Equations for HoTT

Matthieu Sozeau, π.r², Inria Paris & IRIF

HoTT Conference
August 13th 2019
Carnegie Mellon University
Pittsburgh, PA, USA
Epigram/Agda/Idris-style pattern-matching definitions with first-match semantics and inaccessible (/dot) patterns

with and where clauses, pattern-matching lambdas

Inductive fin : nat → Set :=
  | fz : ∀ n : nat, fin (S n)  fin n ≃ [0, n)
  | fs : ∀ n : nat, fin n → fin (S n).

Equations fineq {k} (n m : fin k) : { n = m } + { n ≠ m } :=
  fineq fz fz := left idpath ;
  fineq (fs n) (fs m) with fineq n m ⇒ {
    fineq (fs n) (fs ?(n)) (left idpath) := left idpath ;
    fineq (fs n) (fs m) (right p) :=
      right (λ{ | idpath := p idpath })
  } ;
  fineq x y := right _.
EQUATIONS Reloaded (Sozeau & Mangin, ICFP’19)

- **Epigram/Agda/Idris**-style pattern-matching definitions
- *with* and *where* clauses, pattern-matching lambdas
- Nested and mutual structurally recursive and well-founded definitions: applies to inductive families (heterogeneous subterm relation)
Epigram/Agda/Idris-style pattern-matching definitions

*with* and *where* clauses, pattern-matching lambdas

Nested and mutual structurally recursive and well-founded definitions: applies to *inductive families* (heterogeneous subterm relation)

Propositional equations for defining clauses
EQUATIONS Reloaded (Sozeau & Mangin, ICFP’19)

- Epigram/Agda/Idris-style pattern-matching definitions
- with and where clauses, pattern-matching lambdas
- Nested and mutual structurally recursive and well-founded definitions: applies to inductive families (heterogeneous subterm relation)
- Propositional equations for defining clauses
- Systematic derivation of functional elimination principles (involves transport in the dependent case)
Epigram/Agda/Idris-style pattern-matching definitions

with and where clauses, pattern-matching lambdas

Nested and mutual structurally recursive and well-founded definitions: applies to inductive families (heterogeneous subterm relation)

Propositional equations for defining clauses

Systematic derivation of functional elimination principles (involves transport in the dependent case)

Original no-confusion notion and homogeneous equality (UIP on a type-by-type basis, configurable)
EQUATIONS Reloaded (Sozeau & Mangin, ICFP’19)

- Epigram/Agda/Idris-style pattern-matching definitions
- with and where clauses, pattern-matching lambdas
- Nested and mutual structurally recursive and well-founded definitions: applies to inductive families (heterogeneous subterm relation)
- Propositional equations for defining clauses
- Systematic derivation of functional elimination principles (involves transport in the dependent case)
- Original no-confusion notion and homogeneous equality (UIP on a type-by-type basis, configurable)
- Parameterized by a logic: Prop (extraction-friendly), Type (proof-relevant equality), SProp (strict proof-irrelevance), ...
Epigram/Agda/Idris-style pattern-matching definitions

with and where clauses, pattern-matching lambdas

Nested and mutual structurally recursive and well-founded definitions: applies to inductive families (heterogeneous subterm relation)

Propositional equations for defining clauses

Systematic derivation of functional elimination principles (involves transport in the dependent case)

Original no-confusion notion and homogeneous equality (UIP on a type-by-type basis, configurable)

Parameterized by a logic: Prop (extraction-friendly), Type (proof-relevant equality), SProp (strict proof-irrelevance), ...

Purely definitional, axiom-free translation to Coq (CIC) terms
Reasoning support: elimination principle

Equations $\text{filter} \ {\{A\}} (l : \text{list} \ A) (p : A \rightarrow \text{bool}) : \text{list} \ A :=$

filter nil $p := \text{nil}$;

filter $(\text{cons} \ a \ l) \ p$ with $p \ a := \{
\ | \ \text{true} := a :: \text{filter} \ l \ p ;
\ | \ \text{false} := \text{filter} \ l \ p \}.$
Reasoning support: elimination principle

Equations filter \(\{A\}\) \((l : \text{list} \ A) \ (p : A \rightarrow \text{bool}) : \text{list} \ A :=\)

\[
\text{filter} \ \text{nil} \ p := \text{nil};
\]

\[
\text{filter} \ (\text{cons} \ a \ l) \ p \ \text{with} \ p \ a := \{
| \text{true} := a :: \text{filter} \ l \ p ;
| \text{false} := \text{filter} \ l \ p \}.
\]

Check \((\text{filter}\_\text{elim} : \)

\[
\forall \ (P : \forall (A : \text{Type}) \ (l : \text{list} \ A) \ (p : A \rightarrow \text{bool}), \ \text{list} \ A \rightarrow \text{Type}),
\]

let \(P0 := \text{fun} \ (A : \text{Type}) \ (a : A) \ (l : \text{list} \ A) \ (p : A \rightarrow \text{bool})\)

\[
(\text{refine} : \text{bool}) \ (\text{res} : \text{list} \ A) \Rightarrow
\]

\[
p \ a = \text{refine} \rightarrow P \ A \ (a :: l) \ p \ \text{res}
\]

in

\[
(\forall \ (A : \text{Type}) \ (p : A \rightarrow \text{bool}), \ P \ A \ [] \ p \ []) \rightarrow
\]

\[
(\forall \ (A : \text{Type}) \ (a : A) \ (l : \text{list} \ A) \ (p : A \rightarrow \text{bool}),
\]

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

\[
(\forall \ (A : \text{Type}) \ (a : A) \ (l : \text{list} \ A) \ (p : A \rightarrow \text{bool}),
\]

\[
P \ A \ l \ p \ (\text{filter} \ l \ p) \rightarrow P0 \ A \ a \ l \ p \ \text{false} \ (\text{filter} \ l \ p)) \rightarrow
\]

\[
\forall \ (A : \text{Type}) \ (l : \text{list} \ A) \ (p : A \rightarrow \text{bool}), \ P \ A \ l \ p \ (\text{filter} \ l \ p))\).
Outline

1. Dependent Pattern-Matching 101
   - Pattern-Matching and Unification
   - Covering

2. Dependent Pattern-Matching and Axiom K
   - History and preliminaries
   - A homogeneous no-confusion principle
   - Support for HoTT
Pattern-matching and unification

Idea: reasoning up-to the theory of equality and constructors

Example: to eliminate \( t : \text{vector } A \ m \), we unify with:

1. \( \text{vector } A \ 0 \) for \( \text{vnil} \)
2. \( \text{vector } A \ (S \ n) \) for \( \text{vcons} \)

Unification \( t \equiv u \simsto Q \) can result in:

- \( Q = \text{Fail} \)
- \( Q = \text{Success } \sigma \) (with a substitution \( \sigma \));
- \( Q = \text{Stuck } t \) if \( t \) is outside the theory (e.g. a constant)

Two successes in this example for \([m := 0]\) and \([m := S \ n]\) respectively.
Unification rules

**Solution**

\[
x \not\in \mathcal{FV}(t) \\
x \equiv t \leadsto \text{Success } \sigma[x := t]
\]

**Occur-check**

\[
C \text{ constructor context} \\
x \equiv C[x] \leadsto \text{Fail}
\]

**Discrimination**

\[
C \_ \equiv D \_ \leadsto \text{Fail}
\]

**Injectivity**

\[
t_1 \ldots t_n \equiv u_1 \ldots u_n \leadsto Q \\
C \ t_1 \ldots t_n \equiv C \ u_1 \ldots u_n \leadsto Q
\]

**Patterns**

\[
p_1 \equiv q_1 \leadsto \text{Success } \sigma \\
(p_2 \ldots p_n)\sigma \equiv (q_2 \ldots q_n)\sigma \leadsto Q \\
p_1 \ldots p_n \equiv q_1 \ldots q_n \leadsto Q \cup \sigma
\]

**Deletion**

\[
t \equiv t \leadsto \text{Success } []
\]

**Stuck**

\[
\text{Otherwise} \\
t \equiv u \leadsto \text{Stuck } u
\]
Unification examples

- \( O \equiv S \ n \leadsto \text{Fail} \)
- \( S \ m \equiv S \ (S \ n) \leadsto \text{Success} \ [m := S \ n] \)
- \( O \equiv m + O \leadsto \text{Stuck} \ (m + O) \)

Stuck cases indicate a variable to eliminate, to refine the pattern-matching problem (here variable \( m \)).

Pattern-matching compilation uses unification to:
- Decide which program clause to choose
- Decide which constructors can apply when we eliminate a variable
Overlapping clauses and first-match semantics:

Equations \(\text{equal} \ (m \ n : \text{nat}) : \text{bool} :=\)

\begin{align*}
\text{equal} \ O \ O & := \text{true}; \\
\text{equal} \ (S \ m') \ (S \ n') & := \text{equal} \ m' \ n'; \\
\text{equal} \ m \ n & := \text{false}.
\end{align*}
Overlapping clauses and first-match semantics:

Equations equal \((m \ n : \text{nat}) : \text{bool} :=\)

\[
\begin{align*}
equal \text{O O} & \equiv \text{true}; \\
equal (S m') (S n') & \equiv \text{equal } m' \ n'; \\
equal m \ n & \equiv \text{false}.
\end{align*}
\]

cover\((m \ n : \text{nat} \vdash m \ n) \rightarrow \text{O O} \equiv m \ n \leadsto \text{Stuck } m\)
Overlapping clauses and first-match semantics:

\textbf{Equations} equal \((m \ n : \text{nat}) : \text{bool} :=
\text{equal} \ O \ O := \text{true};
\text{equal} \ (\text{S} \ m') \ (\text{S} \ n') := \text{equal} \ m' \ n';
\text{equal} \ m \ n := \text{false}.

\text{Split}(m \ n : \text{nat} \vdash m \ n, m, [ ])

Overlapping clauses and first-match semantics:

\[\text{Equations} \quad \text{equal} \ (m \ n : \ \text{nat}) : \ \text{bool} := \]
\[\text{equal} \ O \ O := \text{true}; \]
\[\text{equal} \ (\text{S} \ m') \ (\text{S} \ n') := \text{equal} \ m' \ n'; \]
\[\text{equal} \ m \ n := \text{false}. \]

\[\text{Split} (m \ n : \ \text{nat} \vdash n \ m, m, [\]
\[\text{cover}(n : \ \text{nat} \vdash O \ n) \]
\[\text{cover}(m' \ n : \ \text{nat} \vdash (\text{S} \ m') \ n)]\]
Overlapping clauses and first-match semantics:

Equations equal \( (m \ n : \text{nat}) : \text{bool} := \)

\[
equal \ O \ O := \text{true};
\]

\[
equal \ (S \ m') \ (S \ n') := \text{equal} \ m' \ n';
\]

\[
equal \ m \ n := \text{false}.
\]

\[
\text{Split}(m \ n : \text{nat} \vdash m \ n, m, [
\text{Split}(n : \text{nat} \vdash O \ n, n, [
\text{Compute}(\vdash O \ O \Rightarrow \text{true}),
\text{Compute}(n' : \text{nat} \vdash O \ (S \ n') \Rightarrow \text{false})]),
\text{cover}(m' \ n : \text{nat} \vdash (S \ m') \ n)])
\]
Overlapping clauses and first-match semantics:

Equations equal (m n : nat) : bool :=
  equal O O := true;
  equal (S m') (S n') := equal m' n';
  equal m n := false.

Split(m n : nat ⊢ m n, m, [
  Split(n : nat ⊢ O n, n, [
    Compute(⊢ O O ⇒ true),
    Compute(n' : nat ⊢ O (S n') ⇒ false)]),
  Split(m' n : nat ⊢ (S m') n, n, [
    Compute(m' : nat ⊢ (S m') O ⇒ false),
    Compute(m' n' : nat ⊢ (S m') (S n') ⇒ equal m' n')]])}
Dependent pattern-matching

Inductive vector \((A : \text{Type}) : \text{nat} \rightarrow \text{Type} : =\) 
| \text{vnil} : \text{vector } A \; 0 
| \text{vcons} : A \rightarrow \forall (n : \text{nat}), \text{vector } A \; n \rightarrow \text{vector } A \; (S \; n). 

Equations \text{vtail } \ A \; n \ (v : \text{vector } A \; (S \; n)) : \text{vector } A \; n : = \text{vtail } A \; n \ (\text{vcons } ?(n) \; v) : = v.

Each variable must appear only once, except in inaccessible patterns.

\text{cover}(A \; n \; v : \text{vector } A \; (S \; n) \vdash A \; n \; v)
Inductive vector (A : Type) : nat → Type :=
| vnil : vector A 0
| vcons : A → ∀ (n : nat), vector A n → vector A (S n).

Equations vtail A n (v : vector A (S n)) : vector A n :=
vtail A n (vcons _ ?(n) v) := v.

Each variable must appear only once, except in inaccessible patterns.

Split(A n (v : vector A (S n)) ⊢ A n v, v, [
  Fail; // O ̸= S n
  cover(A n' a (v' : vector A n') ⊢ A n' (vcons a ?(n') v'))]])
Dependent pattern-matching

\textbf{Inductive vector} \((A : \text{Type}) : \text{nat} \to \text{Type} :=\)

\begin{itemize}
  \item \text{vnil} : \text{vector} A 0
  \item \text{vcons} : A \to \forall (n : \text{nat}), \text{vector} A n \to \text{vector} A (S n).
\end{itemize}

\textbf{Equations} \text{vtail} A n (v : \text{vector} A (S n)) : \text{vector} A n :=

\begin{align*}
\text{vtail} A n (\text{vcons} a ?(n) v) &:= v.
\end{align*}

Each variable must appear only once, except in \textit{inaccessible patterns}.

\begin{align*}
\text{Split}(A n (v : \text{vector} A (S n)) &\vdash A n v, v, [ \\
\text{Fail}; &// \ S n \neq O \\
\text{Compute}(A n' a (v' : \text{vector} A n') &\vdash A n' (\text{vcons} a ?(n') v') \\
&\Rightarrow v')])
\end{align*}
Equations \( \text{nth} \{A\ n\} (v : \text{vector } A\ n) (f : \text{fin } n) : A := \)
\[
\text{nth} (\text{cons } x\ _\ _) (\text{fz } _) := x; \\
\text{nth} (\text{cons } _\ ?(n)\ v) (\text{fs } n\ f) := \text{nth } v\ f.
\]
Outline

1. Dependent Pattern-Matching 101
   - Pattern-Matching and Unification
   - Covering

2. Dependent Pattern-Matching and Axiom K
   - History and preliminaries
   - A homogeneous no-confusion principle
   - Support for HoTT
Coquand (1992) introduced the dependent pattern-matching notion as a new primitive in type theory, introducing K at the same time:

\[ K : \forall A \ (x : A) \ (e : x = x), \ e = \text{eq\_refl} \]
A bit of history

- Coquand (1992) introduced the dependent pattern-matching notion as a new primitive in type theory, introducing K at the same time:

\[ K : \forall A \ (x : A) \ (e : x = x), e = \text{eq\_refl} \]

- This axiom was shown independent from type theory (MLTT or CIC) by Hofmann and Streicher (1994).
  - It is a consequence of proof-irrelevance.
Coquand (1992) introduced the dependent pattern-matching notion as a new primitive in type theory, introducing $K$ at the same time:

$$K : \forall A (x : A) \ (e : x = x), e = \text{eq\_refl}$$

This axiom was shown independent from type theory (MLTT or CIC) by Hofmann and Streicher (1994).

- It is a consequence of proof-irrelevance.
- It is incompatible with the univalence axiom.
A bit of history

- **Coquand (1992)** introduced the dependent pattern-matching notion as a new primitive in type theory, introducing K at the same time:

  \[ K : \forall A (x : A) \ (e : x = x), e = \text{eq\_refl} \]

- This axiom was shown independent from type theory (MLTT or CIC) by **Hofmann and Streicher (1994)**.
  - It is a consequence of proof-irrelevance.
  - It is incompatible with the univalence axiom

- **McBride (1999); Goguen et al. (2006)** introduce the idea of “internalizing“ dependent pattern-matching using just the eliminators for inductive families and equality. This uses an axiomatized heterogeneous equality type, even stronger than K.
Derived constructions on inductives

To internalize dependent pattern-matching, we must witness the unification steps with proof terms.

- For the **Solution** rule we just use $J$. 

---

All simplification steps must have good computational behavior: they are strong unifiers / type equivalences ($\simeq$).

Going back and forth through a strong equivalence must preserve reflexive equalities definitionally.
Derived constructions on inductives

To internalize dependent pattern-matching, we must witness the unification steps with proof terms.

- For the `SOLUTION` rule we just use `J`.
- For manipulations of telescopes, standard `Σ`-types (with their `η` law) suffice.
Derived constructions on inductives

To internalize dependent pattern-matching, we must witness the unification steps with proof terms.

- For the Solution rule we just use $J$.
- For manipulations of telescopes, standard $\Sigma$-types (with their $\eta$ law) suffice.
- For inductives $I : \Pi \Delta, \textbf{Type}$, we automatically derive:
  1. Their standard case-analysis eliminator
  2. A signature: $\bar{I} := \Sigma i : \bar{\Delta}. I \ i$ (i.e., the total space over $I$)
  3. NoConfusion$_\bar{I}$: for Injectivity and Discrimination.
  4. EqDec$_\bar{I}$: decidable equality (if derivable) for Deletion (which requires UIP in general).
  5. Subterm$_\bar{I}$: the subterm relation, and its well-foundedness, which allows to prove acyclicity of inductive values (e.g. $x \neq S \ x$) (Occur-Check)
Derived constructions on inductives

To internalize dependent pattern-matching, we must witness the unification steps with proof terms.

▶ For the Solution rule we just use $J$.
▶ For manipulations of telescopes, standard $\Sigma$-types (with their $\eta$ law) suffice.
▶ For inductives $I : \Pi \Delta, \text{Type}$, we automatically derive:

1. Their standard case-analysis eliminator
2. A signature: $\bar{I} := \Sigma i : \Delta. I \ i$ (i.e., the total space over $I$)
3. NoConfusion$_I$: for Injectivity and Discrimination.
4. EqDec$_I$: decidable equality (if derivable) for Deletion (which requires UIP in general).
5. Subterm$_I$: the subterm relation, and its well-foundedness, which allows to prove acyclicity of inductive values (e.g. $x \neq S \ x$) (Occur-Check)

▶ All simplification steps must have good computational behavior: they are strong unifiers / type equivalences ($\equiv_s$).

Going back and forth through a strong equivalence must preserve reflexive equalities definitionally.
Heterogeneous vs homogeneous equality:
To eliminate $v$ in

$$\Gamma = n : \mathbb{N}, v : \text{vector } A (S \ n) \vdash \tau$$

We generalize $v$ and its index:

$$\Gamma' = n' : \mathbb{N}, v' : \text{vector } A n'$$

We also add an equality to get a goal equivalent to the original:

$$\Gamma' \vdash \forall \Gamma, (n', v') = \Sigma n : \mathbb{N}. \text{vector } A n (S n, v) \rightarrow \tau$$

Eliminate $v'$ and simplify the equalities in the theory of constructors and uninterpreted functions (decidable). Done!
To compile pattern-matching, we use the no-confusion principle on inductive families to solve equations like:

\[ \text{fs } n \ f = \text{fin } (S \ n) \ \text{fs } n \ f' \]
To compile pattern-matching, we use the no-confusion principle on inductive families to solve equations like:

\[ \text{fs } n \ f =_{\text{fin}} (S \ n) \ fs \ n \ f' \]
\[ \simeq_s (n; f) =_{\Sigma x: \text{nat}. \text{fin } x} (n; f') \]
To compile pattern-matching, we use the no-confusion principle on inductive families to solve equations like:

\[
\begin{align*}
\text{fs } n \ f &= \text{fin } (S \ n) \ \text{fs } n \ f' \\
\simeq& \ (n; f) = \sum_{x : \text{nat}} \text{fin } x (n; f') \\
\simeq& \ \sum (e : n = \text{nat } n). e \# f = \text{fin } n \ f'
\end{align*}
\]
Dependent Pattern-Matching and Axiom K

To compile pattern-matching, we use the no-confusion principle on inductive families to solve equations like:

\[
\text{fs } n \ f =_{\text{fin}} (S \ n) \ \text{fs } n \ f' \\
\simeq_s (n; f) =_{\sum x : \text{nat} \ \text{fin} \ x} (n; f') \\
\simeq_s \sum (e : n =_{\text{nat}} n). e \# f =_{\text{fin}} n \ f'
\]

To simplify and obtain \( f = f' \) through a strong equivalence, we would need to know

\[
(e : n =_{\text{nat}} n) \equiv \text{eq_refl}
\]

As you know, not true!
To compile pattern-matching, we use the no-confusion principle on inductive families to solve equations like:

\[
fs \ n \ f =_{\text{fin}} (S \ n) \ fs \ n \ f' \\
\simeq_{s} (n; f) =_{\Sigma x: \text{nat}. \text{fin}} x (n; f') \\
\simeq_{s} \Sigma(e : n =_{\text{nat}} n).e \# f =_{\text{fin}} n \ f'
\]

To simplify and obtain \( f = f' \) through a strong equivalence, we would need to know

\[
(e : n =_{\text{nat}} n) \equiv \text{eq\_refl}
\]

As you know, not true!

However, \text{isHSet} \text{nat}, so \( (n =_{\text{nat}} n) \simeq \top \) for a regular (non-strong) type equivalence. These do not provide the right reduction behavior for elaborated definitions however: they get stuck on the indices, as the \text{UIP} proof for \text{nat} inspects the indices recursively.
Strong equivalence

\[ S \ n = S \ m \]

\[ \text{NoConf} \ (S \ n)(S \ m) \equiv n = m \]

\[ \text{noconf}^{-1} x \ y \ (\text{noconf} \ x \ y \ e) = e \] (regular)

\[ \text{noconf}^{-1} (S \ n) (S \ n) (\text{noconf} (S \ n) (S \ n) \text{idpath}) \equiv \text{idpath} \] (strong)
Question: how to restrict pattern-matching to not rely on K?

Cockx (2017): proof-relevant unification algorithm based on simplification of equalities, avoiding K by restricting deletion (can't be made into a strong equivalence):

**DELETION**

\[
\text{eq\_refl} : t =_T t \rightsquigarrow \text{Success} \[
\]
Pattern-Matching without K

**Question:** how to restrict pattern-matching to not rely on K?

**Cockx (2017):** proof-relevant unification algorithm based on simplification of equalities, avoiding K by restricting deletion (can’t be made into a strong equivalence):

\[
\text{Deletion} \\
\text{eq_refl} : t =_T t \rightsquigarrow \text{Success} []
\]

Also restricts the injectivity rule for indexed inductive types, e.g.:

\[
\text{Injectivity} \\
\text{noconf} e : n =_{\text{nat}} n \rightsquigarrow Q \\
e : \text{fz } n =_{\text{fin}} (\text{S } n) \text{fz } n \rightsquigarrow Q
\]

Huge restriction in practice, lifted by Cockx and Devriese (2018) using a higher-dimensional unification algorithm. We propose a more direct way to treat it.
Brady et al. (2003) proposed the notion of *forced argument* of constructors to justify compile-time optimizations for the representation of constructors. For fin:

\[
\text{Inductive } \text{fin} : \text{nat} \rightarrow \text{Set} := \\
| \text{fz} : \forall n : \text{nat}, \text{fin} (\text{S } n) \\
| \text{fs} : \forall n : \text{nat}, \text{fin } n \rightarrow \text{fin } (\text{S } n).
\]

The \( n \) argument of each constructor is “forced” by the index \( \text{S } n \).
Brady et al. (2003) proposed the notion of *forced argument* of constructors to justify compile-time optimizations for the representation of constructors. For `fin`:

\[
\text{Inductive } \text{fin} : \text{nat} \rightarrow \text{Set} := \\
| \text{fz} : \forall n : \text{nat}, \text{fin} (S \ n) \\
| \text{fs} : \forall n : \text{nat}, \text{fin} n \rightarrow \text{fin} (S \ n).
\]

The \( n \) argument of each constructor is “forced” by the index \( S \ n \).

The associated *heterogeneous* no-confusion principle:

Equations `NoConfHet` \{\( n \) \( n' \)\} (\( f \) : fin \( n \)) (\( f' \) : fin \( n' \)) : Type :=

\[
\text{NoConfHet} (\text{fz} \ n) (\text{fz} \ n') := n = n';
\]
\[
\text{NoConfHet} (\text{fs} \ n \ f) (\text{fs} \ n' \ f') := (n; f) = (n' ; f');
\]
\[
\text{NoConfHet } - - := \bot.
\]
Brady et al. (2003) proposed the notion of *forced argument* of constructors to justify compile-time optimizations for the representation of constructors. For `fin`:

```
Inductive fin : nat → Set :=
| fz : ∀ n : nat, fin (S n)
| fs : ∀ n : nat, fin n → fin (S n).
```

The `n` argument of each constructor is “forced” by the index `S n`.

A refined **homogeneous** no-confusion principle to handle this:

```
Equations NoConf {n} (f f' : fin n) : Type :=
  NoConf (fz ?(n)) (fz n) := ⊤;
  NoConf (fs ?(n) f) (fs n f') := f = f';
  NoConf _ _ := ⊥.
```
These two functions form strong type equivalence: it transports reflexivity proofs to reflexivity proofs definitionally for equalities between constructor-headed terms.
Pattern-matching without K

This justifies the new injectivity rule:

\[
\text{noconf eq} \equiv \text{eq_refl : (fz n) = (S n) (fz n)} \rightsquigarrow \text{Success} \]

\[
e : \text{fz n = (S n) fz n} \rightsquigarrow \text{Success} [e := \text{eq_refl}]\]

⇒ Forced arguments do not need to be unified: they are definitionally equal by typing.

- In Agda, this justifies the unification used in the --without-K mode
- Equations uses this refined no-confusion principle to provide axiom-free definitions, even when they involved complex inversions on inductive families.
- We also have a mode where user-provided proofs of UIP can be used, but do not guarantee the computational behavior in that case (it can be useful in proofs though).
Implementation details

To implement pattern-matching simplification **internally** we use:

- Cumulative, universe polymorphic notions of equality and sigma-types/telescopes.
To implement pattern-matching simplification *internally* we use:

- Cumulative, universe polymorphic notions of equality and sigma-types/telescopes.
- Derivation of the homogeneous no-confusion principle for indexed families in addition to the heterogeneous one.
To implement pattern-matching simplification internally we use:

- Cumulative, universe polymorphic notions of equality and sigma-types/telescopes.
- Derivation of the homogeneous no-confusion principle for indexed families in addition to the heterogeneous one.
- **Definitional** fixpoint equations for recursion on the derived subterm relation, otherwise *propositional*. Similar to Epigram/Lean’s Below definitions.
Comparison with Cockx and Devriese

Cockx and Devriese (2018): higher-dimensional unification.

- More expressive, based on the functoriality of $\text{ap}$ and the fact that it preserves equivalences. Can solve box-filling problems (e.g. pattern matching on squares of equalities).
Comparison with Cockx and Devriese

Cockx and Devriese (2018): higher-dimensional unification.

- More expressive, based on the functoriality of $ap$ and the fact that it preserves equivalences. Can solve box-filling problems (e.g. pattern matching on squares of equalities).
- Gives the same notion of path equality for the forced arguments problems, albeit with a more complex proof term.
Comparison with Cockx and Devriese

Cockx and Devriese (2018): higher-dimensional unification.

- More expressive, based on the functoriality of $\text{ap}$ and the fact that it preserves equivalences. Can solve box-filling problems (e.g. pattern matching on squares of equalities).
- Gives the same notion of path equality for the forced arguments problems, albeit with a more complex proof term.
- Unfortunately, quite incompatible with Coq’s syntactic guardedness check (esp due to representation of telescopes using sigma-types). Equations provides support for well-founded recursion though (see paper for details).
Comparison with Cockx and Devriese

Cockx and Devriese (2018): higher-dimensional unification.

- More expressive, based on the functoriality of \( \text{ap} \) and the fact that it preserves equivalences. Can solve box-filling problems (e.g. pattern matching on squares of equalities).
- Gives the same notion of path equality for the forced arguments problems, albeit with a more complex proof term.
- Unfortunately, quite incompatible with \( \text{Coq} \)'s syntactic guardedness check (esp due to representation of telescopes using sigma-types). \text{Equations} provides support for well-founded recursion though (see paper for details).
- Methodological difference: \( \text{Agda} \) relies on the metatheoretical proofs of op. cit. (many done internally), but one must trust the implementation and its interaction with e.g. sized-types. Here we elaborate instead (not touching the kernel).
Comparison with Cockx and Devriese

Cockx and Devriese (2018): higher-dimensional unification.

- More expressive, based on the functoriality of \( \text{ap} \) and the fact that it preserves equivalences. Can solve box-filling problems (e.g. pattern matching on squares of equalities).

- Gives the same notion of path equality for the forced arguments problems, albeit with a more complex proof term.

- Unfortunately, quite incompatible with \texttt{Coq}'s syntactic guardedness check (esp due to representation of telescopes using sigma-types). \texttt{Equations} provides support for well-founded recursion though (see paper for details).

- Methodological difference: \texttt{Agda} relies on the metatheoretical proofs of op. cit. (many done internally), but one must trust the implementation and its interaction with e.g. sized-types. Here we elaborate instead (not touching the kernel).

- Future work: integration in our simplification engine.
Equations \( \text{sing CONTR} \) \{\( A \}\) \( (x : A) : \text{Contr} (\Sigma y : A, x = y) := \)
\[
\begin{align*}
sing\_\text{contr} x & := \{| \text{center} := (x, 1); \text{contr} := \text{contr} |\} \\
\text{where} \ \text{contr} & := \forall y : (\Sigma y : A, x = y), (x, 1) = y := \\
\text{contr} (y, 1) & := 1.
\end{align*}
\]

▶ A new elimination tactic dependent elimination foo as \([p1 \ldots p_n]\) based on the simplification engine. Gives robust naming and ordering of inversions, and patterns can even use notations.

▶ Use of arbitrary user-provided proofs UIP is configurable: show \( \text{UIP} \ 1 \ (\equiv \text{isHSet} \ 1) \) and simplification uses it for the deletion and injectivity rules. These UIP proofs are relevant for reduction (unless using \( \text{SProp} \)).

▶ Integration with dependent elimination tactic: abstracts away that, e.g. \( \text{nat} \) has UIP to eliminate \( (e : n = n) \) to \( \text{idpath} \).
Ongoing and Future work

Ongoing work:

▶ IDE support for refinement mode (Proof-General & VS-Coq)
▶ Support for Coq-HoTT and UniMath (reusing the basic definitions from those libraries).
▶ Integration with \textit{SProp} and an equality with built-in UIP ("strict" pattern-matching).
▶ Integration with Almost-Full relations for (foundational) termination checking: subsumes Size-Change Termination, Terminator (Vytiniotis et al., 2012).

Future work:

▶ Implementation and elaboration correctness proof in \textsc{MetaCoq} (Sozeau et al., 2019).
▶ Link with rewrite rules: dependent pattern-matching and well-founded recursion as a definitional translation to CIC.
▶ Extension to co-patterns and co-recursion.
github.com/mattam82/Coq-Equations#hott-logic

# opam install coq.dev coq-hott.dev \coq-equations-hott.dev

Soon to be released along with Coq 8.10 (uses the equality type as defined in the HoTT/Coq library).


Input syntax

**term, type**
\[ t, \tau ::= x | \lambda x : \tau, t | \forall x : \tau, \tau' | \lambda \{ u \mapsto t \} \]

**binding**
\[ d ::= (x : \tau) | (x := t : \tau) \]

**context**
\[ \Gamma, \Delta ::= \overrightarrow{d} \]

**programs**
\[ progs ::= prog \mathbf{mutual}. \]

**mutual programs**
\[ mutual ::= with p | where \]

**where clause**
\[ where ::= \text{``string''} := \text{term (}:\text{scope})? \]

**notation**
\[ not ::= \text{``string''} := \text{term (}:\text{scope})? \]

**program**
\[ p, prog ::= f \Gamma : \tau (\text{by annot})? := clauses \]

**annotation**
\[ annot ::= \text{struct } x | \text{wf } t \overset{R}{\rightarrow} \]

**clauses**
\[ clauses ::= \overrightarrow{c} | \{ \overrightarrow{c} \} \]

**user clause**
\[ c ::= f \overset{\overrightarrow{u}}{\mapsto} n | \overset{\overrightarrow{u}}{\mapsto} n \]

**user pattern**
\[ up ::= x | C \overset{\overrightarrow{u}}{\mapsto} ?(t) | (x := up) \]

**user node**
\[ n ::= := \text{t where } | := ! x \]
\[ \overset{\overrightarrow{w}}{=} \text{with } t \overset{\overrightarrow{1}}{=}, t \overset{\overrightarrow{2}}{=} clauses \]
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>context map</strong></td>
<td>$c ::= \Delta \vdash \overrightarrow{p} : \Gamma$</td>
</tr>
<tr>
<td><strong>pattern</strong></td>
<td>$p ::= x \mid C \overrightarrow{p} \mid ?(t)$</td>
</tr>
<tr>
<td><strong>splitting</strong></td>
<td>$spl ::= \text{Split}(c, x, (spl?)^n) \mid \text{Compute}(c'' = \ell '' rhs)$</td>
</tr>
<tr>
<td><strong>node</strong></td>
<td>$rhs ::= t, w \mid \text{Refine}(t, c, \ell, spl)$</td>
</tr>
<tr>
<td><strong>label</strong></td>
<td>$\ell ::= \epsilon \mid \ell.n \quad (n \in \mathbb{N})$</td>
</tr>
</tbody>
</table>
Elimination principle: inductive graph

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

$\text{AbsRec}(f, t)$ 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 $x : \Pi \Delta, f_{\text{comp}} \rightarrow^t t$ for all the recursive calls.

Define $\text{HYPS}(\Gamma)$ by a map to produce the corresponding inductive hyps of the form $H_x : \Pi \Delta, f_{\text{ind}} \rightarrow^t (x \Delta)$.
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**\((\Delta \vdash \overrightarrow{p} : \Gamma'' = \xi'' \text{rhs})\) : By case on \(\text{rhs}\):
  - \(t\) : 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**\((t, \Delta' \vdash \overrightarrow{u}^x, x, \overrightarrow{u}_x : \Delta^x, x : \tau, \Delta_x, \ell.n, s)\) : Compute \(\Psi \vdash t' = \text{AbsRec}(f, t)\) and return:

\[
\Pi \Delta \Psi \text{HYPS}(\Psi) (res : f_{\text{comp}} \overrightarrow{p})
\]

\[
f.\ell.n_{\text{ind}} \overrightarrow{\Delta^x} t' \overrightarrow{\Delta_x} res \rightarrow f.\ell_{\text{ind}} \overrightarrow{p} res
\]

We continue with the generation of the \(f.\ell.n_{\text{ind}}\) graph.
Outline

1. Dependent Pattern-Matching 101
   - Pattern-Matching and Unification
   - Covering

2. Dependent Pattern-Matching and Axiom K
   - History and preliminaries
   - A homogeneous no-confusion principle
   - Support for HoTT
Recursion

- Syntactic guardness checks are fragile (and buggy)
- Do not work well with abstraction/modularity
- Restricted to structural recursion on a single argument, with no currying allowed

**Idea** Use the logic instead: well-founded recursion!
Use **well-founded** recursion on the subterm relation for inductive families $I : \Pi \Delta, \text{Type.}$
Use **well-founded** recursion on the subterm relation for inductive families $l : \Pi \Delta, \text{Type}$.

- **General definition of direct subterm:**
  $$l_{\text{sub}} : \Pi \Delta_l \Delta_r, l \Delta_l \rightarrow l \Delta_r \rightarrow \text{Prop}$$

- **Define the subterm relation on telescopes:**
  $$l_{\text{sub}} : \text{relation} \ (\Sigma \Delta, l \ \Delta)$$
Derive Subterm for vector.
Derive Subterm for vector.

**Inductive** vector\_strict\_subterm (A : Type)
\[ : \forall H H0 : \text{nat}, \text{vector} A H \to \text{vector} A H0 \to \text{Prop} := \]

vector\_strict\_subterm\_1_1 : \forall (a : A) (n : \text{nat}) (H : \text{vector} A n),

vector\_strict\_subterm A n (S n) H (\text{Vcons} a H).

**Check** vector\_subterm : \forall A : \text{Type}, \text{relation} \{\text{index} : \text{nat} \& \text{vector} A \text{index}\}.
Derive Subterm for `vector`.

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

: ∀ H H0 : nat, vector A H → vector A H0 → Prop :=

  `vector_strict_subterm_1_1` : ∀ (a : A) (n : nat) (H : vector A n),

  `vector_strict_subterm A n (S n) H (Vcons a H)`.  

**Check** `vector_subterm : ∀ A : Type, relation {index : nat & vector A index}`.

**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) Vnil := (Vnil, Vnil) ;`

`unzip A B ?(S n) (Vcons (pair x y) n v) with unzip v := {`  

| (pair xs ys) := (Vcons x xs, Vcons y ys) }.`

---

*Equations for HoTT*
1 Dependent Pattern-Matching 101
   - Pattern-Matching and Unification
   - Covering

2 Dependent Pattern-Matching and Axiom K
   - History and preliminaries
   - A homogeneous no-confusion principle
   - Support for HoTT
Goal: keep an abstract view of definitions if desired.

- Equations for the clauses hold definitionally in CCI.
- If UIP is used, only propositionally.
- All put together in a rewrite database, \( f \) can be considered opaque.
Elimination principle

- Abstracts away the pattern-matching and recursion pattern of the program.
- Can be used to modularly work on definitions not yet proven terminating.
- Generates equalities for each `with` in the program.
- Supports nested and mutual structural or well-founded recursions: one predicate by function/`where` clause.
- Generated in `Type` if possible, to allow proof-relevant definitions: useful in HoTT for example, or to prove `reflect` predicates.
▶ simp \( f \) allows to rewrite with the equations of a definition \( f \)

▶ noconf \( H \) uses pattern-matching simplification to simplify an equality hypothesis (combines injection, discriminate, subst, and acyclicity)

▶ dependent elimination \( id \) as \([p1 \ldots pn]\) launches a dependent pattern-matching covering on the goal variable \( id \). You can use arbitrary notations for patterns, no more cryptic destruct as clauses!