mathbb{C}_{\langle a,\ i\rangle} \wedge x \in P \to \mathbb{C}_{\langle a,\ i\rangle} \ltimes U$

We then introduce `x`, break the conjunction (the `*;` command is the equivalent of `ncases` but operates on the first hypothesis that can be introduced). We then introduce the two sides of the conjunction.

```
        #x; *; #xC; #xP;
```


9573

fish_rec : $\forall A:\mathrm{Ax}$
$\forall U:\Omega^A$
$\forall P:\Omega^A$
$P \subseteq U$
$\to (\forall a:A\ a \in P \to \forall j:\mathbb{I}_{\langle a\rangle}.\mathbb{C}_{\langle a,\ j\rangle} \between P)$
$\to \forall a:A\ a \in P \to a \ltimes U$



$A : \mathrm{Ax}$
$U : \Omega^A$
$P : \Omega^A$
$\mathrm{H1} : P \subseteq U$
$\mathrm{H2} : \forall a{:}A.\, a \in P \to \forall j : \mathbb{I}_{\langle a\rangle}.\, \mathbb{C}_{\langle a,\ j\,\rangle} \between P$
$a : A$
$\mathrm{a\_in\_P} : a \in P$
$i : \mathbb{I}_{\langle a\rangle}$
$x : A$
$\mathrm{xC} : x \in \mathbb{C}_{\langle a,\ i\rangle}$
$\mathrm{xP} : x \in P$

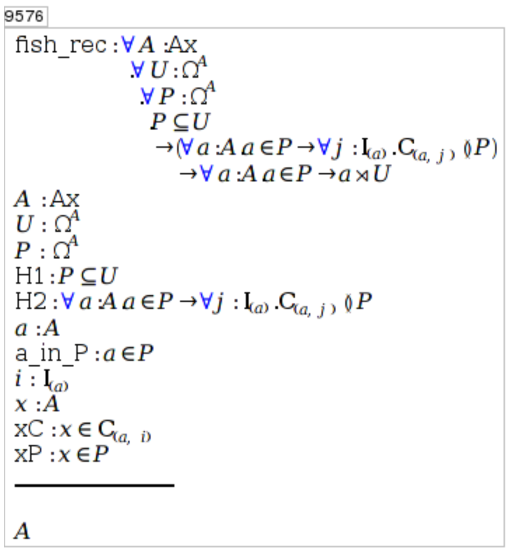
$\mathbb{C}_{\langle a,\ i\rangle} \rtimes U$

The goal is now the existence of a point in `C a i` fished by `U`. We thus need to use the introduction rule for the existential quantifier. In CIC it is a defined notion, that is an inductive type with just one constructor (one introduction rule) holding the witness and the proof that the witness satisfies a proposition.

    ncheck Ex.

Instead of trying to remember the name of the constructor, that should be used as the argument of `napply`, we can ask the system to find by itself the constructor name and apply it with the `@` tactic. Note that some inductive predicates, like the disjunction, have multiple introduction rules, and thus `@` can be followed by a number identifying the constructor.

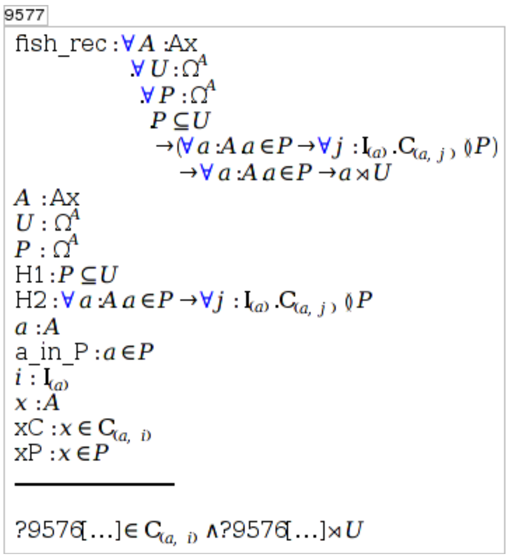
```
@;
```



9576

$\mathrm{fish\_rec} : \forall A{:}\mathrm{Ax}.\ \forall U{:}\Omega^A.\ \forall P{:}\Omega^A.\ P \subseteq U \to (\forall a{:}A.\, a \in P \to \forall j : \mathbb{I}_{\langle a\rangle}.\, \mathbb{C}_{\langle a,\ j\,\rangle} \between P) \to \forall a{:}A.\, a \in P \to a \rtimes U$
$A : \mathrm{Ax}$
$U : \Omega^A$
$P : \Omega^A$
$\mathrm{H1} : P \subseteq U$
$\mathrm{H2} : \forall a{:}A.\, a \in P \to \forall j : \mathbb{I}_{\langle a\rangle}.\, \mathbb{C}_{\langle a,\ j\,\rangle} \between P$
$a : A$
$\mathrm{a\_in\_P} : a \in P$
$i : \mathbb{I}_{\langle a\rangle}$
$x : A$
$\mathrm{xC} : x \in \mathbb{C}_{\langle a,\ i\rangle}$
$\mathrm{xP} : x \in P$

A



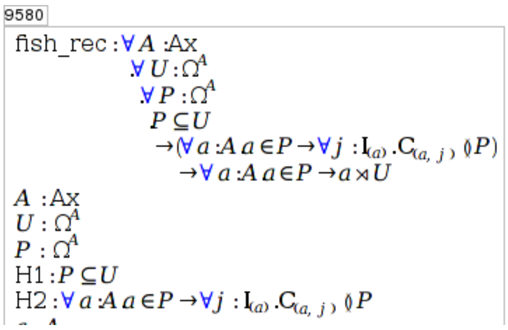
9577

$\mathrm{fish\_rec} : \forall A{:}\mathrm{Ax}.\ \forall U{:}\Omega^A.\ \forall P{:}\Omega^A.\ P \subseteq U \to (\forall a{:}A.\, a \in P \to \forall j : \mathbb{I}_{\langle a\rangle}.\, \mathbb{C}_{\langle a,\ j\,\rangle} \between P) \to \forall a{:}A.\, a \in P \to a \rtimes U$
$A : \mathrm{Ax}$
$U : \Omega^A$
$P : \Omega^A$
$\mathrm{H1} : P \subseteq U$
$\mathrm{H2} : \forall a{:}A.\, a \in P \to \forall j : \mathbb{I}_{\langle a\rangle}.\, \mathbb{C}_{\langle a,\ j\,\rangle} \between P$
$a : A$
$\mathrm{a\_in\_P} : a \in P$
$i : \mathbb{I}_{\langle a\rangle}$
$x : A$
$\mathrm{xC} : x \in \mathbb{C}_{\langle a,\ i\rangle}$
$\mathrm{xP} : x \in P$

$?9576[\ldots] \in \mathbb{C}_{\langle a,\ i\rangle} \wedge ?9576[\ldots] \rtimes U$

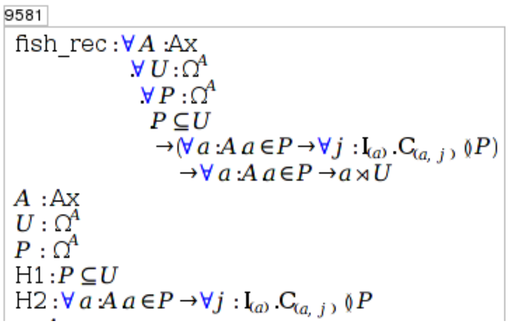
After choosing `x` as the witness, we have to prove a conjunction, and we again apply the introduction rule for the inductively defined predicate `∧`.

```
##[ napply x
##| @;
```



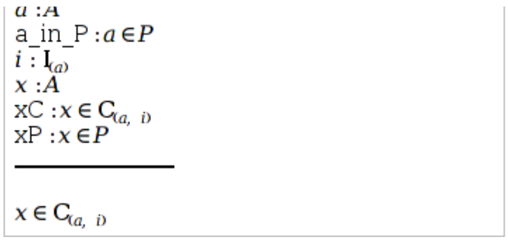
9580

$\mathrm{fish\_rec} : \forall A{:}\mathrm{Ax}.\ \forall U{:}\Omega^A.\ \forall P{:}\Omega^A.\ P \subseteq U \to (\forall a{:}A.\, a \in P \to \forall j : \mathbb{I}_{\langle a\rangle}.\, \mathbb{C}_{\langle a,\ j\,\rangle} \between P) \to \forall a{:}A.\, a \in P \to a \rtimes U$
$A : \mathrm{Ax}$
$U : \Omega^A$
$P : \Omega^A$
$\mathrm{H1} : P \subseteq U$
$\mathrm{H2} : \forall a{:}A.\, a \in P \to \forall j : \mathbb{I}_{\langle a\rangle}.\, \mathbb{C}_{\langle a,\ j\,\rangle} \between P$

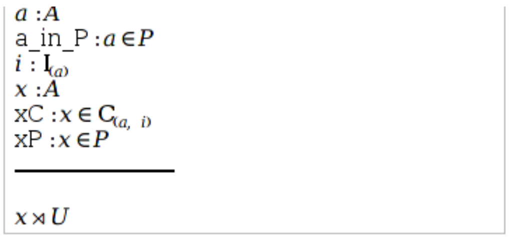


9581

$\mathrm{fish\_rec} : \forall A{:}\mathrm{Ax}.\ \forall U{:}\Omega^A.\ \forall P{:}\Omega^A.\ P \subseteq U \to (\forall a{:}A.\, a \in P \to \forall j : \mathbb{I}_{\langle a\rangle}.\, \mathbb{C}_{\langle a,\ j\,\rangle} \between P) \to \forall a{:}A.\, a \in P \to a \rtimes U$
$A : \mathrm{Ax}$
$U : \Omega^A$
$P : \Omega^A$
$\mathrm{H1} : P \subseteq U$
$\mathrm{H2} : \forall a{:}A.\, a \in P \to \forall j : \mathbb{I}_{\langle a\rangle}.\, \mathbb{C}_{\langle a,\ j\,\rangle} \between P$



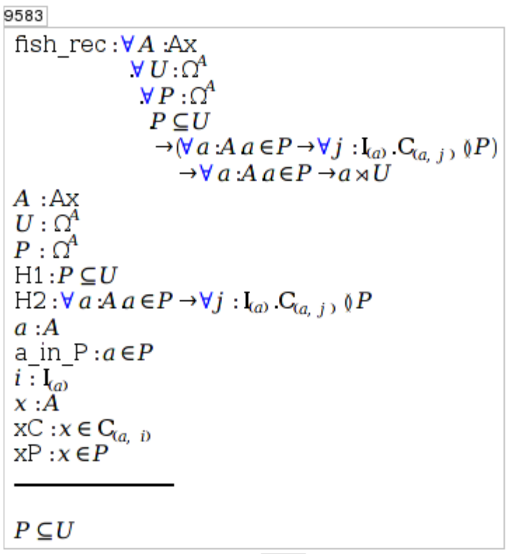
```
a :A
a_in_P : a ∈ P
i : I⟨a⟩
x :A
xC : x ∈ C⟨a, i⟩
xP : x ∈ P
──────────

x ∈ C⟨a, i⟩
```



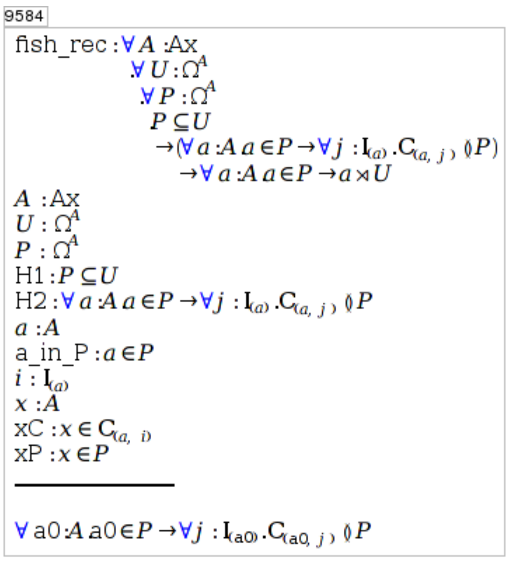
```
a :A
a_in_P : a ∈ P
i : I⟨a⟩
x :A
xC : x ∈ C⟨a, i⟩
xP : x ∈ P
──────────

x ⋉ U
```

The left hand side of the conjunction is trivial to prove, since it is already in the context. The right hand side needs to perform the co-recursive call.

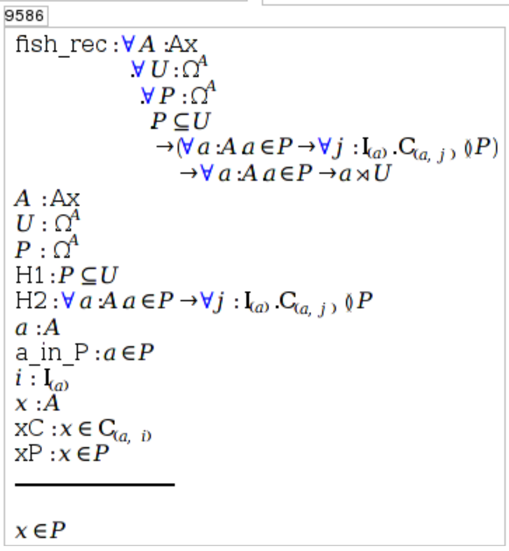
```
          ##[ napply xC;
          ##| napply (fish_rec ? U P);
```



```
9583
fish_rec : ∀A :Ax.
             ∀U : Ω^A.
              ∀P : Ω^A.
               P ⊆ U
                →(∀a :A.a ∈ P → ∀j : I⟨a⟩.C⟨a, j⟩ ≬ P)
                  → ∀a :A.a ∈ P → a ⋉ U
A :Ax
U : Ω^A
P : Ω^A
H1 : P ⊆ U
H2 : ∀a :A.a ∈ P → ∀j : I⟨a⟩.C⟨a, j⟩ ≬ P
a :A
a_in_P : a ∈ P
i : I⟨a⟩
x :A
xC : x ∈ C⟨a, i⟩
xP : x ∈ P
──────────

P ⊆ U
```



```
9584
fish_rec : ∀A :Ax.
             ∀U : Ω^A.
              ∀P : Ω^A.
               P ⊆ U
                →(∀a :A.a ∈ P → ∀j : I⟨a⟩.C⟨a, j⟩ ≬ P)
                  → ∀a :A.a ∈ P → a ⋉ U
A :Ax
U : Ω^A
P : Ω^A
H1 : P ⊆ U
H2 : ∀a :A.a ∈ P → ∀j : I⟨a⟩.C⟨a, j⟩ ≬ P
a :A
a_in_P : a ∈ P
i : I⟨a⟩
x :A
xC : x ∈ C⟨a, i⟩
xP : x ∈ P
──────────

∀a0:A.a0 ∈ P → ∀j : I⟨a0⟩.C⟨a0, j⟩ ≬ P
```



```
9586
fish_rec : ∀A :Ax.
             ∀U : Ω^A.
              ∀P : Ω^A.
               P ⊆ U
                →(∀a :A.a ∈ P → ∀j : I⟨a⟩.C⟨a, j⟩ ≬ P)
                  → ∀a :A.a ∈ P → a ⋉ U
A :Ax
U : Ω^A
P : Ω^A
H1 : P ⊆ U
H2 : ∀a :A.a ∈ P → ∀j : I⟨a⟩.C⟨a, j⟩ ≬ P
a :A
a_in_P : a ∈ P
i : I⟨a⟩
x :A
xC : x ∈ C⟨a, i⟩
xP : x ∈ P
──────────

x ∈ P
```

The co-recursive call needs some arguments, but all of them are in the context. Instead of explicitly mention them, we use the `nassumption` tactic, that simply tries to apply every context item.

```
              nassumption;
          ##]
      ##]
  ##]
  nqed.
```

Subset of covered/fished points

We now have to define the subset of `S` of points covered by `U`. We also define a prefix notation for it. Remember that the precedence of the prefix form of a symbol has to be higher than the precedence of its infix form.

```
ndefinition coverage : ∀A:Ax.∀U:Ω^A.Ω^A ≝ λA,U.{ a | a ◃ U }.

notation "◃U" non associative with precedence 55 for @{ 'coverage $U }.

interpretation "coverage cover" 'coverage U = (coverage ? U).
```

Here we define the equation characterizing the cover relation. Even if it is not part of the paper, we proved that `◃(U)` is the minimum solution for such equation, the interested reader should be able to reply the proof with Matita.

```
ndefinition cover_equation : ∀A:Ax.∀U,X:Ω^A.CProp[0] ≝  λA,U,X.
  ∀a.a ∈ X ↔ (a ∈ U ∨ ∃i:𝐈 a.∀y.y ∈ 𝐂 a i → y ∈ X).

ntheorem coverage_cover_equation : ∀A,U. cover_equation A U (◃U).
#A; #U; #a; @; #H;
##[ nelim H; #b;
    ##[ #bU; @1; nassumption;
    ##| #i; #CaiU; #IH; @2; @ i; #c; #cCbi; ncases (IH ? cCbi);
        ##[ #E; @; napply E;
        ##| #_; napply CaiU; nassumption; ##] ##]
##| ncases H; ##[ #E; @; nassumption]
    *; #j; #Hj; @2 j; #w; #wC; napply Hj; nassumption;
##]
nqed.

ntheorem coverage_min_cover_equation :
  ∀A,U,W. cover_equation A U W → ◃U ⊆ W.
#A; #U; #W; #H; #a; #aU; nelim aU; #b;
##[ #bU; ncases (H b); #_; #H1; napply H1; @1; nassumption;
##| #i; #CbiU; #IH; ncases (H b); #_; #H1; napply H1; @2; @i; napply IH;
##]
nqed.
```

We similarly define the subset of points "fished" by `F`, the equation characterizing `⋉(F)` and prove that fish is the biggest solution for such equation.

```
notation "⋉F" non associative with precedence 55
for @{ 'fished $F }.

ndefinition fished : ∀A:Ax.∀F:Ω^A.Ω^A ≝ λA,F.{ a | a ⋉ F }.

interpretation "fished fish" 'fished F = (fished ? F).

ndefinition fish_equation : ∀A:Ax.∀F,X:Ω^A.CProp[0] ≝ λA,F,X.
  ∀a. a ∈ X ↔ a ∈ F ∧ ∀i:𝐈 a.∃y.y ∈ 𝐂 a i ∧ y ∈ X.

ntheorem fished_fish_equation : ∀A,F. fish_equation A F (⋉F).
#A; #F; #a; @; (* *; non genera outtype che lega a *) #H; ncases H;
##[ #b; #bF; #H2; @ bF; #i; ncases (H2 i); #c; *; #cC; #cF; @c; @ cC;
    napply cF;
##| #aF; #H1; @ aF; napply H1;
##]
nqed.

ntheorem fished_max_fish_equation : ∀A,F,G. fish_equation A F G → G ⊆ ⋉F.
#A; #F; #G; #H; #a; #aG; napply (fish_rec … aG);
#b; ncases (H b); #H1; #_; #bG; ncases (H1 bG); #E1; #E2; nassumption;
nqed.
```

Part 2, the new set of axioms

Since the name of defined objects (record included) has to be unique within the same file, we prefix every field name in the new definition of the axiom set with `n`.

```
nrecord nAx : Type[1] ≝ {
    nS:> Type[0];
    nI: nS → Type[0];
    nD: ∀a:nS. nI a → Type[0];
    nd: ∀a:nS. ∀i:nI a. nD a i → nS
  }.
```

We again define a notation for the projections, making the projected record an implicit argument. Note that, since we already have a notation for **I**, we just add another interpretation for it. The system, looking at the argument of **I**, will be able to choose the correct interpretation.

```
notation "D \sub ( ⦇a,\emsp i⦈ )" non associative with precedence 70 for @{ 'D $a $i
notation "d \sub ( ⦇a,\emsp i,\emsp j⦈ )" non associative with precedence 70 for @{

notation > "D term 90 a term 90 i" non associative with precedence 70 for @{ 'D $a $
notation > "d term 90 a term 90 i term 90 j" non associative with precedence 70 for

interpretation "D" 'D a i = (nD ? a i).
interpretation "d" 'd a i j = (nd ? a i j).
interpretation "new I" 'I a = (nI ? a).
```

The first result the paper presents to motivate the new formulation of the axiom set is the possibility to define and old axiom set starting from a new one and vice versa. The key definition for such construction is the image of d(a,i). The paper defines the image as

Im[d(a,i)] = { d(a,i,j) | j : D(a,i) }

but this not so formal notation poses some problems. The image is often used as the left hand side of the ⊆ predicate

Im[d(a,i)] ⊆ V

Of course this writing is interpreted by the authors as follows

∀j:D(a,i). d(a,i,j) ∈ V

If we need to use the image to define **C** (a subset of `S`) we are obliged to form a subset, i.e. to place a single variable `{ here | … }` of type `S`.

Im[d(a,i)] = { y | ∃j:D(a,i). y = d(a,i,j) }

This poses no theoretical problems, since `S` is a Type and thus equipped with the `Id` equality. If `S` was a setoid, here the equality would have been the one of the setoid.

Unless we define two different images, one for stating that the image is ⊆ of something and another one to define **C**, we end up using always the latter. Thus the statement `Im[d(a,i)] ⊆ V` unfolds to

∀x:S. (∃j.x = d(a,i,j)) → x ∈ V

That, up to rewriting with the equation defining `x`, is what we mean. Since we decided to use `Id` the rewriting always work (the elimination principle for `Id` is Leibniz's equality, that is quantified over the context.

The problem that arises if we decide to make `S` a setoid is that `V` has to be extensional w.r.t. the equality of `S` (i.e. the characteristic functional proposition has to quotient its input with a relation bigger than the one of `S`.

∀x,y:S. x = y → x ∈ V → y ∈ V

If `V` is a complex construction, the proof may not be trivial.

```
include "logic/equality.ma".
```

```
ndefinition image ≝ λA:nAx.λa:A.λi. { x | ∃j:𝐃 a i. x = 𝐝 a i j }.

notation > "Im  [d term 90 a term 90 i]" non associative with precedence 70 for @{ ']
notation < "Im  [d \sub ( ❨a,\emsp i❩ )]" non associative with precedence 70 for @{ '

interpretation "image" 'Im a i = (image ? a i).
```

Thanks to our definition of image, we can define a function mapping a new axiom set to an old one and vice versa. Note that in the second definition, when we give the **d** component, the projection of the Σ-type is inlined (constructed on the fly by `*;`) while in the paper it was named `fst`.

```
ndefinition Ax_of_nAx : nAx → Ax.
#A; @ A (nI ?); #a; #i; napply (Im [d a i]);
nqed.

ndefinition nAx_of_Ax : Ax → nAx.
#A; @ A (I ?);
##[ #a; #i; napply (Σx:A.x ∈ C a i);
##| #a; #i; *; #x; #_; napply x;
##]
nqed.
```

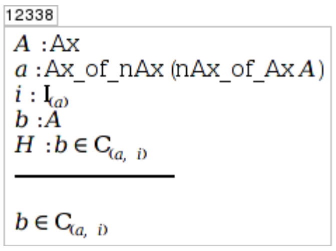
We now prove that the two function above form a retraction pair for the `C` component of the axiom set. To prove that we face a little problem since CIC is not equipped with η-conversion. This means that the followin does not hold (where `A` is an axiom set).

A = (S A, I A, C A)

This can be proved only under a pattern mach over `A`, that means that the resulting computation content of the proof is a program that computes something only if `A` is a concrete axiom set.

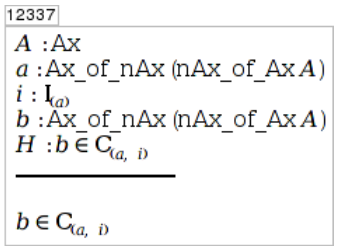
To state the lemma we have to drop notation, and explicitly give the axiom set in input to the `C` projection.

```
nlemma Ax_nAx_equiv :
  ∀A:Ax. ∀a,i. C (Ax_of_nAx (nAx_of_Ax A)) a i ⊆ C A a i ∧
               C A a i ⊆ C (Ax_of_nAx (nAx_of_Ax A)) a i.
#A; #a; #i; @; #b; #H;
```


12338

A : Ax
a : Ax_of_nAx (nAx_of_Ax A)
i : $\mathbf{I}_{(a)}$
b : A
H : $b \in \mathbf{C}_{(a,\ i)}$

$b \in \mathbf{C}_{(a,\ i)}$

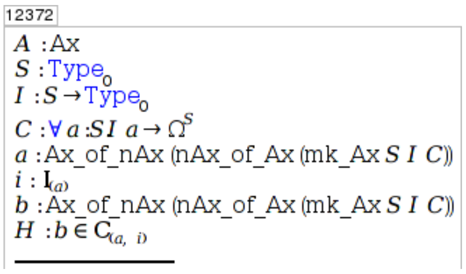

12337

A : Ax
a : Ax_of_nAx (nAx_of_Ax A)
i : $\mathbf{I}_{(a)}$
b : Ax_of_nAx (nAx_of_Ax A)
H : $b \in \mathbf{C}_{(a,\ i)}$

$b \in \mathbf{C}_{(a,\ i)}$

Look for example the type of `a`. The command `nchange in a with A` would fail because of the missing η-conversion rule. We have thus to pattern match over `A` and introduce its pieces.

```
##[  ncases A in a i b H; #S; #I; #C; #a; #i; #b; #H;
```


12372

A : Ax
S : Type_0
I : $S \to \mathrm{Type}_0$
C : $\forall a : S\, I\ a \to \Omega^S$
a : Ax_of_nAx (nAx_of_Ax (mk_Ax S I C))
i : $\mathbf{I}_{(a)}$
b : Ax_of_nAx (nAx_of_Ax (mk_Ax S I C))
H : $b \in \mathbf{C}_{(a,\ i)}$

$$b \in C\ a\ i$$

Now the system accepts that the type of `a` is the fist component of the axiom set, now called `S`. Unfolding definitions in `H` we discover there is still some work to do.

```
        nchange in a with S; nwhd in H;
```

12374

$$\begin{array}{l}
A : \mathrm{Ax} \\
S : \mathrm{Type}_0 \\
I : S \to \mathrm{Type}_0 \\
C : \forall a : S\, I\ a \to \Omega^S \\
a : S \\
i : \mathbf{I}_{\langle a \rangle} \\
b : \mathrm{Ax\_of\_nAx}\ (\mathrm{nAx\_of\_Ax}\ (\mathrm{mk\_Ax}\ S\ I\ C)) \\
H : \exists j : \mathrm{D}_{\langle a,\ i \rangle}\ b = \mathbf{d}_{\langle a,\ i,\ j \rangle} \\
\hline
b \in C\ a\ i
\end{array}$$

To use the equation defining `b` we have to eliminate `H`. Unfolding definitions in `x` tell us there is still something to do. The `nrewrite` tactic is a shortcut for the elimination principle of the equality. It accepts an additional argument `<` or `>` to rewrite left-to-right or right-to-left.

```
        ncases H; #x; #E; nrewrite > E; nwhd in x;
```

12400

$$\begin{array}{l}
A : \mathrm{Ax} \\
S : \mathrm{Type}_0 \\
I : S \to \mathrm{Type}_0 \\
C : \forall a : S\, I\ a \to \Omega^S \\
a : S \\
i : \mathbf{I}_{\langle a \rangle} \\
b : \mathrm{Ax\_of\_nAx}\ (\mathrm{nAx\_of\_Ax}\ (\mathrm{mk\_Ax}\ S\ I\ C)) \\
H : \exists j : \mathrm{D}_{\langle a,\ i \rangle}\ b = \mathbf{d}_{\langle a,\ i,\ j \rangle} \\
x : \Sigma x : S\, x \in C\ a\ i \\
E : b = \mathbf{d}_{\langle a,\ i,\ x \rangle} \\
\hline
\mathbf{d}_{\langle a,\ i,\ x \rangle} \in C\ a\ i
\end{array}$$

We defined `𝐝` to be the first projection of `x`, thus we have to eliminate `x` to actually compute `𝐝`.

The remaining part of the proof it not interesting and poses no new problems.

```
        ncases x; #b; #Hb; nnormalize; nassumption;
    ##|  ncases A in a i b H; #S; #I; #C; #a; #i; #b; #H; @;
        ##[ @ b; nassumption;
        ##| nnormalize; @; ##]
    ##]
    nqed.
```

We then define the inductive type of ordinals, parametrized over an axiom set. We also attach some notations to the constructors.

```
    ninductive Ord (A : nAx) : Type[0] ≝
     | oO : Ord A
     | oS : Ord A → Ord A
     | oL : ∀a:A.∀i.∀f:𝐃 a i → Ord A. Ord A.

    notation "0" non associative with precedence 90 for @{ 'oO }.
    notation "x+1" non associative with precedence 50 for @{'oS $x }.
    notation "Λ term 90 f" non associative with precedence 50 for @{ 'oL $f }.

    interpretation "ordinals Zero" 'oO = (oO ?).
    interpretation "ordinals Succ" 'oS x = (oS ? x).
```

```
interpretation "ordinals Lambda" 'oL f = (oL ? ? ? f).
```

The definition of `U⎽x` is by recursion over the ordinal `x`. We thus define a recursive function using the `nlet rec` command. The `on x` directive tells the system on which argument the function is (structurally) recursive.

In the `oS` case we use a local definition to name the recursive call since it is used twice.

Note that Matita does not support notation in the left hand side of a pattern match, and thus the names of the constructors have to be spelled out verbatim.

```
nlet rec famU (A : nAx) (U : Ω^A) (x : Ord A) on x : Ω^A ≝
  match x with
  [ oO ⇒ U
  | oS y ⇒ let U_n ≝ famU A U y in U_n ∪ { x | ∃i.Im[d x i] ⊆ U_n}
  | oL a i f ⇒ { x | ∃j.x ∈ famU A U (f j) } ].

notation < "term 90 U \sub (term 90 x)" non associative with precedence 50 for @{ 'f
notation > "U ⎽ term 90 x" non associative with precedence 50 for @{ 'famU $U $x }.

interpretation "famU" 'famU U x = (famU ? U x).
```

We attach as the input notation for U_x the similar `U⎽x` where underscore, that is a character valid for identifier names, has been replaced by `⎽` that is not. The symbol `⎽` can act as a separator, and can be typed as an alternative for `_` (i.e. pressing ALT-L after `_`).

The notion ◃(U) has to be defined as the subset of elements `y` belonging to `U⎽x` for some `x`. Moreover, we have to define the notion of cover between sets again, since the one defined at the beginning of the tutorial works only for the old axiom set.

```
ndefinition ord_coverage : ∀A:nAx.∀U:Ω^A.Ω^A ≝
  λA,U.{ y | ∃x:Ord A. y ∈ famU ? U x }.

ndefinition ord_cover_set ≝ λc:∀A:nAx.Ω^A → Ω^A.λA,C,U.
  ∀y.y ∈ C → y ∈ c A U.

interpretation "coverage new cover" 'coverage U = (ord_coverage ? U).
interpretation "new covers set" 'covers a U = (ord_cover_set ord_coverage ? a U).
interpretation "new covers" 'covers a U = (mem ? (ord_coverage ? U) a).
```

Before proving that this cover relation validates the reflexivity and infinity rules, we prove this little technical lemma that is used in the proof for the infinity rule.

```
nlemma ord_subset: ∀A:nAx.∀a:A.∀i,f,U.∀j:D a i. U⎽(f j) ⊆ U⎽(Λ f).
#A; #a; #i; #f; #U; #j; #b; #bUf; @ j; nassumption;
nqed.
```

The proof of infinity uses the following form of the Axiom of Choice, that cannot be proved inside Matita, since the existential quantifier lives in the sort of predicative propositions while the sigma in the conclusion lives in the sort of data types, and thus the former cannot be eliminated to provide the witness for the second.

```
naxiom AC : ∀A,a,i,U.
  (∀j:D a i.∃x:Ord A.d a i j ∈ U⎽x) → (Σf.∀j:D a i.d a i j ∈ U⎽(f j)).
```

Note that, if we will decide later to identify ∃ and Σ, `AC` is trivially provable

```
nlemma AC_exists_is_sigma : ∀A,a,i,U.
  (∀j:D a i.Σx:Ord A.d a i j ∈ U⎽x) → (Σf.∀j:D a i.d a i j ∈ U⎽(f j)).
#A; #a; #i; #U; #H; @;
##[ #j; ncases (H j); #x; #_; napply x;
##| #j; ncases (H j); #x; #Hx; napply Hx; ##]
nqed.
```

In case we made `S` a setoid, the following property has to be proved

nlemma Uxis_ext: ∀A:nAx.∀a,b:A.∀x.∀U. a = b → b ∈ U_x → a ∈ U_x.

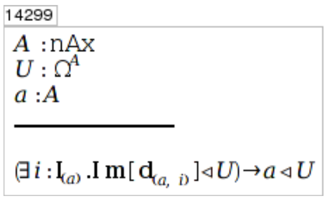
Anyway this proof is a non trivial induction over x, that requires `I` and `D` to be declared as morphisms.

The reflexivity proof is trivial, it is enough to provide the ordinal `0` as a witness, then `◃(U)` reduces to `U` by definition, hence the conclusion. Note that `0` is between `(` and `)` to make it clear that it is a term (an ordinal) and not the number of the constructor we want to apply (that is the first and only one of the existential inductive type).

```
ntheorem new_coverage_reflexive: ∀A:nAx.∀U:Ω^A.∀a. a ∈ U → a ◃ U.
#A; #U; #a; #H; @ (0); napply H;
nqed.
```

We now proceed with the proof of the infinity rule.

```
alias symbol "covers" (instance 3) = "new covers set".
ntheorem new_coverage_infinity:
  ∀A:nAx.∀U:Ω^A.∀a:A. (∃i:I a. Im[d a i] ◃ U) → a ◃ U.
#A; #U; #a;
```

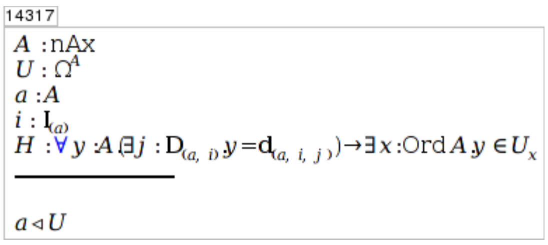

14299

A : nAx
$U : \Omega^A$
$a : A$

$(\exists i : I_{(a)} . \mathrm{Im}[d_{(a,\ i)}] \triangleleft U) \rightarrow a \triangleleft U$

We eliminate the existential, obtaining an `i` and a proof that the image of `d a i` is covered by U. The `nnormalize` tactic computes the normal form of `H`, thus expands the definition of cover between sets.

```
*; #i; #H; nnormalize in H;
```

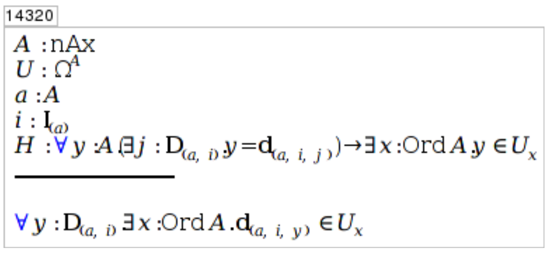

14317

A : nAx
$U : \Omega^A$
$a : A$
$i : I_{(a)}$
$H : \forall y : A . (\exists j : D_{(a,\ i)} . y = d_{(a,\ i,\ j)}) \rightarrow \exists x : \mathrm{Ord}\, A . y \in U_x$

$a \triangleleft U$

When the paper proof considers `H`, it implicitly substitutes assumed equation defining `y` in its conclusion. In Matita this step is not completely trivial. We thus assert (`ncut`) the nicer form of `H` and prove it.

```
ncut (∀y:D a i.∃x:Ord A.d a i y ∈ U_x); ##[
```

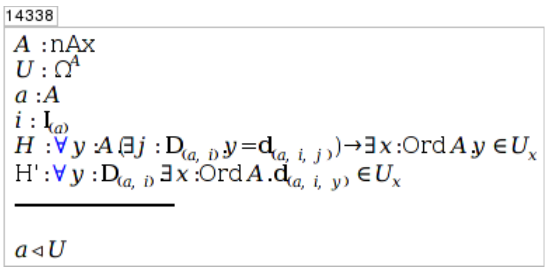

14320

A : nAx
$U : \Omega^A$
$a : A$
$i : I_{(a)}$
$H : \forall y : A . (\exists j : D_{(a,\ i)} . y = d_{(a,\ i,\ j)}) \rightarrow \exists x : \mathrm{Ord}\, A . y \in U_x$

$\forall y : D_{(a,\ i)} \exists x : \mathrm{Ord}\, A . d_{(a,\ i,\ y)} \in U_x$

After introducing `z`, `H` can be applied (choosing `d a i z` as `y`). What is the left to prove is that `∃j: D a j. d a i z = d a i j`, that becomes trivial if `j` is chosen to be `z`.

```
    #z; napply H; @ z; @; ##] #H';
```

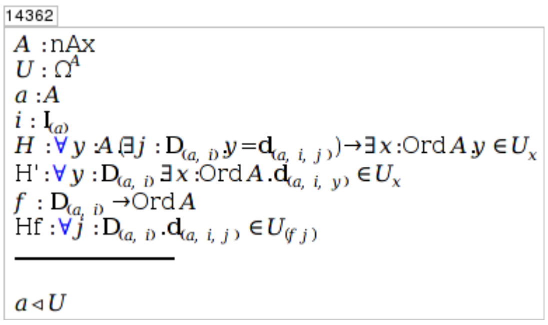

14338

$A : \mathrm{nAx}$
$U : \Omega^A$
$a : A$
$i : \mathbf{I}_{\langle a \rangle}$
$H : \forall y : A.(\exists j : \mathbf{D}_{\langle a,\ i \rangle}.y = \mathbf{d}_{\langle a,\ i,\ j \rangle}) \to \exists x : \mathrm{Ord}\, A.y \in U_x$
$\mathrm{H}' : \forall y : \mathbf{D}_{\langle a,\ i \rangle}.\exists x : \mathrm{Ord}\, A.\mathbf{d}_{\langle a,\ i,\ y \rangle} \in U_x$

$a \triangleleft U$

Under `H'` the axiom of choice `AC` can be eliminated, obtaining the `f` and its property. Note that the axiom `AC` was abstracted over `A,a,i,U` before assuming `(∀j:D a i.∃x:Ord A.d a i j ∈ U_x)`. Thus the term that can be eliminated is `AC ???? H'` where the system is able to infer every `?`. Matita provides a facility to specify a number of `?` in a compact way, i.e. `…`. The system expand `…` first to zero, then one, then two, three and finally four question marks, "guessing" how may of them are needed.

```
  ncases (AC … H'); #f; #Hf;
```

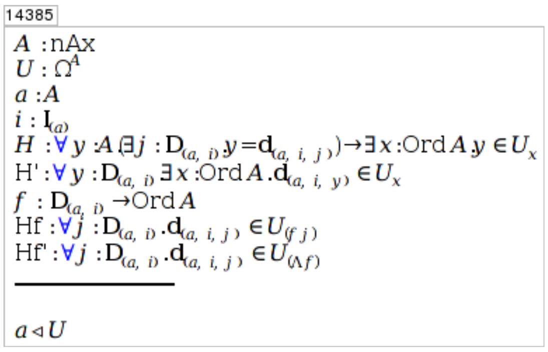

14362

$A : \mathrm{nAx}$
$U : \Omega^A$
$a : A$
$i : \mathbf{I}_{\langle a \rangle}$
$H : \forall y : A.(\exists j : \mathbf{D}_{\langle a,\ i \rangle}.y = \mathbf{d}_{\langle a,\ i,\ j \rangle}) \to \exists x : \mathrm{Ord}\, A.y \in U_x$
$\mathrm{H}' : \forall y : \mathbf{D}_{\langle a,\ i \rangle}.\exists x : \mathrm{Ord}\, A.\mathbf{d}_{\langle a,\ i,\ y \rangle} \in U_x$
$f : \mathbf{D}_{\langle a,\ i \rangle} \to \mathrm{Ord}\, A$
$\mathrm{Hf} : \forall j : \mathbf{D}_{\langle a,\ i \rangle}.\mathbf{d}_{\langle a,\ i,\ j \rangle} \in U_{\langle f\ j \rangle}$

$a \triangleleft U$

The paper proof does now a forward reasoning step, deriving (by the ord_subset lemma we proved above) `Hf'` i.e. **d** a i j ∈ U_(Λf).

```
  ncut (∀j.d a i j ∈ U_(Λ f));
    ##[ #j; napply (ord_subset … f … (Hf j));##] #Hf';
```

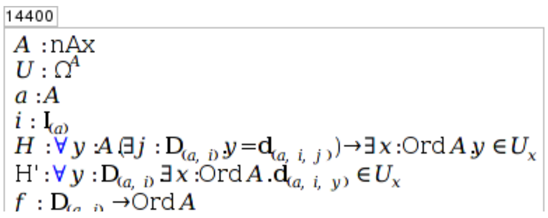

14385

$A : \mathrm{nAx}$
$U : \Omega^A$
$a : A$
$i : \mathbf{I}_{\langle a \rangle}$
$H : \forall y : A.(\exists j : \mathbf{D}_{\langle a,\ i \rangle}.y = \mathbf{d}_{\langle a,\ i,\ j \rangle}) \to \exists x : \mathrm{Ord}\, A.y \in U_x$
$\mathrm{H}' : \forall y : \mathbf{D}_{\langle a,\ i \rangle}.\exists x : \mathrm{Ord}\, A.\mathbf{d}_{\langle a,\ i,\ y \rangle} \in U_x$
$f : \mathbf{D}_{\langle a,\ i \rangle} \to \mathrm{Ord}\, A$
$\mathrm{Hf} : \forall j : \mathbf{D}_{\langle a,\ i \rangle}.\mathbf{d}_{\langle a,\ i,\ j \rangle} \in U_{\langle f\ j \rangle}$
$\mathrm{Hf}' : \forall j : \mathbf{D}_{\langle a,\ i \rangle}.\mathbf{d}_{\langle a,\ i,\ j \rangle} \in U_{\langle \Lambda f \rangle}$

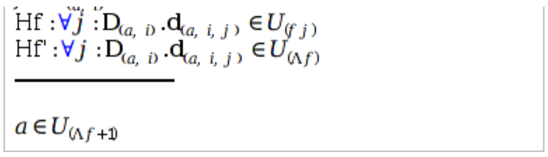
$a \triangleleft U$

To prove that `a◃U` we have to exhibit the ordinal x such that `a ∈ U_x`.

```
  @ (Λ f+1);
```


14400

$A : \mathrm{nAx}$
$U : \Omega^A$
$a : A$
$i : \mathbf{I}_{\langle a \rangle}$
$H : \forall y : A.(\exists j : \mathbf{D}_{\langle a,\ i \rangle}.y = \mathbf{d}_{\langle a,\ i,\ j \rangle}) \to \exists x : \mathrm{Ord}\, A.y \in U_x$
$\mathrm{H}' : \forall y : \mathbf{D}_{\langle a,\ i \rangle}.\exists x : \mathrm{Ord}\, A.\mathbf{d}_{\langle a,\ i,\ y \rangle} \in U_x$
$f : \mathbf{D}_{\langle a,\ i \rangle} \to \mathrm{Ord}\, A$

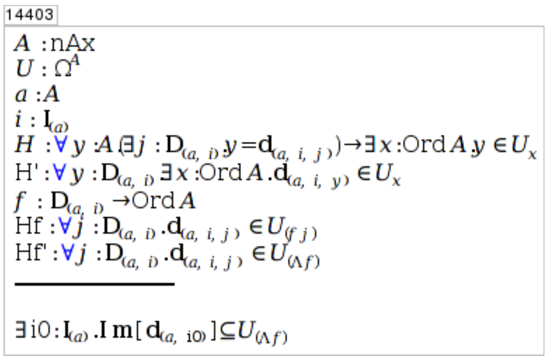


$Hf : \forall j : D_{(a,\ i)} . d_{(a,\ i,\ j)} \in U_{(f\ j)}$
$Hf' : \forall j : D_{(a,\ i)} . d_{(a,\ i,\ j)} \in U_{(\wedge f)}$

$a \in U_{(\wedge f+1)}$

The definition of `U_(…+1)` expands to the union of two sets, and proving that `a ∈ X ∪ Y` is, by definition, equivalent to prove that `a` is in `X` or `Y`. Applying the second constructor `@2;` of the disjunction, we are left to prove that `a` belongs to the right hand side of the union.

```
@2;
```



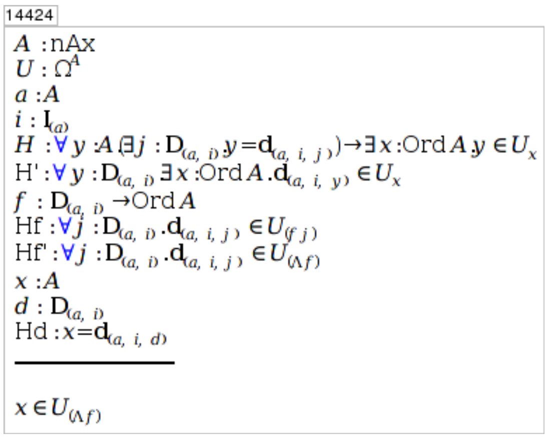
14403

$A : nAx$
$U : \Omega^{A}$
$a : A$
$i : I_{(a)}$
$H : \forall y : A . (\exists j : D_{(a,\ i)} . y = d_{(a,\ i,\ j)}) \to \exists x : Ord\ A . y \in U_{x}$
$H' : \forall y : D_{(a,\ i)} . \exists x : Ord\ A . d_{(a,\ i,\ y)} \in U_{x}$
$f : D_{(a,\ i)} \to Ord\ A$
$Hf : \forall j : D_{(a,\ i)} . d_{(a,\ i,\ j)} \in U_{(f\ j)}$
$Hf' : \forall j : D_{(a,\ i)} . d_{(a,\ i,\ j)} \in U_{(\wedge f)}$

$\exists i0 : I_{(a)} . Im[d_{(a,\ i0)}] \subseteq U_{(\wedge f)}$

We thus provide `i` as the witness of the existential, introduce the element being in the image and we are left to prove that it belongs to `U_(∧f)`. In the meanwhile, since belonging to the image means that there exists an object in the domain …, we eliminate the existential, obtaining `d` (of type `D a i`) and the equation defining `x`.

```
@i; #x; *; #d; #Hd;
```



14424

$A : nAx$
$U : \Omega^{A}$
$a : A$
$i : I_{(a)}$
$H : \forall y : A . (\exists j : D_{(a,\ i)} . y = d_{(a,\ i,\ j)}) \to \exists x : Ord\ A . y \in U_{x}$
$H' : \forall y : D_{(a,\ i)} . \exists x : Ord\ A . d_{(a,\ i,\ y)} \in U_{x}$
$f : D_{(a,\ i)} \to Ord\ A$
$Hf : \forall j : D_{(a,\ i)} . d_{(a,\ i,\ j)} \in U_{(f\ j)}$
$Hf' : \forall j : D_{(a,\ i)} . d_{(a,\ i,\ j)} \in U_{(\wedge f)}$
$x : A$
$d : D_{(a,\ i)}$
$Hd : x = d_{(a,\ i,\ d)}$

$x \in U_{(\wedge f)}$

We just need to use the equational definition of `x` to obtain a conclusion that can be proved with `Hf'`. We assumed that `U_x` is extensional for every `x`, thus we are allowed to use `Hd` and close the proof.

```
nrewrite > Hd; napply Hf';
nqed.
```

The next proof is that ◁(U) is minimal. The hardest part of the proof is to prepare the goal for the induction. The desiderata is to prove `U_o ⊆ V` by induction on `o`, but the conclusion of the lemma is, unfolding all definitions:

∀x. x ∈ { y | ∃o:Ord A.y ∈ U_o } → x ∈ V

```
nlemma new_coverage_min :
  ∀A:nAx.∀U:Ω^A.∀V.U ⊆ V → (∀a:A.∀i.Im[d a i] ⊆ V → a ∈ V) → ◃U ⊆ V.
#A; #U; #V; #HUV; #Im;#b;
```

14483

$A : \mathrm{nAx}$
$U : \Omega^A$
$V : \Omega^A$
$\mathrm{HUV} : U \subseteq V$
$\mathrm{Im} : \forall a : A\, \forall i : \mathbf{I}_{(a)}.\mathbf{Im}[\mathbf{d}_{(a,\ i)}] \subseteq V \to a \in V$
$b : A$

$b \triangleleft U \to b \in V$

After all the introductions, event the element hidden in the ⊆ definition, we have to eliminate the existential quantifier, obtaining the ordinal `o`

```
*; #o;
```

14499

$A : \mathrm{nAx}$
$U : \Omega^A$
$V : \Omega^A$
$\mathrm{HUV} : U \subseteq V$
$\mathrm{Im} : \forall a : A\, \forall i : \mathbf{I}_{(a)}.\mathbf{Im}[\mathbf{d}_{(a,\ i)}] \subseteq V \to a \in V$
$b : A$
$o : \mathrm{Ord}\, A$

$b \in U_o \to b \in V$

What is left is almost right, but the element `b` is already in the context. We thus generalize every occurrence of `b` in the current goal, obtaining `∀c.c ∈ U⎽o → c ∈ V` that is `U⎽o ⊆ V`.

```
ngeneralize in match b; nchange with (U⎽o ⊆ V);
```

14510

$A : \mathrm{nAx}$
$U : \Omega^A$
$V : \Omega^A$
$\mathrm{HUV} : U \subseteq V$
$\mathrm{Im} : \forall a : A\, \forall i : \mathbf{I}_{(a)}.\mathbf{Im}[\mathbf{d}_{(a,\ i)}] \subseteq V \to a \in V$
$b : A$
$o : \mathrm{Ord}\, A$

$U_o \subseteq V$

We then proceed by induction on `o` obtaining the following goals

```
nelim o;
```

14518

$A : \mathrm{nAx}$
$U : \Omega^A$
$V : \Omega^A$
$\mathrm{HUV} : U \subseteq V$
$\mathrm{Im} : \forall a : A\, \forall i : \mathbf{I}_{(a)}.\mathbf{Im}[\mathbf{d}_{(a,\ i)}] \subseteq V \to a \in V$
$b : A$
$o : \mathrm{Ord}\, A$

$U_0 \subseteq V$

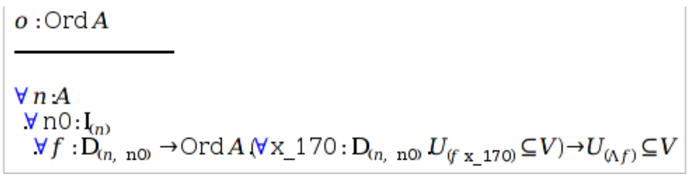
14519

$A : \mathrm{nAx}$
$U : \Omega^A$
$V : \Omega^A$
$\mathrm{HUV} : U \subseteq V$
$\mathrm{Im} : \forall a : A\, \forall i : \mathbf{I}_{(a)}.\mathbf{Im}[\mathbf{d}_{(a,\ i)}] \subseteq V \to a \in V$
$b : A$
$o : \mathrm{Ord}\, A$

$\forall O : \mathrm{Ord}\, A\; U_O \subseteq V \to U_{(O+1)} \subseteq V$

14520

$A : \mathrm{nAx}$
$U : \Omega^A$
$V : \Omega^A$
$\mathrm{HUV} : U \subseteq V$
$\mathrm{Im} : \forall a : A\, \forall i : \mathbf{I}_{(a)}.\mathbf{Im}[\mathbf{d}_{(a,\ i)}] \subseteq V \to a \in V$
$b : A$



$o : \mathrm{Ord}\, A$

$\forall n : A$
$. \forall \mathrm{n0} : \mathbf{I}_{(n)}$
$. \forall f : \mathbf{D}_{(n,\ \mathrm{n0})} \to \mathrm{Ord}\, A . (\forall \mathrm{x\_170} : \mathbf{D}_{(n,\ \mathrm{n0})}\, U_{(f\ \mathrm{x\_170})} \subseteq V) \to U_{(\Lambda f)} \subseteq V$

All of them can be proved using simple set theoretic arguments, the induction hypothesis and the assumption `Im`.

```
##[ napply HUV;
##| #p; #IH; napply subseteq_union_l; ##[ nassumption; ##]
    #x; *; #i; #H; napply (Im ? i); napply (subseteq_trans … IH); napply H;
##| #a; #i; #f; #IH; #x; *; #d; napply IH; ##]
nqed.
```

The notion `F_x` is again defined by recursion over the ordinal `x`.

```
nlet rec famF (A: nAx) (F : Ω^A) (x : Ord A) on x : Ω^A ≝
  match x with
  [ oO ⇒ F
  | oS o ⇒ let F_o ≝ famF A F o in F_o ∩ { x | ∀i:I x.∃j:D x i.d x i j ∈ F_o }
  | oL a i f ⇒ { x | ∀j:D a i.x ∈ famF A F (f j) }
  ].

interpretation "famF" 'famU U x = (famF ? U x).

ndefinition ord_fished : ∀A:nAx.∀F:Ω^A.Ω^A ≝ λA,F.{ y | ∀x:Ord A. y ∈ F_x }.

interpretation "fished new fish" 'fished U = (ord_fished ? U).
interpretation "new fish" 'fish a U = (mem ? (ord_fished ? U) a).
```

The proof of compatibility uses this little result, that we proved outside the main proof.

```
nlemma co_ord_subset: ∀A:nAx.∀F:Ω^A.∀a,i.∀f:D a i → Ord A.∀j. F_(Λ f) ⊆ F_(f j).
#A; #F; #a; #i; #f; #j; #x; #H; napply H;
nqed.
```

We assume the dual of the axiom of choice, as in the paper proof.

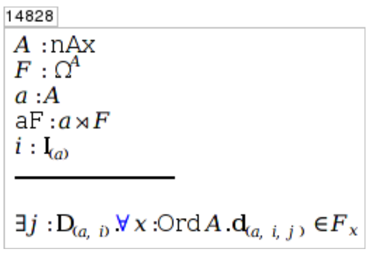
```
naxiom AC_dual: ∀A:nAx.∀a:A.∀i,F.
 (∀f:D a i → Ord A.∃x:D a i.d a i x ∈ F_(f x))
    → ∃j:D a i.∀x:Ord A.d a i j ∈ F_x.
```

Proving the anti-reflexivity property is simple, since the assumption `a ⋉ F` states that for every ordinal `x` (and thus also 0) `a ∈ F_x`. If `x` is choose to be `0`, we obtain the thesis.

```
ntheorem new_fish_antirefl: ∀A:nAx.∀F:Ω^A.∀a. a ⋉ F → a ∈ F.
#A; #F; #a; #H; nlapply (H 0); #aFo; napply aFo;
nqed.
```

We now prove the compatibility property for the new fish relation.

```
ntheorem new_fish_compatible:
 ∀A:nAx.∀F:Ω^A.∀a. a ⋉ F → ∀i:I a.∃j:D a i.d a i j ⋉ F.
#A; #F; #a; #aF; #i; nnormalize;
```



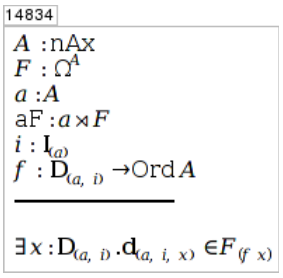
14828

$A : \mathrm{nAx}$
$F : \Omega^{A}$
$a : A$
$\mathrm{aF} : a \rtimes F$
$i : \mathbf{I}_{(a)}$

$\exists j : \mathbf{D}_{(a,\ i)} . \forall x : \mathrm{Ord}\, A . \mathbf{d}_{(a,\ i,\ j\ )} \in F_{x}$

After reducing to normal form the goal, we observe it is exactly the conclusion of the dual axiom of choice we just assumed. We thus apply it ad introduce the fcuntion `f`.

```
napply AC_dual; #f;
```

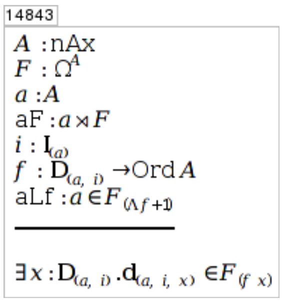

14834

$A : \mathrm{nAx}$
$F : \Omega^{A}$
$a : A$
$\mathrm{aF} : a \ltimes F$
$i : \mathbf{I}_{(a)}$
$f : \mathbf{D}_{(a,\ i)} \to \mathrm{Ord}\, A$

$\exists\, x : \mathbf{D}_{(a,\ i)} . \mathbf{d}_{(a,\ i,\ x)} \in F_{(f\ x)}$

The hypothesis `aF` states that `a⋉F_x` for every `x`, and we choose `Λf+1`.

```
nlapply (aF (Λf+1)); #aLf;
```

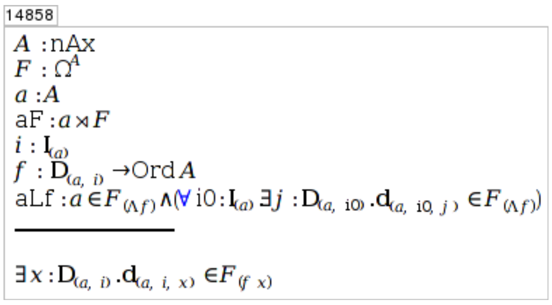

14843

$A : \mathrm{nAx}$
$F : \Omega^{A}$
$a : A$
$\mathrm{aF} : a \ltimes F$
$i : \mathbf{I}_{(a)}$
$f : \mathbf{D}_{(a,\ i)} \to \mathrm{Ord}\, A$
$\mathrm{aLf} : a \in F_{(\Lambda f+1)}$

$\exists\, x : \mathbf{D}_{(a,\ i)} . \mathbf{d}_{(a,\ i,\ x)} \in F_{(f\ x)}$

Since F_(Λf+1) is defined by recursion and we actually have a concrete input `Λf+1` for that recursive function, it can be computed. Anyway, using the `nnormalize` tactic would reduce too much (both the `+1` and the `Λf` steps would be performed); we thus explicitly give a convertible type for that hypothesis, corresponding the computation of the `+1` step, plus the unfolding the definition of the intersection.

```
nchange in aLf with
  (a ∈ F_(Λ f) ∧ ∀i:𝐈 a.∃j:𝐃 a i.𝐝 a i j ∈ F_(Λ f));
```

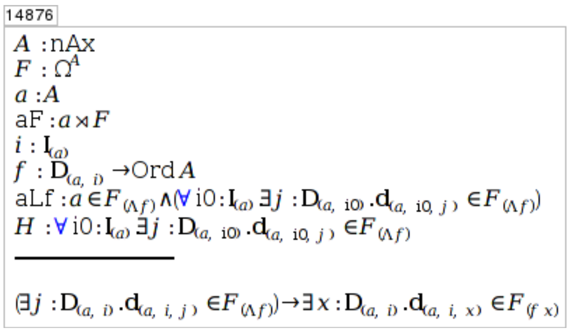

14858

$A : \mathrm{nAx}$
$F : \Omega^{A}$
$a : A$
$\mathrm{aF} : a \ltimes F$
$i : \mathbf{I}_{(a)}$
$f : \mathbf{D}_{(a,\ i)} \to \mathrm{Ord}\, A$
$\mathrm{aLf} : a \in F_{(\Lambda f)} \wedge (\forall\, \mathrm{i0} : \mathbf{I}_{(a)}\, \exists\, j : \mathbf{D}_{(a,\ \mathrm{i0})} . \mathbf{d}_{(a,\ \mathrm{i0},\ j)} \in F_{(\Lambda f)})$

$\exists\, x : \mathbf{D}_{(a,\ i)} . \mathbf{d}_{(a,\ i,\ x)} \in F_{(f\ x)}$

We are interested in the right hand side of `aLf`, an in particular to its intance where the generic index in `𝐈 a` is `i`.

```
ncases aLf; #_; #H; nlapply (H i);
```

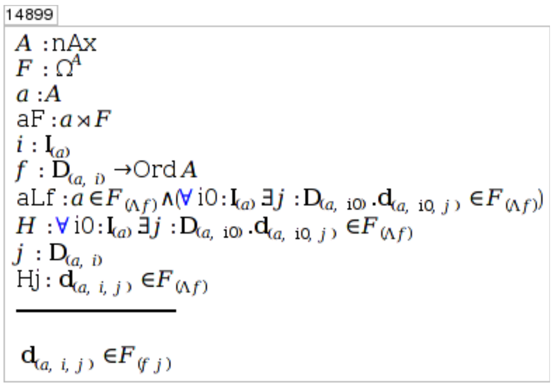

14876

$A : \mathrm{nAx}$
$F : \Omega^{A}$
$a : A$
$\mathrm{aF} : a \ltimes F$
$i : \mathbf{I}_{(a)}$
$f : \mathbf{D}_{(a,\ i)} \to \mathrm{Ord}\, A$
$\mathrm{aLf} : a \in F_{(\Lambda f)} \wedge (\forall\, \mathrm{i0} : \mathbf{I}_{(a)}\, \exists\, j : \mathbf{D}_{(a,\ \mathrm{i0})} . \mathbf{d}_{(a,\ \mathrm{i0},\ j)} \in F_{(\Lambda f)})$
$H : \forall\, \mathrm{i0} : \mathbf{I}_{(a)}\, \exists\, j : \mathbf{D}_{(a,\ \mathrm{i0})} . \mathbf{d}_{(a,\ \mathrm{i0},\ j)} \in F_{(\Lambda f)}$

$(\exists\, j : \mathbf{D}_{(a,\ i)} . \mathbf{d}_{(a,\ i,\ j)} \in F_{(\Lambda f)}) \to \exists\, x : \mathbf{D}_{(a,\ i)} . \mathbf{d}_{(a,\ i,\ x)} \in F_{(f\ x)}$

We then eliminate the existential, obtaining `j` and its property `Hj`. We provide the same witness

```
*; #j; #Hj; @j;
```



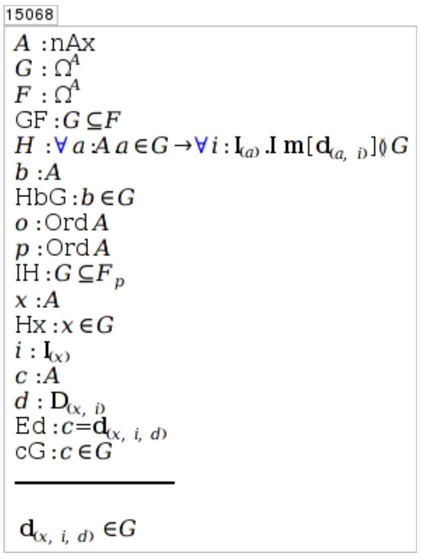
```
14899
A :nAx
F : Ω^A
a :A
aF : a ⋊ F
i : I⟨a⟩
f : D⟨a, i⟩ → Ord A
aLf : a ∈ F⟨⋀f⟩ ∧ (∀ i0 : I⟨a⟩ ∃j : D⟨a, i0⟩ . d⟨a, i0, j⟩ ∈ F⟨⋀f⟩)
H : ∀ i0 : I⟨a⟩ ∃j : D⟨a, i0⟩ . d⟨a, i0, j⟩ ∈ F⟨⋀f⟩
j : D⟨a, i⟩
Hj : d⟨a, i, j⟩ ∈ F⟨⋀f⟩
____________________

d⟨a, i, j⟩ ∈ F⟨f j⟩
```

What is left to prove is exactly the `co_ord_subset` lemma we factored out of the main proof.

```
napply (co_ord_subset … Hj);
nqed.
```

The proof that `⋉(F)` is maximal is exactly the dual one of the minimality of `◃(U)`. Thus the main problem is to obtain `G ⊆ F⎽o` before doing the induction over `o`.

```
ntheorem max_new_fished:
  ∀A:nAx.∀G:Ω^A.∀F:Ω^A.G ⊆ F → (∀a.a ∈ G → ∀i.Im[d a i] ≬ G) → G ⊆ ⋉F.
#A; #G; #F; #GF; #H; #b; #HbG; #o;
ngeneralize in match HbG; ngeneralize in match b;
nchange with (G ⊆ F⎽o);
nelim o;
##[ napply GF;
##| #p; #IH; napply (subseteq_intersection_r … IH);
    #x; #Hx; #i; ncases (H … Hx i); #c; *; *; #d; #Ed; #cG;
    @d; napply IH;
```



```
15068
A :nAx
G : Ω^A
F : Ω^A
GF : G ⊆ F
H : ∀ a :A a ∈ G → ∀ i : I⟨a⟩ .I m[d⟨a, i⟩] ≬ G
b :A
HbG : b ∈ G
o : Ord A
p : Ord A
IH : G ⊆ F_p
x :A
Hx : x ∈ G
i : I⟨x⟩
c :A
d : D⟨x, i⟩
Ed : c = d⟨x, i, d⟩
cG : c ∈ G
____________________

d⟨x, i, d⟩ ∈ G
```

Note that here the right hand side of `∈` is `G` and not `F⎽p` as in the dual proof. If `S` was declare to be a setoid, to finish this proof would be enough to assume `G` extensional, and no proof of the extensionality of `F⎽p` is required.

Appendix: tactics explanation

In this appendix we try to give a description of tactics in terms of sequent calculus rules annotated with proofs. The `:` separator has to be read as *is a proof of*, in the spirit of the Curry-Howard isomorphism.

```
                Γ ⊢  f  :  A_1 → … → A_n → B      Γ ⊢ ?_i  :  A_i
napply f;     ———————————————————————————————————————————————————————
                         Γ ⊢ (f ?_1 … ?_n )  :  B

                 Γ ⊢  ?  :  F → B        Γ ⊢ f  :  F
nlapply f;     ——————————————————————————————————————
                           Γ ⊢ (? f)  :  B


              Γ; x : T  ⊢ ?  :  P(x)
#x;       ——————————————————————————————
             Γ ⊢ λx:T.?  :  ∀x:T.P(x)


                     Γ ⊢ ?_i  :  args_i → P(k_i args_i)
ncases x;      ——————————————————————————————————————————————————————————————————
               Γ ⊢ match x with [ k1 args1 ⇒ ?_1 | … | kn argsn ⇒ ?_n ]  :  P(x)


                    Γ ⊢ ?i  :  ∀t. P(t) → P(k_i … t …)
nelim x;     —————————————————————————————————————————————————
                 Γ ⊢ (T_rect_CProp0 ?_1 … ?_n)  :  P(x)
```

Where `T_rect_CProp0` is the induction principle for the inductive type `T`.

```
                       Γ ⊢ ?  :  Q      Q ≡ P
nchange with Q;   ——————————————————————————————
                       Γ ⊢ ?  :  P
```

Where the equivalence relation between types `≡` keeps into account β-reduction, δ-reduction (definition unfolding), ζ-reduction (local definition unfolding), ι-reduction (pattern matching simplification), μ-reduction (recursive function computation) and ν-reduction (co-fixpoint computation).

```
                            Γ; H : Q; Δ ⊢ ?  :  G      Q ≡ P
nchange in H with Q; ————————————————————————————————————————————
                            Γ; H : P; Δ ⊢ ?  :  G


                            Γ; H : Q; Δ ⊢ ?  :  G      P →* Q
nnormalize in H; ————————————————————————————————————————————
                            Γ; H : P; Δ ⊢ ?  :  G
```

Where `Q` is the normal form of `P` considering βδζιμν-reduction steps.

```
                     Γ ⊢ ?  :  Q      P →* Q
nnormalize; ————————————————————————————————————————————
                     Γ ⊢ ?  :  P


                  Γ ⊢ ?_2  :  T → G     Γ ⊢ ?_1  :  T
ncut T;      ————————————————————————————————————————————
                            Γ ⊢ (?_2 ?_1)  :  G


                                   Γ ⊢ ?  :  ∀x.P(x)
ngeneralize in match t; ——————————————————————————
                                   Γ ⊢ (? t)  :  P(t)
```

```
    nrewrite < Ed; napply cG;
##| #a; #i; #f; #Hf; nchange with (G ⊆ { y | ∀x. y ∈ F_(f x) });
    #b; #Hb; #d; napply (Hf d); napply Hb;
##]
nqed.
```

Last updated: $Date: 2009-10-28 15:46:19 +0100 (Wed, 28 Oct 2009) $