How to Make Ad Hoc Proof Automation Less Ad Hoc

Georges Gonthier[1] Beta Ziliani[2]
Aleks Nanevski[3] Derek Dreyer[2]

[1]Microsoft Research Cambridge

[2]Max Planck Institute for Software Systems (MPI-SWS)

[3]IMDEA Software Institute, Madrid

ICFP 2011, Tokyo

Why proof automation at ICFP?

Ad hoc polymorphism	$\approx$	Overloading terms
Ad hoc proof automation	$\approx$	Overloading lemmas

"How to make ad hoc polymorphism less ad hoc"

- Haskell type classes (Wadler & Blott '89)

"How to make ad hoc proof automation less ad hoc"

- Canonical structures: A generalization of type classes that's already present in Coq

Motivating example from program verification

Lemma noalias:
If pointers x_1 and x_2 appear in disjoint heaps, they do not alias.

In formal syntax:

$$\begin{aligned} \text{noalias} \ : \ & \forall h : \text{heap}. \forall x_1\, x_2 : \text{ptr}. \forall v_1 : A_1. \forall v_2 : A_2. \\ & \quad \text{def} \ \ (\ x_1 \mapsto v_1 \ \uplus \ x_2 \mapsto v_2 \ \uplus \ h\ ) \to \ x_1 \ != x_2 \end{aligned}$$

Motivating example from program verification

Lemma noalias:
If pointers x_1 and x_2 appear in disjoint heaps, they do not alias.

In formal syntax:

$$\begin{array}{l} \text{noalias} : \forall h : \text{heap}. \forall x_1\, x_2 : \text{ptr}. \forall v_1 : A_1. \forall v_2 : A_2. \\ \qquad\qquad \text{def}\ (\underbrace{x_1 \mapsto v_1}_{\text{singleton heaps}} \uplus \underbrace{x_2 \mapsto v_2}_{\text{singleton heaps}} \uplus h) \to x_1\ != x_2 \end{array}$$

Motivating example from program verification

Lemma noalias:
If pointers x_1 and x_2 appear in disjoint heaps, they do not alias.

In formal syntax:

$$\text{noalias} : \forall h : \text{heap}.\ \forall x_1\, x_2 : \text{ptr}.\ \forall v_1 : A_1.\ \forall v_2 : A_2.$$
$$\text{def}\ (x_1 \mapsto v_1 \uplus x_2 \mapsto v_2 \uplus h) \to x_1\ != x_2$$

disjoint union (undefined if heaps overlap)

Motivating example from program verification

Lemma noalias:
If pointers x_1 and x_2 appear in disjoint heaps, they do not alias.

In formal syntax:

$$\text{noalias} : \forall h : \text{heap}.\, \forall x_1\, x_2 : \text{ptr}.\, \forall v_1 : A_1.\, \forall v_2 : A_2.$$
$$\text{def}\ (x_1 \mapsto v_1 \uplus x_2 \mapsto v_2 \uplus h) \to x_1 \mathrel{!=} x_2$$

test for definedness/disjointness

Motivating example from program verification

Lemma noalias:
If pointers x_1 and x_2 appear in disjoint heaps, they do not alias.

In formal syntax:

$$\text{noalias} : \forall h : \text{heap}. \forall x_1\, x_2 : \text{ptr}. \forall v_1 : A_1. \forall v_2 : A_2.$$
$$\text{def}\ (x_1 \mapsto v_1 \uplus x_2 \mapsto v_2 \uplus h) \to x_1 \mathrel{!=} x_2$$

no alias

Using noalias requires a lot of "glue proof"

$$\text{noalias} : \forall\, h\ x_1\ x_2\ v_1\ v_2.\quad \text{def}\ (x_1 \mapsto v_1 \uplus x_2 \mapsto v_2 \uplus h) \to x_1\ != x_2$$

$$\frac{D : \text{def}\ (h_1 \uplus (y_1 \mapsto w_1 \uplus y_2 \mapsto w_2) \uplus (h_2 \uplus y_3 \mapsto w_3))}{(y_1\ != y_2)\ \&\&\ (y_2\ != y_3)\ \&\&\ (y_3\ != y_1)}$$

Using noalias requires a lot of "glue proof"

$$\text{noalias} : \forall\, h\ x_1\ x_2\ v_1\ v_2.$$
$$\qquad \text{def}\ (x_1 \mapsto v_1 \uplus x_2 \mapsto v_2 \uplus h) \to x_1\ != x_2$$

$$\frac{D : \text{def}\ (h_1 \uplus (y_1 \mapsto w_1 \uplus y_2 \mapsto w_2) \uplus (h_2 \uplus y_3 \mapsto w_3))}{(y_1\ != y_2)\ \&\&\ (y_2\ != y_3)\ \&\&\ (y_3\ != y_1)}$$

Using noalias requires a lot of "glue proof"

$$\text{noalias} : \forall\, h\ x_1\ x_2\ v_1\ v_2.$$
$$\text{def}\ (x_1 \mapsto v_1 \uplus x_2 \mapsto v_2 \uplus h) \to x_1\ != x_2$$

$$\frac{D : \text{def}\ (h_1 \uplus (y_1 \mapsto w_1 \uplus y_2 \mapsto w_2) \uplus (h_2 \uplus y_3 \mapsto w_3))}{(y_1\ != y_2)\ \&\&\ (y_2\ != y_3)\ \&\&\ (y_3\ != y_1)}$$

Using noalias requires a lot of "glue proof"

$$\text{noalias} : \forall\, h\ x_1\ x_2\ v_1\ v_2.$$
$$\text{def}\ (x_1 \mapsto v_1 \uplus x_2 \mapsto v_2 \uplus h) \to x_1\ != x_2$$

$$\frac{D : \text{def}\ (h_1 \uplus (y_1 \mapsto w_1 \uplus y_2 \mapsto w_2) \uplus (h_2 \uplus y_3 \mapsto w_3))}{(y_1\ != y_2)\ \&\&\ (y_2\ != y_3)\ \&\&\ (y_3\ != y_1)}$$

Using noalias requires a lot of “glue proof”

$$\text{noalias} : \forall\, h\ x_1\ x_2\ v_1\ v_2.$$
$$\text{def}\ (x_1 \mapsto v_1 \uplus x_2 \mapsto v_2 \uplus h) \to x_1\ != x_2$$

$$\frac{D : \text{def}\ (y_1 \mapsto w_1 \uplus y_2 \mapsto w_2 \uplus (h_1 \uplus (h_2 \uplus y_3 \mapsto w_3)))}{(y_1\ != y_2)\ \&\&\ (y_2\ != y_3)\ \&\&\ (y_3\ != y_1)}$$

Using noalias requires a lot of "glue proof"

$$\text{noalias} : \forall\, h\ x_1\ x_2\ v_1\ v_2.\quad \text{def}\ (x_1 \mapsto v_1 \uplus x_2 \mapsto v_2 \uplus h) \to x_1\ != x_2$$

$$\frac{D : \text{def}\ (y_1 \mapsto w_1 \uplus y_2 \mapsto w_2 \uplus (h_1 \uplus (h_2 \uplus y_3 \mapsto w_3)))}{\text{true}\ \&\&\ (y_2\ != y_3)\ \&\&\ (y_3\ != y_1)}$$

Using noalias requires a lot of "glue proof"

$$\text{noalias} : \forall\, h\ x_1\ x_2\ v_1\ v_2.$$
$$\text{def}\ (x_1 \mapsto v_1 \uplus x_2 \mapsto v_2 \uplus h) \to x_1\ \mathrel{!=}\ x_2$$

$$\frac{D : \text{def}\ (y_1 \mapsto w_1 \uplus y_2 \mapsto w_2 \uplus (h_1 \uplus (h_2 \uplus y_3 \mapsto w_3)))}{\text{true}\ \&\&\ (y_2\ \mathrel{!=}\ y_3)\ \&\&\ (y_3\ \mathrel{!=}\ y_1)}$$

Using noalias requires a lot of "glue proof"

$$\text{noalias} : \forall\, h\ x_1\ x_2\ v_1\ v_2.\quad \text{def}\ (x_1 \mapsto v_1 \uplus x_2 \mapsto v_2 \uplus h) \to x_1\ != x_2$$

$$\dfrac{D : \text{def}\ (y_1 \mapsto w_1 \uplus y_2 \mapsto w_2 \uplus (h_1 \uplus (h_2 \uplus y_3 \mapsto w_3)))}{\text{true}\ \&\&\ (y_2\ != y_3)\ \&\&\ (y_3\ != y_1)}$$

Using noalias requires a lot of "glue proof"

$$\text{noalias} : \forall\, h\ x_1\ x_2\ v_1\ v_2.$$
$$\text{def}\ (x_1 \mapsto v_1 \uplus x_2 \mapsto v_2 \uplus h) \to x_1\ \texttt{!=}\ x_2$$

$$\frac{D : \text{def}\ (y_2 \mapsto w_2 \uplus y_3 \mapsto w_3 \uplus (y_1 \mapsto w_1 \uplus h_1 \uplus h_2))}{\text{true}\ \&\&\ (y_2\ \texttt{!=}\ y_3)\ \&\&\ (y_3\ \texttt{!=}\ y_1)}$$

Using noalias requires a lot of "glue proof"

$$\text{noalias} : \forall\, h\ x_1\ x_2\ v_1\ v_2.\ \ \text{def}\ (x_1 \mapsto v_1 \uplus x_2 \mapsto v_2 \uplus h) \to x_1\ != x_2$$

$$\frac{D : \text{def}\ (y_2 \mapsto w_2 \uplus y_3 \mapsto w_3 \uplus (y_1 \mapsto w_1 \uplus h_1 \uplus h_2))}{\text{true}\ \&\&\ \text{true}\ \&\&\ (y_3\ != y_1)}$$

Using noalias requires a lot of "glue proof"

$$\text{noalias} : \forall\, h\ x_1\ x_2\ v_1\ v_2.\ \quad \text{def}\ (x_1 \mapsto v_1 \uplus x_2 \mapsto v_2 \uplus h) \to x_1\ \texttt{!=}\ x_2$$

$$\frac{D : \text{def}\ (y_2 \mapsto w_2 \uplus y_3 \mapsto w_3 \uplus (y_1 \mapsto w_1 \uplus h_1 \uplus h_2))}{\text{true}\ \&\&\ \text{true}\ \&\&\ (y_3\ \texttt{!=}\ y_1)}$$

Using noalias requires a lot of "glue proof"

$$\text{noalias} : \forall\, h\ x_1\ x_2\ v_1\ v_2.$$
$$\text{def}\ (x_1 \mapsto v_1 \uplus x_2 \mapsto v_2 \uplus h) \rightarrow x_1\ \mathrel{!=}\ x_2$$

$$\frac{D : \text{def}\ (y_2 \mapsto w_2 \uplus y_3 \mapsto w_3 \uplus (y_1 \mapsto w_1 \uplus h_1 \uplus h_2))}{\text{true}\ \&\&\ \text{true}\ \&\&\ (y_3\ \mathrel{!=}\ y_1)}$$

Using noalias requires a lot of “glue proof”

$$\text{noalias} : \forall\, h\ x_1\ x_2\ v_1\ v_2.$$
$$\text{def}\ (x_1 \mapsto v_1 \uplus x_2 \mapsto v_2 \uplus h) \to x_1\ != x_2$$

$$\frac{D : \text{def}\ (y_3 \mapsto w_3 \uplus y_1 \mapsto w_1 \uplus (y_2 \mapsto w_2 \uplus h_1 \uplus h_2))}{\text{true}\ \&\&\ \text{true}\ \&\&\ (y_3\ != y_1)}$$

Using noalias requires a lot of "glue proof"

$$\text{noalias} : \forall\, h\ x_1\ x_2\ v_1\ v_2.$$
$$\text{def}\ (x_1 \mapsto v_1 \uplus x_2 \mapsto v_2 \uplus h) \to x_1\ != x_2$$

$$\frac{D : \text{def}\ (y_3 \mapsto w_3 \uplus y_1 \mapsto w_1 \uplus (y_2 \mapsto w_2 \uplus h_1 \uplus h_2))}{\text{true}\ \&\&\ \text{true}\ \&\&\ \text{true}}$$

Glue proof, formally (in Coq)

rewrite $-!\mathsf{unA}$ $-!(\mathsf{unCA}\ (y_2 \mapsto \_)$ $-!(\mathsf{unCA}\ (y_1 \mapsto \_))$ unA in D.
rewrite (noalias D).
rewrite $-!\mathsf{unA}$ $-$ $(\mathsf{unC}\ (y_3 \mapsto \_))$ $-!(\mathsf{unCA}\ (y_3 \mapsto \_))$ in D.
rewrite $-!(\mathsf{unCA}\ (y_2 \mapsto \_))$ unA in D.
rewrite (noalias D).
rewrite $-!\mathsf{unA}$ $-!(\mathsf{unCA}\ (y_1 \mapsto \_))$ $-!(\mathsf{unCA}\ (y_3 \mapsto \_))$ unA in D.
rewrite (noalias D).

Glue proof, formally (in Coq)

rewrite $-$!unA $-$!(unCA $(y_2 \mapsto \_)$ $-$!(unCA $(y_1 \mapsto \_)$) unA in D.
rewrite (noalias D).
rewrite $-$!unA $-$ (unC $(y_3 \mapsto \_)$) $-$!(unCA $(y_3 \mapsto \_)$) in D.
rewrite $-$!(unCA $(y_2 \mapsto \_)$) unA in D.
rewrite (noalias D).
rewrite $-$!unA $-$!(unCA $(y_1 \mapsto \_)$) $-$!(unCA $(y_3 \mapsto \_)$) unA in D.
rewrite (noalias D).

Automation as it is today

Write custom tactic:

For each x_i != x_j in the goal:

- rearrange hypothesis, to bring x_i and x_j to the front
- apply the noalias lemma
- repeat

Automation as it is today

Write custom tactic:

For each x_i != x_j in the goal:

- rearrange hypothesis, to bring x_i and x_j to the front
- apply the noalias lemma
- repeat

However, custom tactics have several limitations

- Can be untyped or weakly specified
- Automation as a second class citizen

What we want: automated lemmas!

We really want an automated version of the noalias lemma:

$$\text{noaliasA} : \forall \ \ldots ??? \ldots \ x_1 \ != x_2$$

where ??? asks type inference to construct glue proof.

Why?

- Strongly-typed custom automation!
- Composable, modular custom automation!

Using and composing automated lemmas

$$(y_1 \;!\!= y_2) \;\&\&\; (y_2 \;!\!= y_3) \;\&\&\; (y_3 \;!\!= y_1)$$

Using and composing automated lemmas

$$(y_1 \ \texttt{!=}\ y_2) \ \&\& \ (y_2 \ \texttt{!=}\ y_3) \ \&\& \ (y_3 \ \texttt{!=}\ y_1)$$

$$\downarrow$$

true && true && true

Using and composing automated lemmas

$$(y_1 \ != y_2) \ \&\& \ (y_2 \ != y_3) \ \&\& \ (y_3 \ != y_1)$$

$$\downarrow$$

$$\mathsf{true} \ \&\& \ \mathsf{true} \ \&\& \ \mathsf{true}$$

by performing

<u>rewrite</u> ! (noaliasA D)

Using and composing automated lemmas

$$(y_1 \; \texttt{!=} \; y_2) \;\&\&\; (y_2 \; \texttt{!=} \; y_3) \;\&\&\; (y_3 \; \texttt{!=} \; y_1)$$

$$\downarrow$$

$$\text{true} \;\&\&\; \text{true} \;\&\&\; \text{true}$$

by performing

$$\underline{\text{rewrite}} \;\; ! \; (\text{noaliasA} \; D)$$

multiple times

Using and composing automated lemmas

$$(y_1 \;\texttt{==}\; y_2) \;\&\&\; (y_2 \;\texttt{!=}\; y_3) \;\&\&\; (y_3 \;\texttt{!=}\; y_1)$$

Using and composing automated lemmas

$$(y_1 \;\texttt{==}\; y_2) \;\&\&\; (y_2 \;\texttt{!=}\; y_3) \;\&\&\; (y_3 \;\texttt{!=}\; y_1)$$

$$\downarrow$$

$$\text{false} \;\&\&\; (y_2 \;\texttt{!=}\; y_3) \;\&\&\; (y_3 \;\texttt{!=}\; y_1) = \text{false}$$

Using and composing automated lemmas

$$(y_1 \ \texttt{==} \ y_2) \ \&\& \ (y_2 \ \texttt{!=} \ y_3) \ \&\& \ (y_3 \ \texttt{!=} \ y_1)$$

$$\downarrow$$

$$\text{false} \ \&\& \ (y_2 \ \texttt{!=} \ y_3) \ \&\& \ (y_3 \ \texttt{!=} \ y_1) = \text{false}$$

by performing

$$\underline{\text{rewrite}} \ (\text{negate} \ (\text{noaliasA} \ D))$$

where

$$\text{negate} \ : \ \forall b : \text{bool}. \, !b = \text{true} \to b = \text{false}$$

How? Lemma automation by overloading

Curry-Howard correspondence!

- Overloading:
  infer code for a function based on arguments
- Lemma overloading:
  infer proof for a lemma based on arguments

Our Main Contributions

Idea: proof automation through lemma overloading

Realizing this idea by Coq's canonical structures:

- A generalization of Haskell type classes
- Instances pattern-match terms as well as types

"Design patterns" for controlling Coq type inference

- Several interesting examples from HTT

Our Main Contributions

Idea: proof automation through lemma overloading

Realizing this idea by Coq's canonical structures:

- A generalization of Haskell type classes
- Instances pattern-match terms as well as types

"Design patterns" for controlling Coq type inference

- Several interesting examples from HTT

See the paper for details!

Haskell overloading: equality type class

$\underline{\text{class}}$ Eq a $\underline{\text{where}}$

$\quad (==) :: a \to a \to \text{Bool}$

$\underline{\text{instance}}$ Eq Bool $\underline{\text{where}}$

$\quad (==) = \lambda x\ y.\,(x\ \&\&\ y)\ ||\ (!x\ \&\&\ !y)$

$\underline{\text{instance}}$ (Eq a, Eq b) $\Rightarrow$ Eq $(a \times b)$ $\underline{\text{where}}$

$\quad (==) = \lambda x\ y.\,(\text{fst}\ x == \text{fst}\ y)\ \&\&\ (\text{snd}\ x == \text{snd}\ y)$

Haskell overloading: equality type class

$\underline{\text{class}}$ Eq a $\underline{\text{where}}$

$\quad (==) :: a \to a \to \text{Bool}$

$\underline{\text{instance}}$ Eq Bool $\underline{\text{where}}$

$\quad (==) = \text{eq\_bool}$

$\underline{\text{instance}}$ $(f_1 : \text{Eq } a,\ f_2 : \text{Eq } b) \Rightarrow \text{Eq } (a \times b)$ $\underline{\text{where}}$

$\quad (==) = \text{eq\_pair } f_1\ f_2$

Haskell overloading: equality type class

Example:

$$(x, \mathsf{true}) \;\; \texttt{==} \;\; (\mathsf{false}, y)$$

$\underline{\mathsf{class}}$ $\mathsf{Eq}\ a$ $\underline{\mathsf{where}}$
  $(\texttt{==}) :: a \to a \to \mathsf{Bool}$

$\underline{\mathsf{instance}}$ $\mathsf{Eq\ Bool}$ $\underline{\mathsf{where}}$
  $(\texttt{==}) = \mathsf{eq\_bool}$

$\underline{\mathsf{instance}}$ $(f_1 : \mathsf{Eq}\ a,\ f_2 : \mathsf{Eq}\ b) \Rightarrow \mathsf{Eq}\ (a \times b)$ $\underline{\mathsf{where}}$
  $(\texttt{==}) = \mathsf{eq\_pair}\ f_1\ f_2$

Haskell overloading: equality type class

Example:

$$\underset{\text{eq\_pair eq\_bool eq\_bool}}{(x, \text{true}) \underbrace{==} (\text{false}, y)}$$

$\underline{\text{class}}$ Eq a $\underline{\text{where}}$

$\quad (==) :: a \to a \to \text{Bool}$

$\underline{\text{instance}}$ Eq Bool $\underline{\text{where}}$

$\quad (==) = \text{eq\_bool}$

$\underline{\text{instance}}$ $(f_1 : \text{Eq } a,\ f_2 : \text{Eq } b) \Rightarrow \text{Eq } (a \times b)$ $\underline{\text{where}}$

$\quad (==) = \text{eq\_pair } f_1\ f_2$

Coq overloading: equality type class

Coq structure: just a dependent record type

$$\underline{\text{structure}}\ \overbrace{\text{Eq}}^{\text{name}}\ :=$$

$$\overbrace{\text{mkEq}}^{\text{constructor}}\ \overbrace{\{\text{sort} : \text{Type}; \quad (\_ == \_) : \text{sort} \to \text{sort} \to \text{bool}; \}}^{\text{fields}}$$

Coq overloading: equality type class

Coq structure: just a dependent record type

$$\underline{\text{structure}}\ \overbrace{\text{Eq}}^{\text{name}}\ :=$$

$$\overbrace{\text{mkEq}}^{\text{constructor}}\ \ \overbrace{\{\text{sort} : \text{Type};}^{\text{fields}}$$

$$(\_ == \_) : \text{sort} \to \text{sort} \to \text{bool}; \}$$

Coq overloading: equality type class

Coq structure: just a dependent record type

$$\underline{\text{structure}}\ \overbrace{\text{Eq}}^{\text{name}}\ :=$$

$$\overbrace{\text{mkEq}}^{\text{constructor}}\quad \overbrace{\{\text{sort} : \text{Type}; \ (\_ == \_) : \text{sort} \to \text{sort} \to \text{bool}; \}}^{\text{fields}}$$

Creates projectors for each field, e.g.:

$$\begin{array}{lll} \text{sort} & : & \text{Eq} \to \text{Type} \\ (\_ == \_) & : & \forall e : \text{Eq}.\ \text{sort}\ e \to \text{sort}\ e \to \text{bool} \end{array}$$

Canonical instances

Instances defined as in Haskell

$$\underline{\text{canonical}}\ \text{bool\_inst} := \text{mkEq}\ \overbrace{\text{bool}}^{\text{sort}}\ \overbrace{\text{eq\_bool}}^{(\_ == \_)}$$

$$\underline{\text{canonical}}\ \text{pair\_inst}\ (A\ B : \text{Eq}) :=$$
$$\text{mkEq}\ \underbrace{(\text{sort}\ A \times \text{sort}\ B)}_{\text{sort}}\ \underbrace{(\text{eq\_pair}\ A\ B)}_{(\_ == \_)}$$

Example of instance inference

$$(x, \text{true}) == (\text{false}, y)$$

Example of instance inference

$$(x, \mathsf{true}) == (\mathsf{false}, y)$$

Remember

$$(\_ == \_) \;:\; \forall\, e : \mathsf{Eq}\;.\;\; \mathsf{sort}\; e \to \mathsf{sort}\; e \to \mathsf{bool}$$

Example of instance inference

$$(x, \mathsf{true}) == (\mathsf{false}, y)$$

Remember

$$(\_ == \_) \ : \ \forall\ e : \mathsf{Eq}\ .\ \mathsf{sort}\ e \to \mathsf{sort}\ e \to \mathsf{bool}$$

implicit parameter

Example of instance inference

$$(x, \mathsf{true}) == (\mathsf{false}, y)$$

Remember

$$(\_ == \_) \; : \; \forall\, e : \mathsf{Eq}\; .\; \mathsf{sort}\; e \rightarrow \mathsf{sort}\; e \rightarrow \mathsf{bool}$$

Coq finds an instance of $e : \mathsf{Eq}$ that unifies

$$\mathsf{sort}\; e \quad \text{with} \quad (\mathsf{bool} \times \mathsf{bool})$$

Example of instance inference

$$(x, \mathsf{true}) == (\mathsf{false}, y)$$

Remember

$$(\_ == \_) \; : \; \forall\, e : \mathsf{Eq}\ .\ \ \mathsf{sort}\ e \to \mathsf{sort}\ e \to \mathsf{bool}$$

Coq finds an instance of $e : \mathsf{Eq}$ that unifies

$$\mathsf{sort}\ e \quad \text{with} \quad (\mathsf{bool} \times \mathsf{bool})$$

$$\downarrow$$

$$e = \mathsf{pair\_inst}\ \mathsf{bool\_inst}\ \mathsf{bool\_inst}$$

Adding a proof

We add the proof that $(\_ == \_)$ is equivalent to Coq's $(\_ = \_)$

$$\underline{\text{structure}}\ \overbrace{\text{Eq}}^{\text{name}}\ :=$$

$$\overbrace{\text{mkEq}}^{\text{constructor}}\quad \overbrace{\{\text{sort} : \text{Type}; \quad (\_ == \_) : \text{sort} \to \text{sort} \to \text{bool}; \quad \text{proof} : \forall x\ y : \text{sort}.\ x == y \leftrightarrow x = y\}}^{\text{fields}}$$

Adding a proof

We add the proof that $(\_ == \_)$ is equivalent to Coq's $(\_ = \_)$

$$\underline{\text{structure}}\ \overbrace{\text{Eq}}^{\text{name}}\ :=$$

$$\overbrace{\text{mkEq}}^{\text{constructor}}\ \overbrace{\{\text{sort} : \text{Type};\ (\_ == \_) : \text{sort} \to \text{sort} \to \text{bool};\ \text{proof} : \forall x\ y : \text{sort}.\ x == y \leftrightarrow x = y\}}^{\text{fields}}$$

Now instances also compute the proof:

$\underline{\text{canonical}}$ bool_inst := mkEq bool eq_bool pf_bool

$\underline{\text{canonical}}$ pair_inst $(A\ B : \text{Eq})$:=

mkEq (sort A $\times$ sort B) (eq_pair A B) (pf_pair A B)

Lemma overloading

Overloading a simple lemma

Goal: Prove x is in the domain of

$$\cdots \uplus (\cdots \uplus x \mapsto v \uplus \cdots) \uplus \cdots$$

Overloading a simple lemma

Goal: Prove x is in the domain of

$$\cdots \uplus (\cdots \uplus x \mapsto v \uplus \cdots) \uplus \cdots$$

Naïve lemma:

$$\begin{aligned} \text{indom} \quad : \quad & \forall x : \text{ptr}. \forall v : A. \forall h : \text{heap}. \\ & x \in \text{dom}\ (x \mapsto v \uplus h) \end{aligned}$$

Overloading a simple lemma

Goal: Prove x is in the domain of

$$\cdots \uplus (\cdots \uplus x \mapsto v \uplus \cdots) \uplus \cdots$$

Naïve lemma:

$$\begin{aligned} \mathsf{indom} \quad : \quad & \forall x : \mathsf{ptr}.\, \forall v : A.\, \forall h : \mathsf{heap}. \\ & x \in \mathsf{dom}\ (x \mapsto v \uplus h) \end{aligned}$$

Let's overload it!

indom overloaded

Define structure contains x, of heaps that contain x:

$$\underline{\text{structure}}\ \text{contains}\ (x : \text{ptr}) :=$$
$$\text{Contains}\ \{\ \text{heap\_of} : \text{heap};$$
$$\text{indomO} : x \in \text{dom heap\_of}\ \}$$

indom overloaded

Define structure contains x, of heaps that contain x:

$$\underline{\text{structure}}\ \text{contains}\ (x : \text{ptr}) := \\ \text{Contains}\ \{\ \text{heap\_of} : \text{heap}; \\ \text{indomO} : x \in \text{dom heap\_of}\ \}$$

Induced projections:

$$\text{heap\_of} : \forall x : \text{ptr}.\ \text{contains}\ x \to \text{heap}$$

$$\text{indomO} : \forall x : \text{ptr}.\, \forall c : \text{contains}\ x.\ \ x \in \text{dom}\ (\text{heap\_of}\ c)$$

The second one is our overloaded lemma

indomO algorithm, informally

When solving

$$x \in \mathrm{dom}\ h$$

indomO algorithm, informally

When solving

$$x \in \mathrm{dom}\ h$$

type inference should proceed as follows:

- If h is $x \mapsto v$, succeed with

$$\mathrm{singleton\_pf}\ :\ \forall x\ v.\ \ x \in \mathrm{dom}\ (x \mapsto v)$$

indomO algorithm, informally

When solving

$$x \in \mathrm{dom}\ h$$

type inference should proceed as follows:

- If h is $x \mapsto v$, succeed with

$$\mathrm{singleton\_pf}\ :\ \forall x\ v.\ x \in \mathrm{dom}\ (x \mapsto v)$$

- If h is $h_1 \uplus h_2$:
  - If $x \in \mathrm{dom}\ h_1$, compose with

$$\mathrm{left\_pf}\ :\ \forall h_1\ h_2.\ x \in \mathrm{dom}\ h_1 \to x \in \mathrm{dom}\ (h_1 \uplus h_2)$$

indomO algorithm, informally

When solving

$$x \in \mathsf{dom}\ h$$

type inference should proceed as follows:

- If h is $x \mapsto v$, succeed with

  $$\mathsf{singleton\_pf} \ : \ \forall x\ v.\ \ x \in \mathsf{dom}\ (x \mapsto v)$$

- If h is $h_1 \uplus h_2$:

  - If $x \in \mathsf{dom}\ h_1$, compose with

    $$\mathsf{left\_pf} \ : \ \forall h_1\ h_2.\ \ x \in \mathsf{dom}\ h_1 \to x \in \mathsf{dom}\ (h_1 \uplus h_2)$$

  - If $x \in \mathsf{dom}\ h_2$, compose with

    $$\mathsf{right\_pf} \ : \ \forall h_1\ h_2.\ \ x \in \mathsf{dom}\ h_2 \to x \in \mathsf{dom}\ (h_1 \uplus h_2)$$

Implementation of indomO

Algorithm encoded in canonical instances of contains x:

canonical found A x $(v : A)$:=

Contains x $(x \mapsto v)$ singleton_pf

canonical left x h $(c :$ contains $x)$:=

Contains x ((heap_of c) $\uplus$ h) (left_pf (indomO c))

canonical right x h $(c :$ contains $x)$:=

Contains x (h $\uplus$ (heap_of c)) (right_pf (indomO c))

Implementation of indomO

Algorithm encoded in canonical instances of contains x:

canonical found A x $(v : A)$:=

Contains x $(x \mapsto v)$ singleton_pf

canonical left x h $(c :$ contains $x)$:=

Contains x ((heap_of c) $\uplus$ h) (left_pf (indomO c))

canonical right x h $(c :$ contains $x)$:=

Contains x (h $\uplus$ (heap_of c)) (right_pf (indomO c))

Implementation of indomO

Algorithm encoded in canonical instances of contains x:

As a logic program

canonical found A x $(v : A)$:=

Contains x $(x \mapsto v)$ singleton_pf

canonical left x h (c : contains x) :=

Contains x ((heap_of c) $\uplus$ h) (left_pf (indomO c))

canonical right x h (c : contains x) :=

Contains x (h $\uplus$ (heap_of c)) (right_pf (indomO c))

Implementation of indomO

Algorithm encoded in canonical instances of contains x:

Canonical structures $\approx$ term classes

canonical found A x $(v : A)$:=

Contains x $(x \mapsto v)$ singleton_pf

canonical left x h $(c$: contains $x)$:=

Contains x ((heap_of c) $\uplus$ h) (left_pf (indomO c))

canonical right x h $(c$: contains $x)$:=

Contains x (h $\uplus$ (heap_of c)) (right_pf (indomO c))

Example application of indomO

$$\mathsf{indomO} \;:\; \forall x : \mathsf{ptr}.\, \forall c : \mathsf{contains}\ x.\;\; x \in \mathsf{dom}\ (\mathsf{heap\_of}\ c)$$

$$y \in \mathsf{dom}\ (z \mapsto u \uplus y \mapsto v)$$

Example application of indomO

$$\text{indomO} \;:\; \forall x : \text{ptr}.\, \forall c : \text{contains } x.\;\; x \in \text{dom } (\text{heap\_of } c)$$

$$y \in \text{dom } (z \mapsto u \uplus y \mapsto v)$$

Solve it by apply indomO

Example application of indomO

$$\text{indomO} \;:\; \forall x : \text{ptr}.\, \forall c : \text{contains}\ x.\;\; x \in \text{dom}\ (\text{heap\_of}\ \ c)$$

$$y \in \text{dom}\ (z \mapsto u \uplus y \mapsto v)$$

Solve it by apply indomO

Example application of indomO

$$\text{indomO} \;:\; \forall x : \text{ptr}.\, \forall c : \text{contains } x.\;\; x \in \text{dom } (\text{heap\_of } c)$$

$$y \in \text{dom } (z \mapsto u \uplus y \mapsto v)$$

Solve it by apply indomO , unifying:

x with y

Example application of indomO

$$\text{indomO} \;:\; \forall x : \text{ptr}.\, \forall c : \text{contains}\ x.\;\; x \in \text{dom}\ (\text{heap\_of}\ \ c)$$

$$y \in \text{dom}\ (z \mapsto u \uplus y \mapsto v)$$

Solve it by apply indomO , unifying:

x with y heap_of c with $(z \mapsto u \uplus y \mapsto v)$

Example application of indomO

$$\text{indomO} \;:\; \forall x : \text{ptr}.\, \forall c : \text{contains } x.\;\; x \in \text{dom } (\text{heap\_of } c)$$

$$y \in \text{dom } (z \mapsto u \uplus y \mapsto v)$$

Solve it by apply indomO , unifying:

x with y heap_of c with $(z \mapsto u \uplus y \mapsto v)$

Result:

$$c = \text{right } y\ (z \mapsto u)\ (\text{found } y\ v)$$

Example application of indomO

$$\text{indomO} \;:\; \forall x : \text{ptr}.\, \forall c : \text{contains}\ x.\;\; x \in \text{dom}\ (\text{heap\_of}\ \ c)$$

$$y \in \text{dom}\ (z \mapsto u \uplus y \mapsto v)$$

Solve it by apply indomO , unifying:

x with y heap_of c with $(z \mapsto u \uplus y \mapsto v)$

Result:

$$c = \text{right}\ y\ (z \mapsto u)\ (\text{found}\ y\ \ v)$$

Example application of indomO

$$\text{indomO} \;:\; \forall x : \text{ptr}.\, \forall c : \text{contains } x.\;\; x \in \text{dom } (\text{heap\_of } c)$$

$$y \in \text{dom } (z \mapsto u \uplus y \mapsto v)$$

Solve it by apply indomO , unifying:

x with y $\qquad$ heap_of c with $(z \mapsto u \uplus y \mapsto v)$

Result:

$$c = \text{right } y\ (z \mapsto u)\ (\text{found } y\ v)$$

The truth revealed

Overlapping instances not allowed in Coq!

$\underline{\text{canonical}}$ left x h (c : contains x) :=

Contains x ((heap_of c) $\uplus$ h) (left_pf (indomO c))

$\underline{\text{canonical}}$ right x h (c : contains x) :=

Contains x (h $\uplus$ (heap_of c)) (right_pf (indomO c))

The truth revealed

Overlapping instances not allowed in Coq!

$\boxed{\text{“Tagging” pattern for instance disambiguation!}}$

canonical left x h (c : contains x) :=
  Contains x ((heap_of c) $\uplus$ h) (left_pf (indomO c))

canonical right x h (c : contains x) :=
  Contains x (h $\uplus$ (heap_of c)) (right_pf (indomO c))

What else is in the paper

"Tagging" pattern for instance disambiguation

Example of proof by reflection

"Hoisting" pattern for ordering unification subproblems

"Search-and-replace" pattern for structural in-place-update

Conclusions

Proof automation by lemma overloading

Curry-Howard correspondence!

- Overloading:
  infer code for a function based on arguments
- Lemma overloading:
  infer proof for a lemma based on arguments

Robust, verifiable, composable automation routines

Questions?

Comparison with Coq Type Classes

Coq Type Classes (CTC) [Sozeau and Oury, TPHOLs '08]

- Similar to canonical structures, which predated them
- Instance resolution for CTC guided by proof search, rather than by Coq unification
- They're in beta and it's not my fault!
- Overlapping instances resolved by weighted backtracking

We've ported a number of our examples to CTC

- Could not figure out how to port "search-and-replace" pattern
- Sometimes CTC is faster, sometimes CS is faster
- Further investigation of the tradeoffs is needed
- Future work: unify the two concepts?

A word on performance

Performance for lemma overloading currently not great:

- Time to perform a simple assignment to a unification variable is quadratic in the number of variables in the context, and linear in the size of the term being assigned
- With tactics, it's nearly constant-time

Clearly a bug in the implementation of Coq unification:

- Not always a problem, since interactive proofs often keep variable contexts short
- But it needs to be fixed...

Solution: The “Tagging” Pattern

Present different constants for each instance:

$\underline{\text{canonical}}\ \text{found}\ A\ x\ (v : A) := \text{Contains}\ x\ (\text{found\_tag}\ (x \mapsto v))\ \ldots$

$\underline{\text{canonical}}\ \text{left}\ x\ h\ (c : \text{contains}\ x)\ :=$
$\quad \text{Contains}\ x\ (\text{left\_tag}\ ((\text{heap\_of}\ f) \uplus h))\ \ldots$

$\underline{\text{canonical}}\ \text{right}\ x\ h\ (c : \text{contains}\ x)\ :=$
$\quad \text{Contains}\ x\ (\text{right\_tag}\ (h \uplus (\text{heap\_of}\ f)))\ \ldots$

No overlap anymore! Where:

$$\text{found\_tag} = \text{left\_tag} = \text{right\_tag}$$

Lazy unification algorithm unrolls them upon matching failure!

The Tagging Pattern

We employ the "tagging" design pattern to disambiguate instances.

- Rely on lazy expansion of constant definitions by the unification algorithm.

$$\underline{\mathsf{structure}}\ \mathsf{tagged\_heap} := \mathsf{Tag}\ \{\mathsf{untag} : \mathsf{heap}\}$$

$$\left.\begin{array}{l} \mathsf{right\_tag}\ h := \mathsf{Tag}\ h \\ \mathsf{left\_tag}\ h := \mathsf{right\_tag}\ h \\ \underline{\mathsf{canonical}}\ \mathsf{found\_tag}\ h := \mathsf{left\_tag}\ h \end{array}\right\}\ \text{all synonyms of } \mathsf{Tag}$$

- One synonym of Tag for each canonical instance of find x
- Listed in the reverse order in which we want them to be considered during pattern matching
- Last one marked as a canonical instance of tagged_heap

The Tagging Pattern

And we tag each canonical instance of find x accordingly:

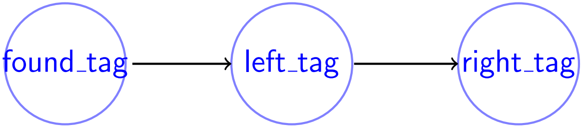
canonical found A x $(v : A)$:= Contains x (found_tag $(x \mapsto v)$) ...

canonical left x h $(f' :$ contains $x)$:=
    Contains x (left_tag (untag (heap_of f') $\uplus$ h)) ...

canonical right x h $(f'' :$ contains $x)$:=
    Contains x (right_tag (h $\uplus$ untag (heap_of f''))) ...

No overlap! Each instance has a different constant.

Example

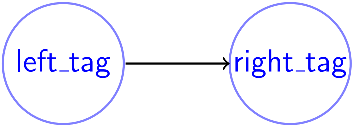


Where f : contains y

$$\mathrm{heap\_of}\ ?f \mathrel{\widehat{=}} \mathrm{found\_tag}\ (x \mapsto u \uplus y \mapsto v)$$

Try instance found and fail

Example



Where f : contains y

$$\text{heap\_of } ?f \;\widehat{=}\; \text{left\_tag } (x \mapsto u \uplus y \mapsto v)$$

Try instance left and fail

Example

right_tag

Where f : contains y

$$\text{heap\_of } ?f \;\widehat{=}\; \text{right\_tag } (x \mapsto u \uplus y \mapsto v)$$

Try instance right and succeed

Overloaded cancellation lemma for heaps

Applying lemma cancelO on a heap equation

$$x \mapsto v_1 \uplus (h_3 \uplus h_4) = h_4 \uplus x \mapsto v_2$$

cancels common terms to produce

$$v_1 = v_2 \wedge h_3 = \emptyset$$

Steps:

1. Logic program turn equations into abstract syntax trees
   - Executed during type inference by unification engine
   - Equal variables turn into equal natural indices
2. Functional program cancels common terms
3. Functional program translate back into equations

Requires only the tagging pattern

Overloaded version of noalias

$$\text{noalias} : \forall h{:}\text{heap}.\, \forall x_1\, x_2{:}\text{ptr}.\, \forall v_1{:}A_1.\, \forall v_2{:}A_2.$$
$$\text{def}(x_1 \mapsto v_1 \uplus x_2 \mapsto v_2 \uplus h) \to x_1 \mathrel{!=} x_2$$

$$\text{noaliasO} :$$
$$\forall x\; y : \text{ptr}.\, \forall s : \text{seq ptr}.\, \forall f : \text{scan } s.\, \forall g : \text{check } x\; y\; s.$$
$$\text{def } (\text{heap\_of } f) \to x \mathrel{!=} (\text{y\_of } g)$$

- Requires two recursive logic programs
  - scan traverses a heap collecting all pointers into a list s
  - then check traverses s searching for x and y
- Somewhat tricky to pass arguments from scan to check
- Employs the hoisting pattern to reorder unification subproblems.

Search-and-replace pattern

Useful lemma for verifying "Hoare triples":

$$\mathsf{bnd\_write} : \mathsf{verify}\ \overbrace{(x \mapsto v \uplus h)}^{\text{initial heap}}\ e\ \overbrace{q}^{\text{post-condition}} \to$$
$$\mathsf{verify}\ (x \mapsto w \uplus h)\ \underbrace{(\mathsf{write}\ x\ v; e)}_{\text{write } v \text{ in location } x \text{ and then run } e}\ q$$

But we'd like to do "in-place update" on the initial heap, rather than shifting $x \mapsto ?$ to the front of the heap and then back again.

Search-and-replace pattern: example

Example 1: To prove the goal

$$G : \mathsf{verify}\ (h_1 \uplus (x_1 \mapsto 1 \uplus x_2 \mapsto 2))\ (\mathsf{write}\ x_2\ 4; e)\ q$$

we can apply bnd_writeO to reduce it to:

$$G : \mathsf{verify}\ (h_1 \uplus (x_1 \mapsto 1 \uplus x_2 \mapsto 4))\ e\ q$$

Search-and-replace pattern: idea

Build a logic program that turns a heap into a function that abstracts the wanted pointer.

Example: Turn $h_1 \uplus (x_1 \mapsto 1 \uplus x_2 \mapsto 2)$ into

$$f = \text{fun } k.\, h_1 \uplus (x_1 \mapsto 1 \uplus k)$$

Then the bnd_writeO lemma can be stated roughly as

$$\begin{array}{l} \text{verify } (f\ (x \mapsto v))\ e\ q \to \\ \text{verify } (f\ (x \mapsto w))\ (\text{write } x\ v; e)\ q \end{array}$$

Search-and-replace pattern: more formally

It turns out that f must have a dependent function type.

$$\underline{\text{structure}}\ \text{partition}\ (k\ r : \text{heap}) := \text{Partition}\ \{\text{heap\_of} : \text{tagged\_heap};\ \_ : \text{heap\_of} = k \uplus r\}$$

$$\text{bnd\_writeO} : \forall r : \text{heap}.\, \forall f : (\Pi k : \text{heap}.\, \text{partition}\ k\ r).\, \forall \ldots$$
$$\text{verify}\ (\text{untag}\ (\text{heap\_of}\ (f\ (x \mapsto v))))\ e\ q \to$$
$$\text{verify}\ (\text{untag}\ (\text{heap\_of}\ (f\ (x \mapsto w))))\ (\text{write}\ x\ v; e)\ q$$

Search-and-replace pattern: forward reasoning

Example 2: Given hypothesis

$$H : \text{verify } (h_1 \uplus (x_1 \mapsto 1 \uplus x_2 \mapsto 4))\ e\ q$$

we can apply $(\text{bnd\_writeO } (x := x_2)\ (w := 2))$ to it:

$$H : \text{verify } (h_1 \uplus (x_1 \mapsto 1 \uplus x_2 \mapsto 2))\ (\text{write } x_2\ 4; e)\ q$$

Note: this duality of use is not possible with tactics