EQUATIONS for HoTT

Matthieu Sozeau, $\pi.r^2$, Inria Paris & IRIF

HoTT Conference
August 13th 2019
Carnegie Mellon University
Pittsburgh, PA, USA

EQUATIONS Reloaded (Sozeau & Mangin, ICFP'19)

- 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)
  | fs : ∀ n : nat, fin n → fin (S n).
```

$\mathsf{fin}\ n \simeq [0, n)$

```
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)

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

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)

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)

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), ...

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), ...

Purely definitional, axiom-free translation to COQ (CIC) terms

Reasoning support: elimination principle

```
Equations filter {A} (l : list A) (p : A → bool) : list A :=
filter nil p := nil ;
filter (cons a l) p with p a := {
  | true := a :: filter l p ;
  | false := filter l p }.
```

Reasoning support: elimination principle

```
Equations filter {A} (l : list A) (p : A → bool) : list A :=
filter nil p := nil ;
filter (cons a l) p with p a := {
  | true := a :: filter l p ;
  | false := filter l p }.
```

```
Check (filter_elim :
  ∀ (P : ∀ (A : Type) (l : list A) (p : A → bool), list A → Type),
  let P0 := fun (A : Type) (a : A) (l : list A) (p : A → bool)
      (refine : bool) (res : list A) ⇒
      p a = refine → P A (a :: l) p res
  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)).
```

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 : vector A m, we unify with:

1. vector A O for vnil
2. vector A $(\mathsf{S}\ n)$ for vcons

Unification $t \equiv u \rightsquigarrow Q$ can result in:

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

Two successes in this example for $[m := 0]$ and $[m := \mathsf{S}\ n]$ respectively.

Unification rules

SOLUTION

$$\frac{x \notin \mathcal{FV}(t)}{x \equiv t \rightsquigarrow \texttt{Success}\ \sigma[x := t]}$$

OCCUR-CHECK

$$\frac{C \text{ constructor context}}{x \equiv C[x] \rightsquigarrow \texttt{Fail}}$$

DISCRIMINATION

$$\frac{}{\mathsf{C}\ \_ \equiv \mathsf{D}\ \_ \rightsquigarrow \texttt{Fail}}$$

INJECTIVITY

$$\frac{t_1 \dots t_n \equiv u_1 \dots u_n \rightsquigarrow Q}{\mathsf{C}\ t_1 \dots t_n \equiv \mathsf{C}\ u_1 \dots u_n \rightsquigarrow Q}$$

PATTERNS

$$\frac{p_1 \equiv q_1 \rightsquigarrow \texttt{Success}\ \sigma \qquad (p_2 \dots p_n)\sigma \equiv (q_2 \dots q_n)\sigma \rightsquigarrow Q}{p_1 \dots p_n \equiv q_1 \dots q_n \rightsquigarrow Q \cup \sigma}$$

DELETION

$$\frac{}{t \equiv t \rightsquigarrow \texttt{Success}\ []}$$

STUCK

$$\frac{\text{Otherwise}}{t \equiv u \rightsquigarrow \texttt{Stuck}\ u}$$

Unification examples

- $\mathsf{O} \equiv \mathsf{S}\ n \rightsquigarrow \texttt{Fail}$
- $\mathsf{S}\ m \equiv \mathsf{S}\ (\mathsf{S}\ n) \rightsquigarrow \texttt{Success}\ [m := \mathsf{S}\ n]$
- $\mathsf{O} \equiv m + \mathsf{O} \rightsquigarrow \texttt{Stuck}\ (m + \mathsf{O})$

`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

Pattern-matching compilation

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.
```

cover$(m\ n : \mathsf{nat} \vdash m\ n : (m\ n : \mathsf{nat})) \leftarrow$ `context map`

Pattern-matching compilation

Overlapping clauses and first-match semantics:

$\texttt{Equations}\ \mathrm{equal}\ (m\ n : \mathsf{nat}) : \mathsf{bool} :=$
$\quad \mathrm{equal}\ \mathsf{O}\ \mathsf{O} := \mathsf{true};$
$\quad \mathrm{equal}\ (\mathsf{S}\ m')\ (\mathsf{S}\ n') := \mathrm{equal}\ m'\ n';$
$\quad \mathrm{equal}\ m\ n := \mathsf{false}.$

$\mathsf{cover}(m\ n : \mathsf{nat} \vdash m\ n) \to \mathsf{O}\ \mathsf{O} \equiv m\ n \rightsquigarrow \texttt{Stuck}\ m$

Pattern-matching compilation

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 $\vdash$ m n, m, [ ])

Pattern-matching compilation

Overlapping clauses and first-match semantics:

$\texttt{Equations}\ \mathrm{equal}\ (m\ n : \mathsf{nat}) : \mathsf{bool} :=$
$\quad \mathrm{equal}\ \mathsf{O}\ \mathsf{O} := \mathsf{true};$
$\quad \mathrm{equal}\ (\mathsf{S}\ m')\ (\mathsf{S}\ n') := \mathrm{equal}\ m'\ n';$
$\quad \mathrm{equal}\ m\ n := \mathsf{false}.$

$\mathsf{Split}(m\ n : \mathsf{nat} \vdash n\ m, m, [$
$\quad \mathsf{cover}(n : \mathsf{nat} \quad \vdash \mathsf{O}\ n)$
$\quad \mathsf{cover}(m'\ n : \mathsf{nat} \vdash (\mathsf{S}\ m')\ n)])$

Pattern-matching compilation

Overlapping clauses and first-match semantics:

$\texttt{Equations}\ \mathrm{equal}\ (m\ n : \mathsf{nat}) : \mathsf{bool} :=$
$\quad \mathrm{equal}\ \mathsf{O}\ \mathsf{O} := \mathsf{true};$
$\quad \mathrm{equal}\ (\mathsf{S}\ m')\ (\mathsf{S}\ n') := \mathrm{equal}\ m'\ n';$
$\quad \mathrm{equal}\ m\ n := \mathsf{false}.$

$\mathsf{Split}(m\ n : \mathsf{nat} \vdash m\ n, m, [$
$\quad \mathsf{Split}(n : \mathsf{nat} \vdash \mathsf{O}\ n, n, [$
$\quad\quad \mathsf{Compute}(\vdash \mathsf{O}\ \mathsf{O} \Rightarrow \mathsf{true}),$
$\quad\quad \mathsf{Compute}(n' : \mathsf{nat} \vdash \mathsf{O}\ (\mathsf{S}\ n') \Rightarrow \mathsf{false})]),$
$\quad \mathsf{cover}(m'\ n : \mathsf{nat} \vdash (\mathsf{S}\ m')\ n)])$

Pattern-matching compilation

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 $\vdash m\ n$, m, [
  Split(n : nat $\vdash$ O n, n, [
    Compute($\vdash$ O O $\Rightarrow$ true),
    Compute(n' : nat $\vdash$ O (S n') $\Rightarrow$ false)]),
  Split($m'\ n$: nat $\vdash$ (S m') n, n, [
    Compute(m' : nat $\vdash$ (S m') O $\Rightarrow$ false),
    Compute($m'\ n'$: nat $\vdash$ (S m') (S n') $\Rightarrow$ equal $m'\ n'$)])])

Dependent pattern-matching

```
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.

$\mathrm{cover}(A\ n\ v : \mathsf{vector}\ A\ (\mathsf{S}\ n)) \vdash A\ n\ v)$

Dependent pattern-matching

$\texttt{Inductive}$ vector $(A : \texttt{Type}) :$ nat $\to \texttt{Type} :=$
$\mid$ vnil : vector A 0
$\mid$ vcons : $A \to \forall\,(n :$ nat$),$ vector $A\ n \to$ vector A (S n).

$\texttt{Equations}$ vtail $A\ n\ (v :$ vector A (S n)) : vector $A\ n :=$
  vtail $A\ n$ (vcons _ $\texttt{?(}n\texttt{)}$ v) $:= v$.

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

Split($A\ n\ (v :$ vector A (S n)) $\vdash A\ n\ v$, v, [
  $\texttt{Fail}$; // O $\neq$ S n
  cover($A\ n'\ a\ (v' :$ vector $A\ n') \vdash A\ n'$ (vcons a $\texttt{?(}n'\texttt{)}$ v'))])

Dependent pattern-matching

```
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($A\ n\ (v : \mathsf{vector}\ A\ (\mathsf{S}\ n)) \vdash A\ n\ v$, v, [
 Fail; // $\mathsf{S}\ n \neq \mathsf{O}$
 Compute($A\ n'\ a\ (v' : \mathsf{vector}\ A\ n') \vdash A\ n'\ (\mathsf{vcons}\ a\ \texttt{?(}n'\texttt{)}\ v')$
  $\Rightarrow v'$)])

Refinement across objects

```
Equations nth {A n} (v : vector A n) (f : fin n) : A :=
  nth (cons x _ _) (fz _) := x;
  nth (cons _ ?(n) v) (fs n f) := nth v f.
```

Outline

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:

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

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:

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

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

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:

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

- 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:

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

- 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.

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

Derived constructions on inductives

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

- For the $\textsc{Solution}$ rule we just use J.
- For manipulations of telescopes, standard Σ-types (with their η law) suffice.
- For inductives $\mathsf{I} : \Pi\Delta, \texttt{Type}$, we automatically derive:
  1. Their standard case-analysis eliminator
  2. A signature: $\bar{\mathsf{I}} := \Sigma i : \overline{\Delta}.\mathsf{I}\ i$ (i.e., the total space over I)
  3. **NoConfusion**$_{\bar{\mathsf{I}}}$: for $\textsc{Injectivity}$ and $\textsc{Discrimination}$.
  4. **EqDec**$_{\bar{\mathsf{I}}}$: decidable equality (if derivable) for $\textsc{Deletion}$ (which requires UIP in general).
  5. **Subterm**$_{\bar{\mathsf{I}}}$: the subterm relation, and its well-foundedness, which allows to prove acyclicity of inductive values (e.g. $x \neq \mathsf{S}\ x$) ($\textsc{Occur-Check}$)
- All simplification steps must have good computational behavior: they are *strong* unifiers / type equivalences ($\simeq_s$). Going back and forth through a strong equivalence must preserve reflexive equalities definitionally.

Dependent Elimination Compilation

Heterogeneous vs homogeneous equality:
To eliminate v in

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

We generalize v and its index:

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

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

- $\Gamma' \vdash \forall \Gamma, (n', v') =_{\Sigma n : \mathbb{N}. \textsf{vector}\ A\ n} (\textsf{S}\ n, v) \to \tau$

Eliminate v' and simplify the equalities in the theory of constructors and uninterpreted functions (decidable). Done!

Dependent Pattern-Matching and Axiom K

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

$$\mathsf{fs}\ n\ f =_{\mathsf{fin}\ (\mathsf{S}\ n)} \mathsf{fs}\ n\ f'$$

Dependent Pattern-Matching and Axiom K

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

$$\begin{aligned} & \mathsf{fs}\ n\ f =_{\mathsf{fin}\ (\mathsf{S}\ n)} \mathsf{fs}\ n\ f' \\ \simeq_s \quad & (n; f) =_{\Sigma x:\mathsf{nat}.\mathsf{fin}\ x} (n; f') \end{aligned}$$

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

$$\begin{array}{rl} & \mathsf{fs}\ n\ f =_{\mathsf{fin}\ (\mathsf{S}\ n)} \mathsf{fs}\ n\ f' \\ \simeq_s & (n; f) =_{\Sigma x:\mathsf{nat}.\mathsf{fin}\ x} (n; f') \\ \simeq_s & \Sigma(e : n =_{\mathsf{nat}} n).e \sharp f =_{\mathsf{fin}\ n} f' \end{array}$$

Dependent Pattern-Matching and Axiom K

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

$$\begin{array}{rl} & \mathsf{fs}\ n\ f =_{\mathsf{fin}\ (\mathsf{S}\ n)} \mathsf{fs}\ n\ f' \\ \simeq_s & (n; f) =_{\Sigma x:\mathsf{nat}.\mathsf{fin}\ x} (n; f') \\ \simeq_s & \Sigma(e : n =_{\mathsf{nat}} n).e \sharp f =_{\mathsf{fin}\ n} f' \end{array}$$

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

$$(e : n =_{\mathsf{nat}} n) \equiv \mathsf{eq\_refl}$$

As you know, not true!

Dependent Pattern-Matching and Axiom K

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

$$\begin{array}{ll} & \mathsf{fs}\ n\ f =_{\mathsf{fin}\ (\mathsf{S}\ n)} \mathsf{fs}\ n\ f' \\ \simeq_s & (n; f) =_{\Sigma x:\mathsf{nat}.\mathsf{fin}\ x} (n; f') \\ \simeq_s & \Sigma(e : n =_{\mathsf{nat}} n).e \sharp f =_{\mathsf{fin}\ n} f' \end{array}$$

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

$$(e : n =_{\mathsf{nat}} n) \equiv \mathsf{eq\_refl}$$

As you know, not true!

However, isHSet nat, so $(n =_{\mathsf{nat}} n) \simeq \top$ 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 UIP proof for nat inspects the indices recursively.

Strong equivalence

$$\begin{tikzcd}
\boxed{\mathsf{S}\ n = \mathsf{S}\ m} \arrow[r, bend left, "\mathrm{noconf}"] & \boxed{\mathrm{NoConf}\ (\mathsf{S}\ n)(\mathsf{S}\ m) \equiv n = m} \arrow[l, bend left, "\mathrm{noconf}^{-1}"]
\end{tikzcd}$$

$$\mathrm{noconf}^{-1}\ x\ y\ (\mathrm{noconf}\ x\ y\ e) = e \qquad \text{(regular)}$$

$$\mathrm{noconf}^{-1}\ (\mathsf{S}\ n)\ (\mathsf{S}\ n)\ (\mathrm{noconf}\ (\mathsf{S}\ n)\ (\mathsf{S}\ n)\ \mathsf{idpath}) \equiv \mathsf{idpath} \qquad \text{(strong)}$$

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):

$$\frac{\textsc{Deletion}}{\mathsf{eq\_refl} : t =_T t \rightsquigarrow \mathtt{Success}\ []}$$

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):

$$\dfrac{\textsc{Deletion}}{\mathsf{eq\_refl} : t =_T t \rightsquigarrow \mathtt{Success}\ []}$$

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

$$\textsc{Injectivity}\ \dfrac{\mathrm{noconf}\ e : n =_{\mathsf{nat}} n \rightsquigarrow Q}{e : \mathsf{fz}\ n =_{\mathsf{fin}\ (\mathsf{S}\ n)} \mathsf{fz}\ n \rightsquigarrow Q}$$

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.

Forced arguments

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.

Forced arguments

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.

The associated heterogeneous no-confusion principle:

```
Equations NoConfHet {n n'} (f : fin n) (f' : fin n') : Type :=
  NoConfHet (fz n) (fz n') := n = n';
  NoConfHet (fs n f) (fs n' f') := (n; f) = (n'; f');
  NoConfHet _ _ := ⊥.
```

Forced arguments

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 _ _ := ⊥.
```

Homogeneous No-Confusion

$\texttt{Eqns}\ \mathrm{noconf}\ \{n\}\ (f\ f' : \mathsf{fin}\ n)\ (e : f = f') : \mathrm{NoConf}\ f\ f' :=$

$\quad \ldots$

$\texttt{Eqns}\ \mathrm{noconfeq}\ \{n\}\ (f\ f' : \mathsf{fin}\ n)\ (e : \mathrm{NoConf}\ f\ f') : f = f' :=$

$\quad \mathrm{noconfeq}\ (\mathsf{fz}\ \texttt{?(}n\texttt{)})\ (\mathsf{fz}\ n)\ \_ := \mathsf{eq\_refl};$

$\quad \ldots$

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:

$$\frac{\text{noconfeq } (\text{noconf } e) \equiv \mathsf{eq\_refl} : (\mathsf{fz}\ n) =_{\mathsf{fin}\ (\mathsf{S}\ n)} (\mathsf{fz}\ n) \rightsquigarrow \mathtt{Success}\ []}{e : \mathsf{fz}\ n =_{\mathsf{fin}\ (\mathsf{S}\ n)} \mathsf{fz}\ n \rightsquigarrow \mathtt{Success}\ [e := \mathsf{eq\_refl}]}$$

$\Rightarrow$ 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.

Implementation details

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.

Implementation details

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 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 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 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).
- Methodological difference: 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 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).
- Methodological difference: 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.

Support for HoTT

```
Equations sing_contr {A} (x : A) : Contr (Σ y : A, x = y) :=
  sing_contr x := {| center := (x, 1); contr := contr |}
    where contr : ∀ y : (Σ y : A, x = y), (x, 1) = y :=
          contr (y, 1) := 1.
```

- A new elimination tactic `dependent elimination foo as [p1|..|pn]` 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 UIP I (≡ isHSet I) and simplification uses it for the deletion and injectivity rules. These UIP proofs are relevant for reduction (unless using `SProp`).
- Integration with dependent elimination tactic: abstracts away that, e.g. nat has UIP to eliminate $(e : n = n)$ to idpath.

Ongoing and Future work

Ongoing work:

- IDE support for refinement mode (Proof-General & VSCoq)
- Support for Coq-HoTT and UniMath (reusing the basic definitions from those libraries).
- Integration with `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 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.

Equations for HoTT

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).

Bibliography

Edwin Brady, Conor McBride, and James McKinna. Inductive Families Need Not Store Their Indices. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *TYPES*, volume 3085 of *Lecture Notes in Computer Science*, pages 115–129. Springer, 2003.

Jesper Cockx. *Dependent Pattern Matching and Proof-Relevant Unification*. PhD thesis, Katholieke Universiteit Leuven, Belgium, 2017.

Jesper Cockx and Dominique Devriese. *Proof-relevant unification: Dependent pattern matching with only the axioms of your type theory*. *J. Funct. Program.*, 28:e12, 2018.

Thierry Coquand. Pattern Matching with Dependent Types, 1992. Proceedings of the Workshop on Logical Frameworks.

Healfdene Goguen, Conor McBride, and James McKinna. Eliminating Dependent Pattern Matching. In Kokichi Futatsugi, Jean-Pierre Jouannaud, and José Meseguer, editors, *Essays Dedicated to Joseph A. Goguen*, volume 4060 of *Lecture Notes in Computer Science*, pages 521–540. Springer, 2006.

Martin Hofmann and Thomas Streicher. A Groupoid Model Refutes Uniqueness of Identity Proofs. In *LICS*, pages 208–212. IEEE Computer Society, 1994.

Conor McBride. *Dependently Typed Functional Programs and Their Proofs*. PhD thesis, University of Edinburgh, 1999.

Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project. Submitted, April 2019.

Dimitrios Vytiniotis, Thierry Coquand, and David Wahlstedt. Stop When You Are Almost-Full - Adventures in Constructive Termination. In Lennart Beringer and Amy P. Felty, editors, *Interactive Theorem Proving - Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings*, volume 7406 of *Lecture Notes in Computer Science*, pages 250–265. Springer, 2012.

`term, type`	$t,\ \tau$	$::=$	$x \mid \lambda x : \tau, t \mid \forall x : \tau, \tau' \mid \lambda\{\ \overrightarrow{\vec{up} := t}^{+}$
`binding`	d	$::=$	$(x\ :\ \tau) \mid (x\ :=\ t\ :\ \tau)$
`context`	Γ, Δ	$::=$	$\vec{d}$
`programs`	$progs$	$::=$	$prog\ \overrightarrow{mutual}\ .$
`mutual programs`	$mutual$	$::=$	`with` $p \mid where$
`where clause`	$where$	$::=$	`where` $p \mid$ `where` not
`notation`	not	$::=$	`''`$string$`''` $:=$ $term$ $(:\ scope)?$
`program`	$p, prog$	$::=$	$\mathsf{f}\ \Gamma\ :\ \tau$ (`by` *annot*)? $:=$ $clauses$
`annotation`	$annot$	$::=$	`struct` $x \mid$ `wf` $t\ R$
`clauses`	$clauses$	$::=$	$\vec{c} \mid \{\ \vec{c}\ \}$
`user clause`	c	$::=$	$\mathsf{f}\ \vec{up}\ n \mid \overrightarrow{\mid\ up}^{+}\ n$
`user pattern`	up	$::=$	$x \mid \mathsf{C}\ \vec{up} \mid$ `?(` t `)` $\mid (x\ :=\ up)$
`user node`	n	$::=$	$:=\ t\ \overrightarrow{where} \mid$ `:=!` x
			$\mid$ `with` $t\ ,\vec{t}\ :=\ clauses$

Program representation: splitting trees

`context map`	c	$::=$	$\Delta \vdash \overrightarrow{p} : \Gamma$
`pattern`	p	$::=$	$x \mid \mathsf{C}\ \overrightarrow{p} \mid$ `?`(t)
`splitting`	spl	$::=$	$\mathsf{Split}(c, x, (spl?)^n) \mid \mathsf{Compute}(c"=¿"rhs)$
`node`	rhs	$::=$	$t, w \mid \mathsf{Refine}(t, c, \ell, spl)$
`label`	ℓ	$::=$	$\epsilon \mid \ell.n \quad (n \in \mathbb{N})$

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 $H_x : \Pi\ \Delta, \mathrm{f}_{\mathsf{ind}}\ \overrightarrow{t}\ (x\ \overline{\Delta})$.

Direct translation from the splitting tree:

- Split(c, x, s), Rec(v, s) : collect the constructors for the subsplitting(s) s, if any.
- Compute$(\Delta \vdash \overrightarrow{p} : \Gamma\text{"=¿"}rhs)$: By case on rhs:
  - 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:

$$\begin{array}{l}\Pi\ \Delta\ \Psi\ \textsc{Hyps}(\Psi)\ (res : \mathrm{f}_{\mathsf{comp}}\ \overrightarrow{p}) \\ \qquad \mathrm{f}.\ell.\mathrm{n}_{\mathsf{ind}}\ \overline{\Delta^{x}}\ t'\ \overline{\Delta_{x}}\ res \to \mathrm{f}.\ell_{\mathsf{ind}}\ \overrightarrow{p}\ res\end{array}$$

    We continue with the generation of the $\mathrm{f}.\ell.\mathrm{n}_{\mathsf{ind}}$ graph.

Outline

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!

Subterm relations and well-founded recursion

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

Subterm relations and well-founded recursion

Use **well-founded** recursion on the subterm relation for inductive families $\mathsf{I} : \Pi\ \Delta, \texttt{Type}$.

- General definition of direct subterm:
  $\mathsf{I}_{sub} : \Pi\ \Delta_l\ \Delta_r, \mathsf{I}\ \overline{\Delta_l} \rightarrow \mathsf{I}\ \overline{\Delta_r} \rightarrow \texttt{Prop}$
- Define the subterm relation on telescopes:
  $\mathsf{I}_{sub} : \text{relation}\ (\Sigma\ \Delta, \mathsf{I}\ \overline{\Delta})$

Subterm relation example: vectors

```
Derive Subterm for vector.
```

Subterm relation example: vectors

```
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}.
```

Subterm relation example: vectors

```
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) }.
```

Outline

Reasoning support: equations

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.

Tactics

- `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 .. 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!