Coq support for HoTT

Matthieu Sozeau
Inria Paris & PPS, Université Paris 7 Diderot

HoTT/UF Workshop
June 29th 2015
Warsaw, Poland
Embracing proof-relevant equality

Much work on DTT in Coq focused on propositional equality, assumed proof-irrelevant.

Two examples which rely deeply on equality:

- A generalized rewriting tactic
- A toolbox for handling definitions by dependent pattern-matching and well-founded recursion

Today: Adapting these tools to the new setting
1 Proof-relevant rewriting strategies
   ■ Generalized rewriting
   ■ Rewriting with Type-valued relations

2 Equations
   ■ Intro
   ■ Dependent pattern-matching compilation
   ■ Recursion
   ■ Reasoning support
Why generalized rewriting when we have \texttt{Id}-elimination?

- \texttt{Id} is not the only interesting relation...
- Even with a univalent equality, \texttt{Id}-elim is not enough: \textit{capturing} rewrites under binders.
Higher-order morphisms

\[
\text{Inductive } \text{ex} \ (\{A : \text{Type}\} \ (P : A \to \text{Prop}) : \text{Prop} := \\
\text{ex}_\text{intro} : \forall x : A, \ P x \to \text{ex} \ P. \\
\text{Instance } \text{ex}_\text{iff} P A : \\
\text{Proper} \ (\text{pointwise\_relation} \ A \iff \leftrightarrow \text{iff}) (\@ \text{ex} A).
\]
Inductive \( \text{ex} \ \{ A : \text{Type} \} \ (P : A \rightarrow \text{Prop}) : \text{Prop} := \)
\[
\text{ex\_intro} : \forall \ x : A, \ P \ x \rightarrow \text{ex} \ P.
\]

Instance \( \text{ex\_iff\_P} \ A : \)
\[
\text{Proper} \ (\text{pointwise\_relation} \ A \ \text{iff} \leftrightarrow \text{iff}) \ (@\text{ex} \ A).
\]

Goal \( \forall \ A \ P \ Q, \ (\forall \ x : A, \ P \ x \leftrightarrow Q \ x) \rightarrow \)
\[
(\exists \ x, \neg P \ x) \rightarrow (\exists \ x, \neg Q \ x).
\]

Proof.
\[
\text{intros} \ A \ P \ Q \ H \ HnP.
\text{setoid} \ \text{rewrite} \ \leftarrow H.
\text{exact} \ HnP.
\text{Qed}.
\]
Inductive \text{ex} \{ A : \text{Type} \} (P : A \to \text{Prop}) : \text{Prop} :=
\begin{align*}
\text{ex\_intro} : & \forall x : A, P \ x \ \to \ \text{ex} \ P. \\
\end{align*}

Instance \text{ex\_iff\_P} A :
\begin{align*}
\text{Proper} & \ (\text{pointwise\_relation} \ A \ \text{iff} \ \leftrightarrow \ \text{iff}) \ (@\text{ex} \ A).
\end{align*}

Goal \forall A \ P \ Q , (\forall x : A, P \ x \leftrightarrow Q \ x) \to
(\exists x, \neg P \ x) \to (\exists x, \neg Q \ x).

Proof. intros A P Q H HnP.

setoid\_rewrite \leftarrow H. exact HnP.
Qed.
Rewriting in Type Theory

Moving from substitution to congruence.

- Built-in substitution: Leibniz equality.
  \[ \Pi A (P : A \rightarrow \text{Type}) (x y : A), \ P x \rightarrow x = y \rightarrow P y. \]
  ✓ Applies to any context
  ✗ Iterated rewrites result in large proof terms: repeats the context that depends on \( x \)
  ✗ Restricted to equality, one rewrite at a time
Rewriting in Type Theory

Moving from substitution to congruence.

- **Built-in substitution: Leibniz equality.**

\[ \Pi A \ (P : A \rightarrow \text{Type}) \ (x \ y : A), \ P \ x \rightarrow x = y \rightarrow P \ y. \]

- Applies to any context
- Iterated rewrites result in large proof terms: repeats the context that depends on \(x\)
- Restricted to equality, one rewrite at a time

- **Congruence.**

\[ \text{ap} : \Pi A \ B \ (f : A \rightarrow B) \ (x \ y : A), \ x = y \rightarrow f \ x = f \ y \]

- Applies at the toplevel only
- Smaller proof term: mentions the changed terms only
- Generalizes to n-ary, parallel rewriting
- Still restricted to equality
Rewriting in Type Theory

Moving from substitution to congruence.

- **Built-in substitution: Leibniz equality.**
  \[ \Pi A \ (P : A \to \text{Type}) \ (x \ y : A), \ P \ x \to x = y \to P \ y. \]
  - ✓ Applies to any context
  - ✗ Iterated rewrites result in large proof terms: repeats the context that depends on \( x \)
  - ✗ Restricted to equality, one rewrite at a time

- **Congruence.**
  \[ \text{ap} : \Pi A \ B \ (f : A \to B) \ (x \ y : A), \ x = y \to f \ x = f \ y \]
  - ✗ Applies at the toplevel only
  - ✓ Smaller proof term: mentions the changed terms only
  - ✓ Generalizes to n-ary, parallel rewriting
  - ✗ Still restricted to equality

One can build a set of combinators to rewrite in depth: HOL conversions [Paulson 83], ELAN strategies.
Generalized Rewriting in Type Theory

D. Basin [NuPRL, 94], C. Sacerdoti Coen [CoQ, 04], Sozeau [CoQ, 09]

- Generalized to any relation
  \[ \text{Proper } (\text{iff } \leftrightarrow \text{ iff}) \not\equiv \Pi P Q, P \leftrightarrow Q \rightarrow \neg P \leftrightarrow \neg Q \]

- Multiple signatures for a given constant
  \[ \text{Proper } (\text{impl } \rightarrow \text{ impl}) \not\equiv \]
Generalized Rewriting in Type Theory

D. Basin [NuPRL, 94], C. Sacerdoti Coen [CoQ, 04], Sozeau [CoQ, 09]

- Generalized to **any** relation
  \[ \text{Proper (iff } \leftrightarrow \text{ iff)} \text{ not } \triangleq \Pi P Q, P \leftrightarrow Q \rightarrow \neg P \leftrightarrow \neg Q \]

- Multiple signatures for a given constant
  \[ \text{Proper (impl } \rightarrow \text{ impl)} \text{ not} \]

Requires **proof search**:

- Heuristic in NuPRL based on subrelations (impl \( \subseteq \) iff)
- Complete procedure in CoQ.
Our Algorithm

Two phases:

1. Constraint generation in ML.
   Recursive descent on the term to find the rewrites to perform, building a proof skeleton.

2. Constraint solving using resolution.
   Depth-first search to solve the constraints with the declared hints / typeclass instances.
The two main rules

\[ \Gamma \mid \psi \vdash \tau \xrightarrow{R} \tau' \vdash \psi' \]
The two main rules

\[
\Gamma \mid \psi \vdash \tau \rightsquigarrow^R_p \tau' - \psi'
\]

**UNIFY**

\[
\text{unify}_\rho(\Gamma, \psi, t) \uparrow \psi', \rho' : R\ t\ u
\]

\[
\Gamma \mid \psi \vdash t \rightsquigarrow^R_{\rho'} u - \psi'
\]
The two main rules

\[ \Gamma | \psi \vdash \tau \rightsquigarrow^R_p \tau' \vdash \psi' \]

Unify

\[
\text{unify}_\rho(\Gamma, \psi, t) \uparrow \psi', \rho' : R \ t \ u
\]

\[
\Gamma | \psi \vdash t \rightsquigarrow^R_{\rho'} u \vdash \psi'
\]

Atom

\[
\text{unify}_\rho^*(\Gamma, \psi, t) \downarrow \quad \tau \triangleq \text{type}(\Gamma, \psi, t)
\]

\[
\psi' \triangleq \{ ?_S : \Gamma \vdash \text{relation} \ \tau, ?_m : \Gamma \vdash \text{Proper} \ \tau \ ?_S t \}
\]

\[
\Gamma | \psi \vdash t \rightsquigarrow^{?_S}_{?_m} t \vdash \psi \cup \psi'
\]
Proper instances

- Extensible signatures (shallow embedding)
  
  \[
  \text{all} : \forall A : \text{Type}, (A \to \text{Prop}) \to \text{Prop} \\
  \Pi A, \text{Proper} (\text{pointwise\_relation} A \text{ iff } \leftrightarrow \text{iff}) (@\text{all} A)
  \]
Proper instances

- Extensible signatures (shallow embedding)
  \[ \forall A : \text{Type}, \ (A \rightarrow \text{Prop}) \rightarrow \text{Prop} \]
  \[ \Pi A, \ \text{Proper} \ (\text{pointwise\_relation} \ A \ i i f f \ \leftrightarrow \ i f f) \ (@\text{all} \ A) \]

- Algebraic presentation, supporting higher-order functions and polymorphism:
  \[ \Pi A \ B \ C \ R_0 \ R_1 \ R_2, \]
  \[ \text{Proper} \ ((R_1 \leftrightarrow R_2) \leftrightarrow (R_0 \leftrightarrow R_1) \leftrightarrow (R_0 \leftrightarrow R_2)) \]
  \[ (@\text{compose} \ A \ B \ C) \]
Proper instances

- Extensible signatures (shallow embedding)
  \[
  \text{all : } \forall A : \text{Type}, (A \to \text{Prop}) \to \text{Prop} \\
  \Pi A, \text{Proper} (\text{pointwise\_relation } A \iff \leftrightarrow \iff) (@all A)
  \]

- Algebraic presentation, supporting higher-order functions and polymorphism:
  \[
  \Pi A B C R_0 R_1 R_2, \\
  \text{Proper} ((R_1 \leftrightarrow R_2) \leftrightarrow (R_0 \leftrightarrow R_1) \leftrightarrow (R_0 \leftrightarrow R_2)) \\
  (@\text{compose } A B C)
  \]

- Generic morphism declarations.
Proper instances

- Extensible signatures (shallow embedding)
  \[ \forall A : \text{Type}, (A \to \text{Prop}) \to \text{Prop} \]
  \[ \Pi A, \text{Proper} \ (\text{pointwise\_relation} A \iff \leftrightarrow \iff) \ (@all A) \]

- Algebraic presentation, supporting higher-order functions and polymorphism:
  \[ \Pi A B C R_0 R_1 R_2, \]
  \[ \text{Proper} \ ((R_1 \leftrightarrow R_2) \leftrightarrow (R_0 \leftrightarrow R_1) \leftrightarrow (R_0 \leftrightarrow R_2)) \]
  \[ (@\text{compose} A B C) \]

- Generic morphism declarations.

- Subrelations, quotienting the signatures.
Proper instances

- Extensible signatures (shallow embedding)

\[ \forall A : \text{Type}, (A \to \text{Prop}) \to \text{Prop} \]

\[ \Pi A, \text{Proper} \left( \text{pointwise\_relation} \ A \iff \leftrightarrow \iff \right) (@\text{all} \ A) \]

- Algebraic presentation, supporting higher-order functions and polymorphism:

\[ \Pi A B C R_0 R_1 R_2, \]

\[ \text{Proper} \left( (R_1 \leftrightarrow R_2) \leftrightarrow (R_0 \leftrightarrow R_1) \leftrightarrow (R_0 \leftrightarrow R_2) \right) \]

\[ (@\text{compose} \ A \ B \ C) \]

- Generic morphism declarations.

- Subrelations, quotienting the signatures.

- Rewriting on operators/functions, parallel rewrites...
1. Proof-relevant rewriting strategies
   - Generalized rewriting
   - Rewriting with Type-valued relations

2. Equations
   - Intro
   - Dependent pattern-matching compilation
   - Recursion
   - Reasoning support
     - Equations
     - Elimination principle
     - Eliminating calls
All fine with relations in Prop, how about Type-valued relations?

**Proper**: $\Pi A : \text{Type}_i, (A \to A \to \text{Type}_j) \to A \to \text{Type}_j$.

Need to show, under $A : \text{Type}_i$:

**Proper** $((A \to A \to \text{Type}_j) \to A \to \text{Type}_j)$

(iso$_\text{rel}$ $A \to$ eq $A \to$ iso)

(Proper $A$)

Requires: $\text{Type}_{\text{max}(i,j+1)} \leq \text{Type}_i$ i.e. $j < i$.

But then $\text{iso} A : \text{Type}_i \not\leq \text{Type}_j \Rightarrow$ inconsistency.
With universe polymorphism (Sozeau & Tabareau [ITP’14]):

$$\text{Proper}_{i,j} : \Pi A : \text{Type}_i, (A \to A \to \text{Type}_j) \to A \to \text{Type}_j$$

We can show, under $$A : \text{Type}_i$$:

$$\text{Proper}_{i',j'} \quad ((A \to A \to \text{Type}_j) \to A \to \text{Type}_j)$$

$$(\text{iso}_\text{rel} A \to \text{eq} A \to \text{iso})$$

$$(\text{Proper}_{i,j} A)$$

The constraint $$\max(i, j + 1) \leq i'$$ is satisfiable.

Actually, $$\text{crelation}(A : \text{Type}_i) := A \to A \to \text{Type}_j$$ is already problematic: no relation equivalence or subrelation definition possible.
Generalized rewriting will now handle:

- The general function space morphism between types.
- Type-level identity, isomorphisms and equivalences of types.
- Computationally relevant relations like CoRN’s appartness relation on reals.
- Hom-types of categories which are not $\text{Prop}$-based setoids, e.g. groupoids...
User-definable strategies: rewrite_strat.

- A tactic/strategy language: bottom-up, innermost, with composition, disjunction, rewrite hint databases...
- Much faster than autorewrite.
- More control on the shape of proof terms.
Suppose the theory of monoids on $T$.
A goal: $x\ y : T \vdash x \cdot ((\epsilon \cdot y) \cdot \epsilon)$.

- **autorewrite with monoids** will do two rewrites with both unit laws, the proof term will be roughly twice the goal size.
- **rewrite_strat (topdown (repeat (hints monoids)))** will first rewrite $\epsilon \cdot y$ to $y$ and directly after, $y \cdot \epsilon$ to $y$, resulting in a proof term of size roughly that of the initial goal, and will be twice as fast as well.
1. **Proof-relevant rewriting strategies**
   - Generalized rewriting
   - Rewriting with Type-valued relations

2. **Equations**
   - Intro
   - Dependent pattern-matching compilation
   - Recursion
   - Reasoning support
     - Equations
     - Elimination principle
     - Eliminating calls
A word of caution:

- **Strong elimination:**
  ```coq
definition match x return if x then unit else nat with |
  true => tt |
  false => 0 |
end
```

⇒ *intensional* character of inductive values.

- **Dependent Pattern-Matching:**
  ```coq
definition match (v : vector bool (S 0)) return bool with |
  Vcons a ?(0) v => a |
end
```

⇒ extends to the *propositional* equational theory of inductive types.
Dependent Pattern-Matching and Equality

1992 Pattern Matching With Dependent Types – Coquand
1999 Dependently Typed Functional Programs and Their Proofs – McBride
2004 The View From The Left – McBride and McKinna
≈ 2010 GADTs in the programming languages community.
2014 Pattern Matching without K – Cockx, Devriese and Piessens
http://github.com/mattam82/Coq-Equations
(opam package coq:equations)

- **Agda/Epigram**-style definitions (including `with`)
- Purely logical handling of recursion.
- Propositional equations for definitional equalities and rewriting.
- Function graph and elimination principle derivation (w/ support for applying it).

Entirely elaborated to the vanilla kernel!
DEMO
Elaboration into \( \text{CIC} + \text{K} \) (as an axiom or a user-provided proof)

1. Generation of a splitting tree from the clauses

Elaboration into $\text{CIC} + K$ (as an axiom or a user-provided proof)

1. Generation of a splitting tree from the clauses
2. Translation from the splitting tree to $\text{Coq}$ terms with holes
Elaboration into $\text{CIC} + K$ (as an axiom or a user-provided proof)

1. Generation of a splitting tree from the clauses
2. Translation from the splitting tree to $\text{Coq}$ terms with holes
3. Proofs of the obligations using a mix of $\text{ML}$ and $\mathcal{L}_\text{tac}$ code

Elaboration into CIC + K (as an axiom or a user-provided proof)

1. Generation of a splitting tree from the clauses
2. Translation from the splitting tree to COQ terms with holes
3. Proofs of the obligations using a mix of ML and \( L_{\text{tac}} \) code
4. Derivation of auxiliary structures from the completed splitting tree

Searching for a splitting tree

pattern $p ::= x | C \overrightarrow{p} | ?(t)$
context map $c ::= \Delta \vdash \overrightarrow{p} : \Gamma$
splitting $spl ::= \text{Split}(c, x, (spl?)^n) | \text{Compute}(c, rhs)$
node $rhs ::= \text{Program}(t) | \text{Refine}(c, t, spl)$

Goal
Starting with $f \Delta : \tau ::= \overrightarrow{p} \ldots$, find a covering of the context map $\Delta \vdash \overrightarrow{\Delta} : \Delta$ by $\overrightarrow{p}$.
Proof search example

Overlapping clauses with first-match semantics.

Equations equal (n m : nat) : { n = m } + { n ≠ m } :=
equal O O := left eq_refl ;
equal (S n) (S m) with equal n m := {
equal (S n) (S ?(n)) (left eq_refl) := left eq_refl ;
equal (S n) (S m) (right p) := right _ } ;
equal x y := right _ .

Split(n m : nat ⊢ n m : n m : nat, n, [ Split(m : nat ⊢ O m : n m : nat, m, [
  Compute(⊢ O O : n m : nat, Program(left eq_refl)),
  Compute(m : nat ⊢ O (S m) : n m : nat, Program(right _))])],
Split(n m : nat ⊢ (S n) m : n m : nat, m, [
  Compute(n : nat ⊢ (S n) O : n m : nat, . . .),
  Compute(n m : nat ⊢ (S n) (S m) : n m : nat,
    Refine(equal n m,
      idsubst(n m : nat, x : {n = m} + {n ≠ m}), l) . . .))))
For each node $\Delta \vdash ps : \Gamma \rightsquigarrow \Pi \Delta$, $f_{\text{comp}} ps$. 

Translation from the splitting to Coq
For each node $\Delta \vdash ps : \Gamma \rightsquigarrow \Pi \Delta, f_{\text{comp}} ps$.

- Split($c, x, s$): witnessed by dependent elimination. dependent destruction, using JMeq or user-given K/hSet proofs.
For each node $\Delta \vdash ps : \Gamma \rightsquigarrow \Pi \Delta, f_{\text{comp}} ps$.

- Split$(c, x, s)$: witnessed by dependent elimination. Dependent destruction, using $\text{JMeq}$ or user-given $K/\text{hSet}$ proofs.
- Program$(t)$: witnessed by $t$ (w/ some substitution).
For each node $\Delta \vdash ps : \Gamma \rightsquigarrow \Pi \Delta, f_{\text{comp}} ps$.

- **Split**$(c, x, s)$: witnessed by dependent elimination.
  dependent destruction, using JMeq or user-given $K/h\text{Set}$
  proofs.
- **Program**$(t)$: witnessed by $t$ (w/ some substitution).
- **Refine**$(t, c, s)$: witnessed by:
  1. inserting a let-definition in the context,
  2. strengthening it,
  3. abstracting it and clearing its body,
  4. applying the compiled term for the subprogram with one
     additional variable.
1 Proof-relevant rewriting strategies
  ■ Generalized rewriting
  ■ Rewriting with Type-valued relations

2 Equations
  ■ Intro
  ■ Dependent pattern-matching compilation
  ■ Recursion
  ■ Reasoning support
    ■ Equations
    ■ Elimination principle
    ■ Eliminating calls
Recursion

- Syntactic guardness checks are fragile (and buggy)
- Incompatible with abstraction/modularity
- In Coq’s case, restricted to structural recursion on a single argument

Idea Use the logic and well-founded recursion instead!
Syntactic guardness checks are fragile (and buggy)
Incompatible with abstraction/modularity
In Coq’s case, restricted to structural recursion on a single argument

Idea Use the logic and well-founded recursion instead!

In comparison with sized types (e.g. Agda’s size annotations):

- More general.
- Avoid extending the type system and the metatheory.
- Relies on the reduction of a well-foundedness proof, necessary for SN. In turn, relies on *logical* information on the *computational* behavior to be available.
Well-founded recursion on the subterm relation for inductive families $I : \Pi \Delta, \text{Type}$.
Well-founded recursion on the subterm relation for inductive families $I : \Pi \Delta, \text{Type}$.

- General definition of direct subterm:
  \[
  I_{\text{subfull}} : \Pi \Delta_l \Delta_r, I \Delta_l \rightarrow I \Delta_r \rightarrow \text{Prop}
  \]

Moving to \text{Type} is easy (though $I_{\text{subfull}}$ is not always a proposition).
Well-founded recursion on the subterm relation for inductive families $I : \Pi \Delta, \text{Type}$.

- General definition of direct subterm:
  $$I_{sub full} : \Pi \Delta_l \Delta_r, I \Delta_l \rightarrow I \Delta_r \rightarrow \text{Prop}$$

- Wrap the inductive type in a sigma and define an homogeneous relation on the sigma type:
  $$I_{sub} : \text{relation} (\Sigma \Delta, I \Delta)$$

Moving to $\text{Type}$ is easy (though $I_{sub full}$ is not always a proposition).
Well-founded recursion on the subterm relation for inductive families $\Delta : \Pi \Delta, \text{Type}$.

- General definition of direct subterm:
  \[ l_{\text{subfull}} : \Pi \Delta_l \Delta_r, l \Delta_l \rightarrow l \Delta_r \rightarrow \text{Prop} \]

- Wrap the inductive type in a sigma and define an homogeneous relation on the sigma type:
  \[ l_{\text{sub}} : \text{relation} (\Sigma \Delta, l \Delta) \]

- Extracts efficiently to a general fixpoint (assuming accessibility is defined in $\text{Prop}$).

Moving to $\text{Type}$ is easy (though $l_{\text{subfull}}$ is not always a proposition).
Example: vectors

Derive Signature for vector.
Derive Subterm for vector.
Derive Signature for vector.
Derive Subterm for vector.

Inductive vector_direct_subterm (A : Type)
  : ∀ n n' : nat, vector A n → vector A n' → Prop :=
  vector_direct_subterm_1_1 : ∀ (h : A) (n : nat) (v : vector A n),
                           vector_direct_subterm A n (S n) v (Vcons h v)

Check vector_subterm : ∀ A : Type, relation {n : nat & vector A n}.
Example: vectors

*Derive Signature for vector.*
*Derive Subterm for vector.*

**Inductive** `vector_direct_subterm (A : Type)`

: `∀ n n' : nat, vector A n → vector A n' → Prop :=
  vector_direct_subterm_1_1 : ∀ (h : A) (n : nat) (v : vector A n),
  vector_direct_subterm A n (S n) v (Vcons h v)

**Check vector_subterm : ∀ A : Type, relation \{n : nat & vector A n\}**.

**Equations unzip \{A B\} \{n\} (v : vector (A × B) n) : vector A n × vector B n :=
  unzip A B n v by rec v :=
  unzip A B ?(O) nil := (nil, nil) ;
  unzip A B ?(S n) (cons (pair x y) n v) with unzip v := {
    | (pair xs ys) := (cons x xs, cons y ys) }

Using dependent elimination on \textit{decidable} indices.

\textbf{Equations} unzip\_dec \{A \ B\} \{'\{\text{EqDec A}\} \{'\{\text{EqDec B}\} \}
\{n\} (v : \text{vector} \ (A \times B) \ n) : \text{vector} \ A \ n \times \text{vector} \ B \ n :=
\text{unzip\_dec} \ A \ B \ _ \ _ \ n \ v \ \text{by rec} \ v :=
\text{unzip\_dec} \ A \ B \ _ \ _ \ ?(O) \ \text{nil} := (\text{nil, nil});
\text{unzip\_dec} \ A \ B \ _ \ _ \ ?(S \ n) \ (\text{cons} \ (\text{pair} \ x \ y) \ n \ v) \ \text{with unzip\_dec} \ v := \{\}
| \text{pair} \ xs \ ys := (\text{cons} \ x \ xs, \text{cons} \ y \ ys) \}.

\textit{Print Assumptions} unzip\_dec.

\textbf{Closed under the global context}
1 Proof-relevant rewriting strategies
   - Generalized rewriting
   - Rewriting with Type-valued relations

2 Equations
   - Intro
   - Dependent pattern-matching compilation
   - Recursion
   - Reasoning support
     - Equations
     - Elimination principle
     - Eliminating calls
Equations derived from the splitting tree hold definitionally in CCI (assuming no use of $K$).

Equations for with nodes are just proxies to helper functions.

All put together in a rewrite database, $f$ can be opacified.

For well-founded definitions, fixpoint unfolding lemma. This requires funext and showing that accessibility is an hProp.
Equations filter \{A\} (l : list A) (p : A \to bool) : list A :=
filter A nil p := nil ;
filter A (cons a l) p with p a := {
  | true := a :: filter l p ;
  | false := filter l p }.

Matthieu Sozeau - Coq support for HoTT
Generated mutual induction principle

\[\text{Check}(\text{filter\_ind\_mut} : \forall (P : \forall (A : \text{Type}) (l : \text{list} A) (p : A \rightarrow \text{bool}), \text{filter\_comp} l p \rightarrow \text{Prop}) (P0 : \forall (A : \text{Type}) (a : A) (l : \text{list} A) (p : A \rightarrow \text{bool}), \text{bool} \rightarrow \text{filter\_comp} (a :: l) p \rightarrow \text{Prop}), \]

\[(\forall A p, P A [] p []) \rightarrow \]

\[(\forall A a l p, \text{filter\_ind\_1} A a l p (p a) (\text{filter\_obligation\_2} (@\text{filter}) A a l p (p a)) \rightarrow P0 A a l p (p a) (\text{filter\_obligation\_2} (@\text{filter}) A a l p (p a)) \rightarrow P A (a :: l) p (\text{filter\_obligation\_2} (@\text{filter}) A a l p (p a))) \rightarrow \]

\[(\forall A a l p, \text{filter\_ind} A l p (\text{filter} l p) \rightarrow P A l p (\text{filter} l p) \rightarrow P0 A a l p \text{true} (a :: \text{filter} l p)) \rightarrow (\forall A a l p, \text{filter\_ind} A l p (\text{filter} l p) \rightarrow P A l p (\text{filter} l p) \rightarrow P0 A a l p \text{false} (\text{filter} l p)) \rightarrow \]

\[\forall A l p (f3 : \text{filter\_comp} l p), \text{filter\_ind} A l p f3 \rightarrow P A l p f3).\]
We prove `filter` respects the generated graph and derive:

Check (filter_elim :
\[ \forall P : \forall (A : \text{Type}) (l : \text{list } A) (p : A \to \text{bool}), \, \text{filter\_comp } l \, p \to \text{Prop}, \]
let \( P0 \) := \text{fun } (A : \text{Type}) (a : A) (l : \text{list } A) (p : A \to \text{bool})
\[ \text{(refine : bool)} (H : \text{filter\_comp } (a :: l) \, p) \Rightarrow \]
\[ p \, a = \text{refine} \to P \, A \, (a :: l) \, p \, H \]
in
\[ (\forall (A : \text{Type}) (p : A \to \text{bool}), \, P \, A \, [] \, p \, []) \to \]
\[ (\forall (A : \text{Type}) (a : A) (l : \text{list } A) (p : A \to \text{bool}), \]
\[ P \, A \, l \, p \, (\text{filter } l \, p) \to P0 \, A \, a \, l \, p \, \text{true} \, (a :: \text{filter } l \, p)) \to \]
\[ (\forall (A : \text{Type}) (a : A) (l : \text{list } A) (p : A \to \text{bool}), \]
\[ P \, A \, l \, p \, (\text{filter } l \, p) \to P0 \, A \, a \, l \, p \, \text{false} \, (\text{filter } l \, p)) \to \]
\[ \forall (A : \text{Type}) (l : \text{list } A) (p : A \to \text{bool}), \, P \, A \, l \, p \, (\text{filter } l \, p)). \]
The elimination principle can only be applied usefully to calls with \textit{solely} variable arguments.

\[
\Pi A \ (l : \text{list } A), \ \text{app} \ l \ [] = l
\]
Eliminating calls

The elimination principle can only be applied usefully to calls with \textit{solely} variable arguments.

\[
\Pi A \ (l : \text{list} \ A), \ \text{app} \ l \ [] = l
\]

Using the “abstraction by equalities” technique again, we can abstract:

\[
(\lambda (l \ l' : \text{list} \ A) \ (r : \text{app}_{\text{comp}} \ l \ l'), \l' = [] \rightarrow r = \text{app} \ l \ [] \rightarrow \text{app} \ l \ [] = l) \quad l \ [] \ (\text{app} \ l \ [])
\]

Directly apply the elimination principle and simplify the equations.
A function definition package handling:

- Full, nested dependent pattern-matching
- Structural and well-founded recursion on dependent types
- Generation of useful support lemmas for reasoning a posteriori
- No axioms if you provide the right proofs.

Benchmarked on a bit-fiddling library and a proof of (relative) consistency for Predicative System F (de Bruijn style, Mangin & Sozeau [LFMTP’15]).
We moved to dependent equality (i.e. equality in sigma types) instead of \texttt{JMeq}. This is necessary to use \texttt{hProp/hSet} hypotheses (\texttt{JMeq} requires $K$ on type equalities).

Efficiency of computation? Consequences of moving to type-valued equality.
Perspectives

- Non-constructor indices and unsolved constraints, e.g.:
  \[ 0 = x + y, \] with a subsequent splitting on \( x \).
- Support for views/arbitrary eliminators (e.g. McBride’s \textit{by}).
- Structural and well-founded mutual recursion.
- Better support for the encode-decode method?
- HITs: low-level and high-level syntax for pattern-matching and solving higher equality obligations (Barras & Mangin).
Thanks!
Consider a current problem $\Delta \vdash \overline{\rho} : \Gamma$ and a user clause $f \overline{\rho}$ with $t_{pre} := \{ e \}$ matching it. We typecheck $t_{pre}$ into $t : \tau$ and use strengthening and abstraction to find a new context

$$\Delta^t, x_t : \tau, \Delta_t[t/x_t]$$

such that $\Delta^t, \Delta_t \sim \Delta$
Consider a current problem $\Delta \vdash \overrightarrow{p} : \Gamma$ and a user clause $f \overrightarrow{w} \text{ with } t_{\text{pre}} := \{ e \}$ matching it. We typecheck $t_{\text{pre}}$ into $t : \tau$ and use strengthening and abstraction to find a new context

$$\Delta^t, x_t : \tau, \Delta_t[t/x_t]$$

such that $\Delta^t, \Delta_t \sim \Delta$

Using the clauses $e$ we then build a subcovering $s$ of the identity context map

$$c = \text{idsubst}(\Delta^t, x_t : \tau_\Delta, \Delta_t[t/x_t])$$

and return $\text{Refine}(t, c, s)$. 
Consider a current problem \( \Delta \vdash \overline{p} : \Gamma \) and a user clause \( f \overline{u} \) with \( t_{pre} := \{ e \} \) matching it. We typecheck \( t_{pre} \) into \( t : \tau \) and use strengthening and abstraction to find a new context \( \Delta, x : \tau, \Delta[t/x] \) such that \( \Delta' = \Delta[t/x] \).

Using the clauses \( e \) we then build a subcovering \( s \) of the identity context map

\[
c = \text{idsubst}(\Delta, x : \tau, \Delta[t/x])
\]

and return \( \text{Refine}(t, c, s) \).

Compilation produces

\[
\ell.n : \Pi \Delta \Delta_0 (x,t : \tau_\Delta) \Delta[t/x], (f_{\text{comp}} \overline{p})[t/x], \text{we build}
\]

\[
(\lambda \Delta, \ell.n \overline{\Delta} t \overline{\Delta} : \Pi \Delta, f_{\text{comp}} \overline{p})
\]
Elimination principle: inductive graph

For \( f \cdot \ell : \Pi \Delta, f_{\text{comp}} \xrightarrow{t} \) we generate \( f \cdot \ell_{\text{ind}} : \Pi \Delta, f_{\text{comp}} \xrightarrow{t} \rightarrow \text{Prop} \) and prove \( \Pi \Delta, f \cdot \ell_{\text{ind}} \Delta (f \cdot \ell \Delta) \).

\text{AbsRec}(f, t)\) abstracts all the calls to \( f_{\text{comp-proj}} \) from the term \( t \), returning a new derivation \( \Gamma' \vdash t' \) where \( \Gamma' \) contains bindings of the form \( x : \Pi \Delta, f_{\text{comp}} \xrightarrow{t} \) for all the recursive calls.

Define \( \text{HypS}(\Gamma) \) by a map to produce the corresponding inductive hyps of the form \( H_x : \Pi \Delta, f_{\text{ind}} \xrightarrow{t} (x \Delta) \).
Inductive graph constructors

Direct translation from the splitting tree:

- **Split** \((c, x, s)\), **Rec** \((v, s)\) : collect the constructors for the subsplitting(s) \(s\), if any.
- **Compute** \((\Delta \vdash \overrightarrow{p} : \Gamma, rhs)\) : By case on \(rhs\):
  - **Program** \((t)\) : Compute \(\Psi \vdash t' = \text{ABSREC}(f, t)\) and return the statement
    \[
    \Pi \Delta \Psi \text{HYPS}(\Psi), \ f.\ell_{\text{ind}} \overrightarrow{p} \ t'
    \]
  - **Refine** \((t, \Delta' \vdash \overrightarrow{v}^x, x, \overrightarrow{v}_x : \Delta^x, x : \tau, \Delta_x, \ell.n)\) : 
    Compute \(\Psi \vdash t' = \text{ABSREC}(f, t)\) and return:
    \[
    \Pi \Delta \Psi \text{HYPS}(\Psi) (\text{res} : f_{\text{comp}} \overrightarrow{p})
    \]
    \[
    f.\ell.n_{\text{ind}} \Delta^x t' \Delta_x \text{res} \rightarrow f.\ell_{\text{ind}} \overrightarrow{p} \text{res}
    \]
    We continue with the generation of the \(f.\ell.n_{\text{ind}}\) graph.