EQUATIONS Reloaded

Matthieu Sozeau & Cyprien Mangin, π.r^2, Inria Paris & IRIF

August 19th 2019
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 \neq 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 (\lambda{ | idpath := p idpath }) ) ) ;
fineq x y := right _.
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
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
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

Based on homogeneous equality and an original no-confusion principle
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

Based on homogeneous equality and an original no-confusion principle

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

Based on homogeneous equality and an original no-confusion principle

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
Outline

1. Dependent Pattern-Matching and Unification

2. A Short History of DPM

3. A Homogeneous No-Confusion Principle

4. In the Paper and Implementation
Idea: reasoning up-to the theory of equality and constructors

Example: to eliminate $t : \text{fin } m$, we unify with:

1. $\text{fin } (S \ n)$ for $fz$
2. $\text{fin } (S \ n)$ for $fs$

Unification $t \equiv u \rightsquigarrow Q$ 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: $[m := S \ n]$. 
### Unification rules

#### Solution

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

#### Occur-check

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

\[C\text{ constructor context}\]
Unification rules

**Solution**

\[
x \notin \mathcal{FV}(t) \\
x \equiv t \leadsto \text{Success } \sigma[x := ?(t)]
\]

**Occur-check**

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

**Discrimination**

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

** Injectivity**

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

**Solution**

\[ x \notin \mathcal{FV}(t) \]

\[ x \equiv t \leadsto \text{Success } \sigma[x := ?(t)] \]

**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**

\[ p_1 \equiv q_1 \leadsto \text{Success } \sigma \]

\[ (p_2 \ldots p_n)\sigma \equiv (q_2 \ldots q_n)\sigma \leadsto Q \]

\[ p_1 \ldots p_n \equiv q_1 \ldots q_n \leadsto Q \cup \sigma \]
Solution
\[ x \notin \mathcal{FV}(t) \]
\[ x \equiv t \leadsto \text{Success } \sigma[x := ?(t)] \]

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

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

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

DPM compilation uses unification to:

- Decide which program clause to choose
- Decide which constructors can apply when we eliminate a variable
Coquand (1992): DPM definitions as a new primitive in type theory, introducing K/UIP at the same time:

\[ K : \forall A \ (x : A) \ (e : x = x), e = \text{eq\_refl} \]
Coquand (1992): DPM definitions as a new primitive in type theory, introducing K/UIP 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).

- Implied by proof-irrelevance
Coquand (1992): DPM definitions as a new primitive in type theory, introducing K/UIP 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).

- Implied by proof-irrelevance
- Incompatible with univalence
A bit of history

- **Coquand (1992)**: DPM definitions as a new primitive in type theory, introducing K/UIP 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).
  - Implied by proof-irrelevance
  - Incompatible with univalence

- **McBride (1999); Goguen et al. (2006)**: internalizing DPM using solely eliminators for inductive families and equality. **But...**
A bit of history

- **Coquand (1992):** DPM definitions as a new *primitive* in type theory, introducing $K/UIP$ at the same time:

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

- This axiom was shown independent from type theory (MLTT or CIC) by Hofmann and Streicher (1994).
  - Implied by proof-irrelevance
  - Incompatible with univalence

- **McBride (1999); Goguen et al. (2006):** internalizing DPM using solely eliminators for inductive families and equality. **But . . .** using an axiomatized heterogeneous equality type, even stronger than equality with UIP.
To eliminate $f$ in

$$\Gamma = n : \mathbb{N}, f : \text{fin} (S\ n) \vdash \tau$$

1. Generalize $f$ and its index and add an equality:

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

2. Eliminate $f'$

3. Simplify the equalities in the theory of constructors and uninterpreted functions (decidable).
Internalizing DPM

Idea: witness unification steps with proof terms.

▶ For the Solution rule we just use \( J \).
Idea: witness unification steps with proof terms.

- For the Solution rule we just use $J$.
- For manipulations of telescopes of equalities, standard $\Sigma$-types (with their $\eta$ law) suffice.
Internalizing DPM

Idea: witness unification steps with proof terms.

- For the Solution rule we just use J.
- For manipulations of telescopes of equalities, 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)
Simplification steps must be *strong* unifiers ([Cockx and Devriese, 2018](#)) / type equivalences ($\simeq_s$).
Computational soundness and strong equivalences

Simplification steps must be \textit{strong} unifiers (Cockx and Devriese, 2018) / type equivalences ($\simeq_s$).

\begin{align*}
\text{noconf} \\
S n = S m & \quad \text{NoConf} (S n)(S m) \equiv n = m \\
\text{noconf}^{-1} := \text{ap } S
\end{align*}

\begin{align*}
\text{noconf}^{-1} x y (\text{noconf } x y e) &= e \quad \text{(regular)} \\
\text{noconf}^{-1} (S n) (S n) (\text{noconf } (S n) (S n) \text{idpath}) &\equiv \text{idpath} \quad \text{(strong)}
\end{align*}

$\Rightarrow$ Cannot be defined by recursion on $n$, only pattern-matching.
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:

\[
\begin{align*}
\text{fs } n \ f & = \text{fin } (\text{S } n) \ \text{fs } n \ f' \\
\sim \quad (n; f) & = \Sigma x: \text{nat}. \ \text{fin } x (n; f')
\end{align*}
\]  

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

\[
\begin{align*}
\text{fs } n \ f & =_{\text{fin}} (S \ n) \ \text{fs } n \ f' \\
\sim & \quad (n; f) =_{\Sigma x: \text{nat}. \text{fin} \ x} (n; f') \\
\simeq & \quad \Sigma (e : n =_{\text{nat}} n). e \not= f =_{\text{fin}} n \ f' \quad \text{DEF. OF TELESCOPES}
\end{align*}
\]
To compile pattern-matching, we use the no-confusion principle on inductive families to solve equations like:

\[
\begin{align*}
fs \ n \ f &= \text{fin} (S \ n) \ fs \ n \ f' \\
\sim (n; f) &= \Sigma x: \text{nat}. \text{fin} \ x (n; f') \\
\sim_s \Sigma (e : n = \text{nat} \ n). e \neq f &= \text{fin} \ n \ f' \quad \text{Def. of telescopes}
\end{align*}
\]

To simplify and obtain \( f = f' \) through a strong equivalence, we would need to apply identity elimination (\( J \)) to

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

Not possible!
Dependent Pattern-Matching and Axiom K

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

\[
\begin{align*}
\text{fs } n \ f & = \text{fin } (S \ n) \ \text{fs } n \ f' \\
\sim & \ (n; f) = \Sigma x: \text{nat}. \text{fin } x \ (n; f') \quad \text{Injectivity} \\
\sim_s & \ \Sigma (e : n = \text{nat } n). \ e \# f = \text{fin } n \ f' \quad \text{Def. of telescopes}
\end{align*}
\]

To simplify and obtain \( f = f' \) through a strong equivalence, we would need to apply identity elimination (\( J \)) to \( (e : n = \text{nat } n) \)

Not possible!

However, UIP \( \text{nat} \), so \( (n = \text{nat } n) \sim \top \), by recursion on \( n \).

Not a strong equivalence!
Question: how to restrict pattern-matching to not rely on K?
Cockx et al. (2014): proof-relevant unification algorithm based on simplification of equalities, avoiding K by forbidding deletion (not strong):

\[
\text{DELETION}\\
\frac{e : t =_T t \leadsto \text{Success} \: [e := \text{eq_refl}]}{}
\]
Question: how to restrict pattern-matching to not rely on K?

Cockx et al. (2014): proof-relevant unification algorithm based on simplification of equalities, avoiding K by forbidding deletion (not strong):

```
Deletion

e : t = T t \leadsto Success [e := eq_refl]
```
Question: how to restrict pattern-matching to not rely on K?

Cockx et al. (2014): proof-relevant unification algorithm based on simplification of equalities, avoiding K by forbidding deletion (not strong):

\[
\text{DELETION}
\]
\[
e : t =_T t \leadsto \text{Success} \ [e := \text{eq_refl}]
\]

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

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

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 (erasure in this case). For fin:

\[
\text{Inductive fin : nat } \rightarrow \text{ Set } \leftarrow \\
| \text{ fz : } \forall n : \text{ nat}, \text{ fin } (S \ n) \\
| \text{ fs : } \forall n : \text{ nat}, \text{ fin } n \rightarrow \text{ fin } (S \ n). \\
\]
Brady et al. (2003) proposed the notion of *forced argument* of constructors to justify compile-time optimizations for the representation of constructors (erasure in this case). For `fin`:

\[
\text{Inductive } \text{fin} : \text{nat} \rightarrow \text{Set} := \\
| \text{fz} : \forall n : \text{nat}, \text{fin} (\text{S } n) \\
| \text{fs} : \forall n : \text{nat}, \text{fin } n \rightarrow \text{fin} (\text{S } n).
\]
Brady et al. (2003) proposed the notion of *forced argument* of constructors to justify compile-time optimizations for the representation of constructors (erasure in this case). For \textit{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 associated heterogeneous no-confusion principle:

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

\begin{verbatim}
Inductive fin : nat → Set :=
  | fz : ∀ n : nat, fin (S n)
  | fs : ∀ n : nat, fin n → fin (S n).
\end{verbatim}

The associated *homogeneous* no-confusion principle:

\textbf{Equations} NoConf \{n\} \((f f' : \text{fin} n)\) : \text{Type} :=

- \text{NoConf} (fz ?(n)) (fz n) := \top;
- \text{NoConf} (fs ?(n) f) (fs n f') := f = f';
- \text{NoConf} _ _ := \bot.

Homogeneous No-Confusion

Eqns noconf \{n\} (f f' : fin n) (e : f = f') : NoConf f f' := …

Eqns noconfeq \{n\} (f f' : fin n) (e : NoConf f f') : f = f' :=
   noconfeq (fz ?(n)) (fz n) _ := eq_refl;
   …

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

**Strong** for equalities *between constructor-headed terms* only
(prerequisite of the Injectivity rule).
Homogeneous Injectivity

\[
\text{Homogeneous Injectivity} \\
\text{noconfeq (noconf } e) : (fz \ n) =_{\text{fin}} (S \ n) (fz \ n) \rightsquigarrow \\
\text{Success } [e := \text{noconfeq (noconf } e)] \\
e : fz \ n =_{\text{fin}} (S \ n) fz \ n \rightsquigarrow
\]
Homogeneous Injectivity

\[\text{Homogeneous Injectivity} \]
\[\text{noconfeq } (\text{noconf } e) : (\text{fz } n) =_{\text{fin}} (\text{s } n) (\text{fz } n) \rightsquigarrow \]
\[\text{Success } [e := \text{noconfeq } l] \]
\[e : \text{fz } n =_{\text{fin}} (\text{s } n) \text{ fz } n \rightsquigarrow \]
Homogeneous Injectivity

Homogeneous Injectivity
noconfeq (noconf e) : (fz n) =fin (s n) (fz n) ↢
Success [e := eq_refl]

\[ e : fz n =_{fin} (s n) fz n \rightsquigarrow \text{Success } [e := \text{eq_refl}] \]
Homogeneous Injectivity

**Homogeneous Injectivity**

\[
\text{noconfpeq } \text{noconf } e : (\text{fz } n) =_{\text{fin}} (S \ n) (\text{fz } n) \rightsquigarrow \text{Success } [e := \text{eq}\_\text{refl}]
\]

\[
e : \text{fz } n =_{\text{fin}} (S \ n) \text{fz } n \rightsquigarrow \text{Success } [e := \text{eq}\_\text{refl}]
\]

\Rightarrow \text{Forced arguments need not be unified: } \equiv \text{ by typing.}
Homogeneous Injectivity

**Homogeneous Injectivity**

\[ \text{noconfeq } (\text{noconf } e) : (\text{fz } n) =_{\text{fin}} (\text{s } n) (\text{fz } n) \rightsquigarrow \text{Success } [e := \text{eq_refl}] \]

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

⇒ Forced arguments need not be unified: \( \equiv \) by typing.

- In \text{Agda}, justifies unification used in \text{--without-K} mode
- In \text{Equations}, prioritize homogeneous injectivity over heterogeneous injectivity
- Simplification can also use arbitrary UIP proofs (w/o guarantees about computation on open terms).
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 \( \text{ap} \) and the fact that it preserves equivalences. Can solve box-filling problems (e.g. pattern matching on squares of equalities).
- Same notion of path equality for forced arguments but 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).
- Same notion of path equality for forced arguments but more complex proof term.
- Unfortunately, quite incompatible with Coq’s syntactic guardedness check. Equations provides support for well-founded recursion though.
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).
- Same notion of path equality for forced arguments but more complex proof term.
- Unfortunately, quite incompatible with Coq’s syntactic guardedness check. Equations provides support for well-founded recursion though.
- Methodology: Agda relies on metatheoretical proofs (many done internally) while we elaborate to CIC.
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).
- Same notion of path equality for forced arguments but more complex proof term.
- Unfortunately, quite incompatible with \textsc{Coq}'s syntactic guardedness check. \textsc{Equations} provides support for well-founded recursion though.
- Methodology: \textsc{Agda} relies on metatheoretical proofs (many done internally) while we elaborate to CIC.
- Future work: integration in our simplification engine.
In the paper

- New tactic: dependent elimination foo as \([p_1|..|p_n]\)
  Robust naming and ordering of inversions, patterns can use
  notations, axiom-free proof terms
- New tactic: noconf \(H\), combines injection,
  discriminate, subst and acyclicity.
- Nested and mutual well-founded recursion on inductive
  families
- Reasoning principles for definitions by DPM and well-founded
  recursion
Examples

- SN for Predicative System F using hereditary substitution
- CPDT (Chlipala) chapters on dependent pattern-matching (neater programs, shorter proofs using functional eliminators)
- A correct-by-construction, reflexive ring tactic (dependently-typed, canonical polynomials, by Raphaël Bocquet)
- Definitional interpreter extended with mutable references and well-typed store (Poulsen et al. POPL’18, port from Agda)

Reference manual and program gallery available on the website.
Ongoing and Future work

Ongoing work:

▶ IDE support for refinement mode (Proof-General & VSCoq)
▶ Integration with Coq/SSReflect intro-patterns
▶ Support for equality in \textit{SProp} (Gilbert et al., 2019): definitional UIP/K rule, “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 \textit{MetaCoq} (Sozeau et al., 2019)
▶ Link with rewrite rules: dependent pattern-matching and well-founded recursion as a conservative extension of CIC
▶ Extension to co-patterns and co-recursion
Equations Reloaded

http://mattam82.github.io/Coq-Equations/

# opam install coq-equations

Also included in Coq’s Windows packages.


Equations filter \{A\} (l : list A) (p : A \to 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 \( \{A\} \) \((l : \text{list} \ A) \ (p : A \to \text{bool}) : \text{list} \ A := \)
filter \(\text{nil} \ p := \text{nil} \); 
filter \((\text{cons} \ a \ l) \ p \ \text{with} \ p \ a := \{ \)
| true := \(a :: \text{filter} \ l \ p \); 
| false := \text{filter} \ l \ p \};
\)

Check \((\text{filter} \_\text{elim} : \)
\(\forall (P : \forall (A : \text{Type}) \ (l : \text{list} \ A) \ (p : A \to \text{bool}), \text{list} \ A \to \text{Type}), \)
let \(P0 := \text{fun} (A : \text{Type}) (a : A) (l : \text{list} \ A) (p : A \to \text{bool}) \)
\((\text{refine} : \text{bool}) (res : \text{list} \ A) \Rightarrow \)
\(p \ a = \text{refine} \to P \ A (a :: l) \ p \ res \)
in \(\forall (A : \text{Type}) (p : A \to \text{bool}), P \ A \ [] \ p \ [] \) \to 
\(\forall (A : \text{Type}) (a : A) (l : \text{list} \ A) (p : A \to \text{bool}), \)
\(P \ A \ l \ p \ (\text{filter} \ l \ p) \to P0 \ A \ a \ l \ p \ true \ (a :: \text{filter} \ l \ p)) \to 
\forall (A : \text{Type}) (a : A) (l : \text{list} \ A) (p : A \to \text{bool}), \)
\(P \ A \ l \ p \ (\text{filter} \ l \ p) \to P0 \ A \ a \ l \ p \ false \ (\text{filter} \ l \ p)) \to 
\forall (A : \text{Type}) (l : \text{list} \ A) (p : A \to \text{bool}), P \ A \ l \ p \ (\text{filter} \ l \ p)) \to \)
term, type $t, \tau ::= x | \lambda x : \tau, t | \forall x : \tau, \tau' | \lambda \vec{up} := t$

binding $d ::= (x : \tau) | (x := t : \tau)$

context $\Gamma, \Delta ::= \vec{d}$

programs $progs ::= prog \text{ mutual.}$

mutual programs $mutual ::= with p | where$

where clause $where ::= \text{where p | where not}$

notation $not ::= 'string' := term (: scope)?$

program $p, prog ::= f \Gamma : \tau (by annot)? := clauses$

annotation $annot ::= \text{struct x | wf t R}$

clauses $clauses ::= \vec{c} | \{ \vec{c} \}$

user clause $c ::= f \vec{up} n | \vec{up}^+ n$

user pattern $up ::= x | C \vec{up} | ?(t) | (x := up)$

user node $n ::= ::= t \text{ where} | ::=! x$

| with $t \vec{t} := clauses$
context map \( c ::= \Delta \vdash \overrightarrow{p} : \Gamma \)

pattern \( p ::= x \mid C \overrightarrow{p} \mid ?(t) \)

splitting \( spl ::= \text{Split}(c, x, (spl?)^n) \mid \text{Compute}(c'' = \overrightarrow{i''} \hspace{1pt} rhs) \)

node \( rhs ::= t, w \mid \text{Refine}(t, c, \ell, spl) \)

label \( \ell ::= \epsilon \mid \ell.n \quad (n \in \mathbb{N}) \)
Elimination principle: inductive graph

For $f.\ell : \Pi \Delta, f_{\text{comp}} \rightarrow t$ we generate $f.\ell_{\text{ind}} : \Pi \Delta, f_{\text{comp}} \rightarrow t \rightarrow \text{Prop}$ 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 $x : \Pi \Delta, f_{\text{comp}} \rightarrow t$ 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**\((\Delta \vdash \vec{p} : \Gamma'' = \xi'' \text{rhs})\) : By case on \(\text{rhs}\):
  - \(t\) : Compute \(\Psi \vdash t' = \text{ABSREC}(f, t)\) and return the statement
    \[
    \Pi \Delta \Psi \text{HYPS}(\Psi), \ f.\ell_{\text{ind}} \vec{p} \ t'
    \]
  - **Refine**\((t, \Delta' \vdash \vec{v}^x, x, \vec{v}^x : \Delta^x, x : \tau, \Delta_x, \ell.n, s)\) : Compute \(\Psi \vdash t' = \text{ABSREC}(f, t)\) and return:
    \[
    \Pi \Delta \Psi \text{HYPS}(\Psi) \ (\text{res} : f_{\text{comp}} \vec{p})
    \quad f.\ell.n_{\text{ind}} \Delta^x \ t' \Delta_x \text{res} \rightarrow f.\ell_{\text{ind}} \vec{p} \text{res}
    \]

We continue with the generation of the \(f.\ell.n_{\text{ind}}\) graph.
1. Dependent Pattern-Matching and Unification

2. A Short History of DPM

3. A Homogeneous No-Confusion Principle

4. In the Paper and Implementation
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 \( \Pi \Delta, \text{Type} \).
Use **well-founded** recursion on the subterm relation for inductive families $I : \Pi \Delta, \text{Type}$.

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

- Define the subterm relation on telescopes:
  $$I_{\text{sub}} : \text{relation} (\Sigma \Delta, I \Delta)$$
Derive Subterm for vector.
Subterm relation example: vectors

Derive Subterm for vector.

Inductive vector\_strict\_subterm \((A : \text{Type})\)
: \(\forall H H0 : \text{nat}, \text{vector} A H \rightarrow \text{vector} A H0 \rightarrow \text{Prop} \) :=
  \(\text{vector\_strict\_subterm\_1\_1} : \forall (a : A) (n : \text{nat}) (H : \text{vector} A n),\)
  \(\text{vector\_strict\_subterm} A n (S n) H (\text{Vcons} a H).\)

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 (\(A : \text{Type}\))

: \(\forall \ H \ H0 : \text{nat}, \text{vector} \ A \ H \rightarrow \text{vector} \ A \ H0 \rightarrow \text{Prop} \):=

\(\text{vector\_strict\_subterm\_1\_1} : \forall (a : A) (n : \text{nat}) (H : \text{vector} \ A \ n), \text{vector\_strict\_subterm} \ A \ n \ (S \ n) \ H \ (\text{Vcons} \ a \ H)\).

Check \(\text{vector\_subterm} : \forall A : \text{Type}, \text{relation} \ \{\text{index} : \text{nat} \& \text{vector} \ A \ \text{index}\}\).

Equations unzip \(\{A B n\} \ (v : \text{vector} \ (A \times B) n)\)

: \(\text{vector} \ A \ n \times \text{vector} \ B \ n :=\)
unzip \(A B n v\) by rec \(v :=\)
unzip \(A B \ ?(O) \ Vnil := (Vnil, Vnil) ;\)
unzip \(A B \ ?(S \ n) \ (Vcons \ (\text{pair} \ x \ y) \ n \ v)\) with unzip \(v := \{
| (\text{pair} \ xs \ ys) := (Vcons \ x \ xs, \ Vcons \ y \ ys) \}.\)
Outline

1. Dependent Pattern-Matching and Unification
2. A Short History of DPM
3. A Homogeneous No-Confusion Principle
4. In the Paper and Implementation
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 .. 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!