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

Coq support for HoTT

1. Proof-relevant rewriting strategies
   - Generalized rewriting
   - Rewriting with Type-valued relations

2. Equations
   - Intro
   - Dependent pattern-matching compilation
   - Recursion
   - Reasoning support

Generalized Rewriting

Why generalized rewriting when we have Id-elimination?

- Id is not the only interesting relation...
- Even with a univalent equality, Id-elim is not enough: *capturing* rewrites under binders.

Higher-order morphisms

```
Inductive ex {A : Type} (P : A → Prop) : Prop :=
    ex_intro : ∀ x : A, P x → ex P.

Instance ex_iff_P A :
  Proper (pointwise_relation A iff ++> iff) (@ex A).
```

Higher-order morphisms

```
Inductive ex {A : Type} (P : A → Prop) : Prop :=
    ex_intro : ∀ x : A, P x → ex P.

Instance ex_iff_P A :
  Proper (pointwise_relation A iff ++> iff) (@ex A).
```

```
Goal ∀ A P Q, (∀ x : A, P x ↔ Q x) →
  (∃ x, ¬ P x) → (∃ x, ¬ Q x).
```

Higher-order morphisms

```
Inductive ex {A : Type} (P : A → Prop) : Prop :=
    ex_intro : ∀ x : A, P x → ex P.

Instance ex_iff_P A :
  Proper (pointwise_relation A iff ++> iff) (@ex A).
```

```
Goal ∀ A P Q, (∀ x : A, P x ↔ Q x) →
  (∃ x, ¬ P x) → (∃ x, ¬ Q x).
Proof. intros A P Q H HnP.
  setoid_rewrite ← H. exact HnP.
Qed.
```

Rewriting in Type Theory

Moving from substitution to congruence.

- Built-in substitution: Leibniz equality.
  $\Pi\ A\ (P : A \to \texttt{Type})\ (x\ y : A),\ P\ x \to x = y \to P\ y.$
  - ✓ 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

Rewriting in Type Theory

Moving from substitution to congruence.

- Built-in substitution: Leibniz equality.
  $\Pi\ A\ (P : A \to \mathtt{Type})\ (x\ y : A),\ P\ x \to x = y \to P\ y.$
  - ✓ 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.
  $\mathrm{ap} : \Pi\ A\ B\ (f : A \to B)\ (x\ y : A),\ x = y \to f\ x = f\ y$
  - ✗ Applies at the toplevel only
  - ✓ Smaller proof term: mentions the changed terms only
  - ✓ Generalizes to n-ary, parallel rewriting
  - ✗ Still restricted to equality

Rewriting in Type Theory

Moving from substitution to congruence.

- Built-in substitution: Leibniz equality.
  $\Pi\ A\ (P : A \to \mathtt{Type})\ (x\ y : A),\ P\ x \to x = y \to P\ y.$
  - ✓ 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.
  $\mathrm{ap} : \Pi\ A\ B\ (f : A \to B)\ (x\ y : A),\ x = y \to f\ x = f\ y$
  - ✗ 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
  Proper (iff ++> iff) not $\triangleq \Pi\ P\ Q,\ P \leftrightarrow Q \rightarrow \neg\ P \leftrightarrow \neg\ Q$
- Multiple signatures for a given constant
  Proper (impl $\longrightarrow$ impl) not

Generalized Rewriting in Type Theory

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

- Generalized to any relation
  Proper (iff ++> iff) not $\triangleq \Pi\ P\ Q,\ P \leftrightarrow Q \rightarrow \neg\ P \leftrightarrow \neg\ Q$
- Multiple signatures for a given constant
  Proper (impl $\longrightarrow$ impl) not

Requires proof search:

- Heuristic in NuPRL based on subrelations (impl $\subset$ 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

$$\boxed{\Gamma \mid \psi \vdash \tau \rightsquigarrow_p^R \tau' \dashv \psi'}$$

The two main rules

$$\boxed{\Gamma \mid \psi \vdash \tau \rightsquigarrow^{R}_{p} \tau' \dashv \psi'}$$

UNIFY

$$\frac{\mathbf{unify}_{\rho}(\Gamma, \psi, t) \Uparrow \psi', \rho' : R\ t\ u}{\Gamma \mid \psi \vdash t \rightsquigarrow^{R}_{\rho'} u \dashv \psi'}$$

The two main rules

$$\boxed{\Gamma \mid \psi \vdash \tau \rightsquigarrow^{R}_{p} \tau' \dashv \psi'}$$

UNIFY

$$\frac{\textbf{unify}_{\rho}(\Gamma, \psi, t) \Uparrow \psi', \rho' : R\ t\ u}{\Gamma \mid \psi \vdash t \rightsquigarrow^{R}_{\rho'} u \dashv \psi'}$$

ATOM

$$\frac{\begin{array}{l}\textbf{unify}^{*}_{\rho}(\Gamma, \psi, t) \Downarrow \quad \tau \triangleq \textbf{type}(\Gamma, \psi, t) \\ \psi' \triangleq \{?_S : \Gamma \vdash \text{relation}\ \tau, ?_m : \Gamma \vdash \text{Proper}\ \tau\ ?_S\ t\}\end{array}}{\Gamma \mid \psi \vdash t \rightsquigarrow^{?_S}_{?_m} t \dashv \psi \cup \psi'}$$

Proper instances

- Extensible signatures (shallow embedding)

  $\mathrm{all} : \forall\ A : \mathtt{Type},\ (A \to \mathtt{Prop}) \to \mathtt{Prop}$

  $\Pi\ A,\ \mathrm{Proper}\ (\mathrm{pointwise\_relation}\ A\ \mathrm{iff} \Rightarrow \mathrm{iff})\ (@\mathrm{all}\ A)$

Proper instances

- Extensible signatures (shallow embedding)

  $\mathrm{all} : \forall\ A : \mathtt{Type},\ (A \to \mathtt{Prop}) \to \mathtt{Prop}$

  $\Pi\ A,\ \mathrm{Proper}\ (\mathrm{pointwise\_relation}\ A\ \mathrm{iff} \Rightarrow \mathrm{iff})\ (@\mathrm{all}\ A)$

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

  $\Pi\ A\ B\ C\ R_0\ R_1\ R_2,$
  $\mathrm{Proper}\ ((R_1 \Rightarrow R_2) \Rightarrow (R_0 \Rightarrow R_1) \Rightarrow (R_0 \Rightarrow R_2))$
  $\quad (@\mathrm{compose}\ A\ B\ C)$

Proper instances

- Extensible signatures (shallow embedding)

  $\mathrm{all} : \forall\ A : \mathtt{Type},\ (A \to \mathtt{Prop}) \to \mathtt{Prop}$

  $\Pi\ A,\ \mathrm{Proper}\ (\mathrm{pointwise\_relation}\ A\ \mathrm{iff} \mathrel{++>} \mathrm{iff})\ (@\mathrm{all}\ A)$

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

  $\Pi\ A\ B\ C\ R_0\ R_1\ R_2,$
  $\mathrm{Proper}\ ((R_1 \mathrel{++>} R_2) \mathrel{++>} (R_0 \mathrel{++>} R_1) \mathrel{++>} (R_0 \mathrel{++>} R_2))$
  $\quad (@\mathrm{compose}\ A\ B\ C)$

- Generic morphism declarations.

Proper instances

- Extensible signatures (shallow embedding)

  $\mathrm{all} : \forall\ A : \mathtt{Type},\ (A \to \mathtt{Prop}) \to \mathtt{Prop}$

  $\Pi\ A,\ \mathrm{Proper}\ (\mathrm{pointwise\_relation}\ A\ \mathrm{iff} \mathrel{++>} \mathrm{iff})\ (@\mathrm{all}\ A)$

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

  $\Pi\ A\ B\ C\ R_0\ R_1\ R_2,$
  $\mathrm{Proper}\ ((R_1 \mathrel{++>} R_2) \mathrel{++>} (R_0 \mathrel{++>} R_1) \mathrel{++>} (R_0 \mathrel{++>} R_2))$
  $(@\mathrm{compose}\ A\ B\ C)$

- Generic morphism declarations.
- Subrelations, quotienting the signatures.

Proper instances

- Extensible signatures (shallow embedding)

  $\mathrm{all} : \forall\ A : \mathtt{Type},\ (A \to \mathtt{Prop}) \to \mathtt{Prop}$

  $\Pi\ A,\ \mathrm{Proper}\ (\mathrm{pointwise\_relation}\ A\ \mathrm{iff} \mathrel{++\!>} \mathrm{iff})\ (@\mathrm{all}\ A)$

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

  $\Pi\ A\ B\ C\ R_0\ R_1\ R_2,$
  $\mathrm{Proper}\ ((R_1 \mathrel{++\!>} R_2) \mathrel{++\!>} (R_0 \mathrel{++\!>} R_1) \mathrel{++\!>} (R_0 \mathrel{++\!>} R_2))$
  $(@\mathrm{compose}\ A\ B\ C)$

- Generic morphism declarations.
- Subrelations, quotienting the signatures.
- Rewriting on operators/functions, parallel rewrites...

Coq support for HoTT

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

Impredicativity helped

All fine with relations in `Prop`, how about `Type`-valued relations?

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

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

$$\begin{array}{ll} \mathsf{Proper} & ((A \to A \to \mathtt{Type}_j) \to A \to \mathtt{Type}_j) \\ & (\mathrm{iso\_rel}\ A \longrightarrow \mathrm{eq}\ A \longrightarrow \mathrm{iso}) \\ & (\mathsf{Proper}\ A) \end{array}$$

Requires: $\mathtt{Type}_{\mathsf{max}(i,j+1)} \leq \mathtt{Type}_i$ i.e. $j < i$.
But then $\mathrm{iso}\ A : \mathtt{Type}_i \not< \mathtt{Type} j \Rightarrow$ inconsistency.

Universe Polymorphism

With universe polymorphism (Sozeau & Tabareau [ITP'14]):

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

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

$$\begin{array}{ll} \mathsf{Proper}_{i'\,j'} & ((A \to A \to \mathtt{Type}_j) \to A \to \mathtt{Type}_j) \\ & (\mathrm{iso\_rel}\ A \longrightarrow \mathrm{eq}\ A \longrightarrow \mathrm{iso}) \\ & (\mathsf{Proper}_{i\,j}\ A) \end{array}$$

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

Actually, $\mathrm{crelation}(A : \mathtt{Type}_i) := A \to A \to \mathtt{Type}_j$ is already problematic: no relation equivalence or subrelation definition possible.

Computational relations

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 `Prop`-based setoids, e.g. groupoids...

Automation

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.

An example

Suppose the theory of monoids on `T`.
A goal: `x y : T |- x • ((ϵ • y) • ϵ).`

- `autorewrite with monoids` will do two rewrites with both unit laws, the proof term will be roughtly twice the goal size.
- `rewrite_strat (topdown (repeat (hints monoids)))` will first rewrite `ϵ • y` to `y` and directly after, `y • ϵ` to `y`, resulting in a proof term of size roughly that of the initial goal, and will be twice as fast as well.

Coq support for HoTT

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

Dependent Pattern-Matching

A word of caution:

- Strong elimination:

```
match x return if x then unit else nat with
 | true => tt
 | false => 0
end
```

  $\Rightarrow$ intensional character of inductive values.

- Dependent Pattern-Matching:

```
match (v : vector bool (S 0)) return bool with
 | Vcons a ?(0) v => a
end
```

  $\Rightarrow$ 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

2006 Eliminating Dependent Pattern-Matching – Goguen, McBride and McKinna.

$\approx$ 2010 GADTs in the programming languages community.

2014 Pattern Matching without K – Cockx, Devriese and Piessens

Overview of Equations

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

Compilation setup

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

1. Generation of a splitting tree from the clauses

Compilation setup

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

Compilation setup

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 $\mathcal{L}_{\mathsf{tac}}$ code

Heavily inspired by Goguen, McBride and McKinna (2006) and Norell (2007).

Compilation setup

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 $\mathcal{L}_{\mathsf{tac}}$ code
4. Derivation of auxilliary structures from the completed splitting tree

Heavily inspired by Goguen, McBride and McKinna (2006) and Norell (2007).

Searching for a splitting tree

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

Goal

Starting with $\mathrm{f}\ \Delta : \tau := \overrightarrow{\overrightarrow{p} \ldots}$,

find a *covering* of the context map $\Delta \vdash \overline{\Delta} : \Delta$ 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)...)])])
```

Translation from the splitting to Coq

For each node $\Delta \vdash ps : \Gamma \rightsquigarrow \Pi\ \Delta, \mathrm{f}_{\mathsf{comp}}\ ps$.

Translation from the splitting to Coq

For each node $\Delta \vdash ps : \Gamma \rightsquigarrow \Pi\ \Delta, \mathrm{f}_{\mathsf{comp}}\ ps$.

- Split(c, x, s): witnessed by dependent elimination. `dependent destruction`, using JMeq or user-given `K`/`hSet` proofs.

Translation from the splitting to Coq

For each node $\Delta \vdash ps : \Gamma \rightsquigarrow \Pi\ \Delta, \mathrm{f}_{\mathsf{comp}}\ ps$.

- Split(c, x, s): witnessed by dependent elimination. `dependent destruction`, using JMeq or user-given `K`/`hSet` proofs.
- Program(t): witnessed by t (w/ some substitution).

Translation from the splitting to Coq

For each node $\Delta \vdash ps : \Gamma \rightsquigarrow \Pi\ \Delta, \mathrm{f}_{\mathsf{comp}}\ ps$.

- Split(c, x, s): witnessed by dependent elimination. `dependent destruction`, using JMeq or user-given `K`/`hSet` 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!

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!

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.

Subterm relations and well-founded recursion

Well-founded recursion on the subterm relation for inductive families $\mathsf{I} : \Pi\ \Delta, \mathtt{Type}$.

Subterm relations and well-founded recursion

Well-founded recursion on the subterm relation for inductive families $\mathsf{I} : \Pi\ \Delta, \mathtt{Type}$.

- General definition of direct subterm:
  $\mathsf{I}_{subfull} : \Pi\ \Delta_l\ \Delta_r, \mathsf{I}\ \overline{\Delta_l} \to \mathsf{I}\ \overline{\Delta_r} \to \mathtt{Prop}$

Moving to $\mathtt{Type}$ is easy (though $\mathsf{I}_{subfull}$ is not always a proposition).

Subterm relations and well-founded recursion

Well-founded recursion on the subterm relation for inductive families $\mathsf{I} : \Pi\ \Delta, \mathtt{Type}$.

- General definition of direct subterm:
  $\mathsf{I}_{subfull} : \Pi\ \Delta_l\ \Delta_r, \mathsf{I}\ \overline{\Delta_l} \to \mathsf{I}\ \overline{\Delta_r} \to \mathtt{Prop}$
- Wrap the inductive type in a sigma and define an homogeneous relation on the sigma type:
  $\mathsf{I}_{sub} : \mathrm{relation}\ (\Sigma\ \Delta, \mathsf{I}\ \overline{\Delta})$

Moving to $\mathtt{Type}$ is easy (though $\mathsf{I}_{subfull}$ is not always a proposition).

Subterm relations and well-founded recursion

Well-founded recursion on the subterm relation for inductive families $\mathsf{I} : \Pi\ \Delta, \mathtt{Type}$.

- General definition of direct subterm:
  $\mathsf{I}_{subfull} : \Pi\ \Delta_l\ \Delta_r, \mathsf{I}\ \overline{\Delta_l} \to \mathsf{I}\ \overline{\Delta_r} \to \mathtt{Prop}$
- Wrap the inductive type in a sigma and define an homogeneous relation on the sigma type:
  $\mathsf{I}_{sub} : \mathrm{relation}\ (\Sigma\ \Delta, \mathsf{I}\ \overline{\Delta})$
- Extracts efficiently to a general fixpoint (assuming accessibility is defined in $\mathtt{Prop}$).

Moving to $\mathtt{Type}$ is easy (though $\mathsf{I}_{subfull}$ is not always a proposition).

Example: vectors

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

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}.
```

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) }.
```

Vectors with decidable equality

Using dependent elimination on decidable indices.

```
Equations unzip_dec {A B} `{EqDec A} `{EqDec B}
          {n} (v : vector (A × B) n) : vector A n × vector B n :=
unzip_dec A B _ _ n v by rec v :=
unzip_dec A B _ _ ?(O) nil := (nil, nil) ;
unzip_dec A B _ _ ?(S n) (cons (pair x y) n v) with unzip_dec v := {
  | pair xs ys := (cons x xs, cons y ys) }.

Print Assumptions unzip_dec.
```

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

- 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.

Generating a function graph

```
Equations filter {A} (l : list A) (p : A → 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 }.
```

Generated mutual induction principle

```
Check(filter_ind_mut :
  ∀ (P : ∀ (A : Type) (l : list A) (p : A → bool), filter_comp l p → Prop)
  (P0 : ∀ (A : Type) (a : A) (l : list A) (p : A → bool),
     bool → filter_comp (a :: l) p → Prop),

  (∀ A p, P A [] p []) →

  (∀ A a l p,
     filter_ind_1 A a l p (p a) (filter_obligation_2 (@filter) A a l p (p a)) →
     P0 A a l p (p a) (filter_obligation_2 (@filter) A a l p (p a)) →
     P A (a :: l) p (filter_obligation_2 (@filter) A a l p (p a))) →

  (∀ A a l p, filter_ind A l p (filter l p) →
     P A l p (filter l p) → P0 A a l p true (a :: filter l p)) →
  (∀ A a l p, filter_ind A l p (filter l p) →
     P A l p (filter l p) → P0 A a l p false (filter l p)) →

  ∀ A l p (f3 : filter_comp l p), filter_ind A l p f3 → P A l p f3).
```

Elimination principle

We prove filter respects the generated graph and derive:

```
Check (filter_elim :
  ∀ P : ∀ (A : Type) (l : list A) (p : A → bool), filter_comp l p → Prop,
  let P0 := fun (A : Type) (a : A) (l : list A) (p : A → bool)
       (refine : bool) (H : filter_comp (a :: l) p) ⇒
       p a = refine → P A (a :: l) p H
  in
  (∀ (A : Type) (p : A → bool), P A [] p []) →
  (∀ (A : Type) (a : A) (l : list A) (p : A → bool),
     P A l p (filter l p) → P0 A a l p true (a :: filter l p)) →
  (∀ (A : Type) (a : A) (l : list A) (p : A → bool),
     P A l p (filter l p) → P0 A a l p false (filter l p)) →
  ∀ (A : Type) (l : list A) (p : A → bool), P A l p (filter l p)).
```

Eliminating calls

The elimination principle can only be applied usefully to calls with *solely* variable arguments.

$$\Pi\ A\ (l : \mathsf{list}\ A),\ \mathrm{app}\ l\ [] = l$$

Eliminating calls

The elimination principle can only be applied usefully to calls with *solely* variable arguments.

$$\Pi\ A\ (l : \mathsf{list}\ A),\ \mathrm{app}\ l\ [] = l$$

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

$$\begin{array}{l}(\lambda\ (l\ l' : \mathsf{list}\ A)\ (r : \mathrm{app}_{\mathsf{comp}}\ l\ l'),\\ \qquad l' = [] \to r = \mathrm{app}\ l\ [] \to \mathrm{app}\ l\ [] = l)\\ \quad l\ []\ (\mathrm{app}\ l\ [])\end{array}$$

Directly apply the elimination principle and simplify the equations.

Summary

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]).

Discussion

- We moved to dependent equality (i.e. equality in sigma types) instead of JMeq. This is necessary to use hProp/hSet hypotheses (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 `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!

With nodes in detail

Consider a current problem $\Delta \vdash \overrightarrow{p} : \Gamma$ and a user clause $\mathtt{f}\ \overrightarrow{up}\ \mathtt{with}\ t_{pre} := \{\ e\ \}$ matching it. We typecheck t_{pre} into $t : \tau$ and use strenghtening and abstraction to find a new context

$$\Delta^t, x_t : \tau, \Delta_t[t/x_t] \text{ such that } \Delta^t, \Delta_t \sim \Delta$$

With nodes in detail

Consider a current problem $\Delta \vdash \overrightarrow{p} : \Gamma$ and a user clause $\mathtt{f}\ \overrightarrow{up}\ \mathtt{with}\ t_{pre} := \{\ e\ \}$ matching it. We typecheck t_{pre} into $t : \tau$ and use strenghtening and abstraction to find a new context

$$\Delta^t, x_t : \tau, \Delta_t[t/x_t] \text{ such that } \Delta^t, \Delta_t \sim \Delta$$

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

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

and return Refine(t, c, s).

With nodes in detail

Consider a current problem $\Delta \vdash \overrightarrow{p} : \Gamma$ and a user clause $\mathrm{f}\ \overrightarrow{up}$ `with` $t_{pre} := \{\ e\ \}$ matching it. We typecheck t_{pre} into $t : \tau$ and use strenghtening and abstraction to find a new context

$$\Delta^t, x_t : \tau, \Delta_t[t/x_t] \text{ such that } \Delta^t, \Delta_t \sim \Delta$$

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

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

and return Refine(t, c, s).
Compilation produces
$\ell.n : \Pi\ \Delta^t\ (x_t : \tau_\Delta)\ \Delta_t[t/x_t], (\mathrm{f}_{\mathsf{comp}}\ \overrightarrow{p})[t/x_t]$, we build

$$(\lambda\Delta, \ell.n\ \overline{\Delta^t}\ t\ \overline{\Delta_t}) : \Pi\ \Delta, \mathrm{f}_{\mathsf{comp}}\ \overrightarrow{p}$$

Elimination principle: inductive graph

For $\mathrm{f}.\ell : \Pi\ \Delta, \mathrm{f}_{\mathsf{comp}}\ \overrightarrow{t}$ we generate $\mathrm{f}.\ell_{\mathsf{ind}} : \Pi\ \Delta, \mathrm{f}_{\mathsf{comp}}\ \overrightarrow{t} \to \mathtt{Prop}$ and prove $\Pi\ \Delta, \mathrm{f}.\ell_{\mathsf{ind}}\ \overline{\Delta}\ (\mathrm{f}.\ell\ \overline{\Delta})$.

$\textsc{AbsRec}(\mathrm{f}, t)$ abstracts all the calls to $\mathrm{f}_{\mathsf{comp\_proj}}$ from the term t, returning a new derivation $\Gamma' \vdash t'$ where Γ' contains bindings of the form $x : \Pi\ \Delta, \mathrm{f}_{\mathsf{comp}}\ \overrightarrow{t}$ for all the recursive calls.

Define $\textsc{Hyps}(\Gamma)$ by a map to produce the corresponding inductive hyps of the form $H_x : \Pi\ \Delta, \mathrm{f}_{\mathsf{ind}}\ \overrightarrow{t}\ (x\ \overline{\Delta})$.

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 $\Psi \vdash t' = \textsc{AbsRec}(f, t)$ and return the statement
    $$\Pi\ \Delta\ \Psi\ \textsc{Hyps}(\Psi),\ \mathrm{f}.\ell_{\mathsf{ind}}\ \overrightarrow{p}\ t'$$
  - Refine$(t, \Delta' \vdash \overrightarrow{v}^x, x, \overrightarrow{v}_x : \Delta^x, x : \tau, \Delta_x, \ell.n)s$:
    Compute $\Psi \vdash t' = \textsc{AbsRec}(f, t)$ and return:
    $$\Pi\ \Delta\ \Psi\ \textsc{Hyps}(\Psi)\ (res : \mathrm{f}_{\mathsf{comp}}\ \overrightarrow{p})\ \mathrm{f}.\ell.\mathrm{n}_{\mathsf{ind}}\ \overline{\Delta^x}\ t'\ \overline{\Delta_x}\ res \to \mathrm{f}.\ell_{\mathsf{ind}}\ \overrightarrow{p}\ res$$
    We continue with the generation of the $\mathrm{f}.\ell.\mathrm{n}_{\mathsf{ind}}$ graph.