How to Make Ad Hoc Proof Automation Less Ad Hoc

Derek Dreyer

Max Planck Institute for Software Systems
(MPI-SWS)

joint work with Georges Gonthier (MSRC),
Beta Ziliani (MPI-SWS) and Aleks Nanevski (IMDEA)

Seoul National University, 15 September 2011

Applying ideas from FP to proof automation

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:

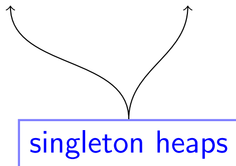
$$\text{noalias} : \forall h{:}\text{heap}.\ \forall x_1\ x_2{:}\text{ptr}.\ \forall v_1{:}A_1.\ \forall v_2{:}A_2.$$
$$\text{def}\ \ (\ x_1 \mapsto v_1\ \ \uplus\ \ x_2 \mapsto v_2\ \ \uplus\ \ h\ ) \to\ x_1\ != x_2$$

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:

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



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:

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

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:

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

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"

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

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

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"

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

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

Using noalias requires a lot of "glue proof"

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

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

Using noalias requires a lot of "glue proof"

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

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

Using noalias requires a lot of "glue proof"

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

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

Using noalias requires a lot of "glue proof"

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

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

Using noalias requires a lot of "glue proof"

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

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

Using noalias requires a lot of "glue proof"

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

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

Using noalias requires a lot of "glue proof"

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

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

Using noalias requires a lot of "glue proof"

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

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

Using noalias requires a lot of "glue proof"

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

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

Using noalias requires a lot of "glue proof"

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

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

Glue proof, formally (in Coq)

```
rewrite -!unA -!(unCA (y2 ↦ _) -!(unCA (y1 ↦ _)) unA in D.
rewrite (noalias D).
rewrite -!unA - (unC (y3 ↦ _)) -!(unCA (y3 ↦ _)) in D.
rewrite -!(unCA (y2 ↦ _)) unA in D.
rewrite (noalias D).
rewrite -!unA -!(unCA (y1 ↦ _)) -!(unCA (y3 ↦ _)) unA in D.
rewrite (noalias D).
```

Glue proof, formally (in Coq)

```
rewrite -!unA -!(unCA (y₂ ↦ _) -!(unCA (y₁ ↦ _)) unA in D.
rewrite (noalias D).
rewrite -!unA  - (unC (y₃ ↦ _)) -!(unCA (y₃ ↦ _)) in D.
rewrite -!(unCA (y₂ ↦ _)) unA in D.
rewrite (noalias D).
rewrite -!unA -!(unCA (y₁ ↦ _)) -!(unCA (y₃ ↦ _)) 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!
- More flexible and composable!

Using and composing automated lemmas

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

Using and composing automated lemmas

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

$$\downarrow$$

true && true && true

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

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

Using and composing automated lemmas

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

$$\downarrow$$

true && true && true

by performing

rewrite ! (noaliasA D)

Using and composing automated lemmas

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

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

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

$$\downarrow$$

$$\mathsf{false} \mathbin{\&\&} (y_2 \mathrel{!=} y_3) \mathbin{\&\&} (y_3 \mathrel{!=} y_1) = \mathsf{false}$$

by performing

$$\underline{\mathsf{rewrite}}\ (\mathsf{negbTE}\ (\mathsf{noaliasA}\ D))$$

where

$$\mathsf{negbTE} : \forall b{:}\mathsf{bool}.\, !b = \mathsf{true} \to b = \mathsf{false}$$

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

Canonical structures 101

Haskell overloading: equality type class

Use the same name for two differently-typed functions:

- x == true
- (x, false) == (true, y)

Implementation:

```
class Eq a where
  (==) :: a → a → Bool

instance Eq Bool where
  x == y = (x && y) || (!x && !y)

instance (Eq a, Eq b) ⇒ Eq (a, b) where
  x == y = (fst x == fst y) && (snd x == snd y)
```

Overloading in Coq: structure definitions

Equality class is a dependent record type (a.k.a. structure):

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

$$\overbrace{\text{EqType}}^{\text{constructor}}\ \overbrace{\{\text{sort} : \text{Type}; \quad \text{equal} : \text{sort} \to \text{sort} \to \text{bool}; \quad _ : \forall x\ y : \text{sort}.\, \text{equal}\ x\ y \leftrightarrow x = y\}}^{\text{fields}}$$

Induces the creation of several projection functions as well:

$$\text{sort} \quad : \quad \text{eqType} \to \text{Type}$$
$$\text{equal} \quad : \quad \forall T\text{:eqType}.\ \text{sort}\ T \to \text{sort}\ T \to \text{bool}$$

Overloading in Coq: implicit arguments

$$\text{equal} : \forall T{:}\text{eqType}.\ \text{sort}\ T \rightarrow \text{sort}\ T \rightarrow \text{bool}$$

We write $x == y$ as a syntactic sugar for

$$\text{equal}\ x\ y$$

where $x, y : \text{sort}\ T$.

Overloading in Coq: implicit arguments

$$\text{equal} : \forall T{:}\text{eqType}.\ \ \text{sort}\ \ T \rightarrow \text{sort}\ \ T \rightarrow \text{bool}$$

We write $x == y$ as a syntactic sugar for

$$\text{equal}\ ?T\ x\ y$$

where x, y : sort $?T$.

The argument $?T$ is implicit. It is omitted in source code.

$?T$ is also a unification variable, resolved during type inference.

Canonical instances

$$\underline{\text{canonical}}\ \text{eqType_bool} := \text{EqType}\ \overbrace{\text{bool}}^{\text{sort}}\ \overbrace{\text{eq_bool}}^{\text{equal}}\ \overbrace{\text{pf_bool}}^{\text{proof}}$$

$$\underline{\text{canonical}}\ \text{eqType_pair}\ (A\ B : \text{eqType}) :=$$
$$\text{EqType}\ \underbrace{(\text{sort}\ A \times \text{sort}\ B)}_{\text{sort}}\ \underbrace{(\text{eq_pair}\ A\ B)}_{\text{equal}}\ \underbrace{(\text{pf_pair}\ A\ B)}_{\text{proof}}$$

where

$$\text{eq_bool}\ x\ y := (x\ \&\&\ y)\ ||\ (!x\ \&\&\ !y)$$

$$\text{eq_pair}\ (A\ B : \text{eqType})\ (u\ v : \text{sort}\ A \times \text{sort}\ B) :=$$
$$\text{equal}\ (\text{fst}\ u)\ (\text{fst}\ v)\ \&\&\ \text{equal}\ (\text{snd}\ u)\ (\text{snd}\ v)$$

Triggering instance resolution

When a unification problem arises of the form

$$\text{sort } ?X \ \widehat{=}\ T$$

Coq matches T against sort field of eqType's canonical instances.

- If $T = \text{bool}$ then $?X = \text{eqType_bool}$
- If $T = (T_1 \times T_2)$ then $?X = \text{eqType_pair } ?X_1\ ?X_2$

  with residual constraints $\text{sort } ?X_1 = T_1$ and $\text{sort } ?X_2 = T_2$.

Canonical structures in overloading

When typechecking

$$\text{equal} \ \ (b_1, b_2)\ (c_1, c_2)$$

for b_i, c_i:bool, we want the system to infer that equal must be:

$$\text{equal}\ (b_1, b_2)\ (c_1, c_2) = \\ (b_1\ \&\&\ c_1\ \ ||\ \ !b_1\ \&\&\ !c_1)\ \&\&\ (b_2\ \&\&\ c_2\ \ ||\ \ !b_2\ \&\&\ !c_2)$$

Canonical structures in overloading

When typechecking

$$\text{equal } ?T \ (b_1, b_2) \ (c_1, c_2)$$

for b_i, c_i:bool, we want the system to infer that equal must be:

$$\text{equal } (b_1, b_2) \ (c_1, c_2) = \\ (b_1 \ \&\& \ c_1 \ || \ !b_1 \ \&\& \ !c_1) \ \&\& \ (b_2 \ \&\& \ c_2 \ || \ !b_2 \ \&\& \ !c_2)$$

Canonical structures in overloading (cont'd)

Typechecking starts with the unification problem:

$$\text{sort } ?T \;\widehat{=}\; \text{bool} \times \text{bool}$$

Using canonical instances, this reduces to:

$$\begin{array}{lcl} ?T & \widehat{=} & \text{eqType_pair} \quad ?T_1 \quad ?T_2 \\ \text{sort } ?T_1 & \widehat{=} & \text{bool} \\ \text{sort } ?T_2 & \widehat{=} & \text{bool} \end{array}$$

Canonical structures in overloading (cont'd)

Typechecking starts with the unification problem:

$$\text{sort } ?T \ \widehat{=}\ \text{bool} \times \text{bool}$$

Using canonical instances, this reduces to:

$$\begin{array}{lll} ?T & \widehat{=} & \text{eqType_pair}\ \ ?T_1\ \ ?T_2 \\ ?T_1 & \widehat{=} & \text{eqType_bool} \\ \text{sort } ?T_2 & \widehat{=} & \text{bool} \end{array}$$

Canonical structures in overloading (cont'd)

Typechecking starts with the unification problem:

$$\mathsf{sort}\ ?T \ \widehat{=}\ \mathsf{bool} \times \mathsf{bool}$$

Using canonical instances, this reduces to:

$$?T \quad \widehat{=} \quad \mathsf{eqType_pair} \quad \mathsf{eqType_bool} \quad ?T_2$$

$$\mathsf{sort}\ ?T_2 \quad \widehat{=} \quad \mathsf{bool}$$

Canonical structures in overloading (cont'd)

Typechecking starts with the unification problem:

$$\mathsf{sort}\ ?T \ \widehat{=}\ \mathsf{bool} \times \mathsf{bool}$$

Using canonical instances, this reduces to:

$$?T \ \widehat{=}\ \mathsf{eqType_pair}\quad \mathsf{eqType_bool}\quad ?T_2$$

$$?T_2 \ \widehat{=}\ \mathsf{eqType_bool}$$

Canonical structures in overloading (cont'd)

Typechecking starts with the unification problem:

$$\text{sort } ?T \;\widehat{=}\; \text{bool} \times \text{bool}$$

Using canonical instances, this reduces to:

$$?T \quad \widehat{=} \quad \text{eqType_pair} \quad \text{eqType_bool} \quad \text{eqType_bool}$$

Canonical structures in overloading (cont'd)

Typechecking starts with the unification problem:

$$\text{sort } ?T \;\widehat{=}\; \text{bool} \times \text{bool}$$

Using canonical instances, this reduces to:

$$?T \;\widehat{=}\; \text{eqType_pair} \quad \text{eqType_bool} \quad \text{eqType_bool}$$

Thereby resolving equal $(b_1, b_2)\ (c_1, c_2)$ as:

$$\text{eq_pair eqType_bool eqType_bool } (b_1, b_2)\ (c_1, c_2)$$

Canonical structures in overloading (cont'd)

Typechecking starts with the unification problem:

$$\text{sort } ?T \;\widehat{=}\; \text{bool} \times \text{bool}$$

Using canonical instances, this reduces to:

$$?T \;\widehat{=}\; \text{eqType_pair} \quad \text{eqType_bool} \quad \text{eqType_bool}$$

Thereby resolving equal (b_1, b_2) (c_1, c_2) as:

$$\text{eq_bool } (b_1, c_1) \;\&\&\; \text{eq_bool } (b_2, c_2)$$

Canonical structures in overloading (cont'd)

Typechecking starts with the unification problem:

$$\text{sort } ?T \ \widehat{=}\ \text{bool} \times \text{bool}$$

Using canonical instances, this reduces to:

$$?T \quad \widehat{=} \quad \text{eqType_pair} \quad \text{eqType_bool} \quad \text{eqType_bool}$$

Thereby resolving $\text{equal } (b_1, b_2)\ (c_1, c_2)$ as:

$$(b_1 \ \&\& \ c_1 \ || \ !b_1 \ \&\& \ !c_1) \ \&\& \ (b_2 \ \&\& \ c_2 \ || \ !b_2 \ \&\& \ !c_2)$$

Lemma overloading

Overloading a simple lemma

A naïve lemma to test that x is in a domain of a certain heap:

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

Fails to apply unless the heap has $x \mapsto v$ at the front.

We shall overload indom to apply to any heap of the form

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

Algorithm for the indom lemma, informally

When applying indom to some goal

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

we should instruct the typechecker to proceed as follows:

- if $h = x \mapsto v$, succeed
- if $h = h_1 \uplus h_2$, recursively test if $x \in \mathsf{dom}\ h_1$
- if not, recursively test if $x \in \mathsf{dom}\ h_2$

Formal specification of the indom algorithm

Define structure ctns x, of heaps that contain x:

$$\underline{\text{structure}}\ \text{ctns}\ x := \text{Ctns}\ \{\text{heap_of} : \text{heap};\\ _ : \text{def heap_of} \to x \in \text{dom heap_of}\}$$

Induced projection heap_of:

$$\text{heap_of} : \forall x : \text{ptr}.\ \text{ctns}\ x \to \text{heap}$$

The second induced projection is our overloaded lemma;

$$\text{indomO} : \forall x{:}\text{ptr}.\ \forall f{:}\text{ctns}\ x.\\ \text{def}\ (\text{heap_of}\ f) \to x \in \text{dom}\ (\text{heap_of}\ f)$$

Implementation of indom

Search algorithm encoded in canonical instances of ctns x:

canonical found $A\ x\ (v : A) :=$ Ctns $x\ (x \mapsto v)\ \ldots$

canonical left $x\ h\ (f : \text{ctns}\ x)\ :=$ Ctns $x\ ((\text{heap_of}\ f) \uplus h)\ \ldots$

canonical right $x\ h\ (f : \text{ctns}\ x)\ :=$ Ctns $x\ (h \uplus (\text{heap_of}\ f))\ \ldots$

Looks like a logic program.

Implementation of indom

Search algorithm encoded in canonical instances of ctns x:

canonical found $A\ x\ (v : A) :=$ Ctns $x\ (x \mapsto v)\ \ldots$

canonical left $x\ h\ (f : \text{ctns}\ x)\ :=$ Ctns $x\ ((\text{heap_of}\ f) \uplus h)\ \ldots$

canonical right $x\ h\ (f : \text{ctns}\ x)\ :=$ Ctns $x\ (h \uplus (\text{heap_of}\ f))\ \ldots$

Looks like a logic program.

(This doesn't quite work, but for now let's pretend it does.)

Using indomO

When applying the overloaded lemma

$$\text{indomO} : \forall x{:}\text{ptr}.\, \forall f{:}\text{ctns } x.$$
$$\text{def (heap_of } f) \to x \in \text{dom (heap_of } f)$$

to some goal

$$G : x \in \text{dom } (h)$$

the inference engine will try to construct a canonical instance f : ctns x such that

$$\text{heap_of } f \ \hat{=}\ h.$$

- if successful, by definition of ctns, x appears in h.
- if unsuccessful, we get type error

Example application of indomO

We want to prove

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

applying

$$\begin{aligned} \mathsf{indomO} : &\ \forall x{:}\mathsf{ptr}.\ \forall f{:}\mathsf{ctns}\ x. \\ &\qquad \mathsf{def}\ (\mathsf{heap_of}\ f) \to x \in \mathsf{dom}\ (\mathsf{heap_of}\ f) \end{aligned}$$

Example application of indomO

This results in the following unification problem:

$$?x \in \mathsf{dom}\ (\mathsf{heap_of}\ ?f) \ \widehat{=}\ y \in \mathsf{dom}\ (z \mapsto u \uplus y \mapsto v)$$

in a context where $?x : \mathsf{ptr}$ and $?f : \mathsf{ctns}\ ?x$.

Example application of indomO

This results in the following unification problem:

$$?x \in \mathsf{dom}\ (\mathsf{heap_of}\ ?f) \ \widehat{=}\ y \in \mathsf{dom}\ (z \mapsto u \uplus y \mapsto v)$$

in a context where $?x : \mathsf{ptr}$ and $?f : \mathsf{ctns}\ ?x$.

This leads in turn to

$$?x \ \widehat{=}\ y$$

Example application of indomO

This results in the following unification problem:

$$y \in \mathsf{dom}\ (\mathsf{heap_of}\ ?f) \ \widehat{=}\ y \in \mathsf{dom}\ (z \mapsto u \uplus y \mapsto v)$$

in a context where $?f : \mathsf{ctns}\ y$.

Example application of indomO

This results in the following unification problem:

$$y \in \mathsf{dom}\ (\mathsf{heap_of}\ ?f) \ \widehat{=}\ y \in \mathsf{dom}\ (z \mapsto u \uplus y \mapsto v)$$

in a context where $?f : \mathsf{ctns}\ y$.

This leads in turn to

$$\mathsf{heap_of}\ ?f \ \widehat{=}\ (z \mapsto u \uplus y \mapsto v)$$

Example application of indomO

$$\text{heap_of } ?f \;\widehat{=}\; (z \mapsto u \uplus y \mapsto v)$$

Two instances apply

$\underline{\text{canonical}}$ left x h (f : ctns x) := Ctns x ((heap_of f) $\uplus$ h) ...

$\underline{\text{canonical}}$ right x h (f : ctns x) := Ctns x (h $\uplus$ (heap_of f)) ...

But which one? We'd like an order, i.e., left and then right.

Let's assume that for now.

Example application of indomO

$$\text{heap_of } ?f \;\widehat{=}\; (z \mapsto u \uplus y \mapsto v)$$

1. Match left

$\underline{\text{canonical}}$ left x h $(f : \text{ctns } x)$ $:=$ Ctns x $((\text{heap_of } f) \uplus h)$ ...

Example application of indomO

$$\text{heap_of } ?f \;\widehat{=}\; (z \mapsto u \uplus y \mapsto v)$$

1. Match left

$\underline{\text{canonical}}$ left x h $(f : \text{ctns } x)$ $:=$ Ctns x $((\text{heap_of } f) \uplus h)$ ...

Example application of indomO

$$\text{heap_of } ?f \ \widehat{=}\ (z \mapsto u \uplus y \mapsto v)$$

1. Match left

$\underline{\text{canonical}}$ left x h $(f : \text{ctns } x)$:= Ctns x $((\text{heap_of } f) \uplus h)$ ...

New unification problem for $?f' : \text{ctns } y$

$$\text{heap_of } ?f' \ \widehat{=}\ z \mapsto u$$

Example application of indomO

$$\text{heap_of } ?f \ \widehat{=} \ (z \mapsto u \uplus y \mapsto v)$$

1. Match left

canonical left x h (f : ctns x) := Ctns x ((heap_of f) $\uplus$ h) ...

New unification problem for $?f'$: ctns y

$$\text{heap_of } ?f' \ \widehat{=} \ z \mapsto u \qquad\qquad \text{fail}$$

Example application of indomO

$$\text{heap_of } ?f \ \widehat{=}\ (z \mapsto u \uplus y \mapsto v)$$

1. Match left – fail
2. Match right

canonical right x h $(f : \text{ctns } x)$ $:=$ Ctns x $(h \uplus (\text{heap_of } f))$ ...

Example application of indomO

$$\text{heap_of } ?f \ \widehat{=} \ (z \mapsto u \uplus y \mapsto v)$$

1. Match left – fail
2. Match right

canonical right x h (f : ctns x) := Ctns x ($h \uplus$ (heap_of f)) ...

Example application of indomO

$$\text{heap_of } ?f \;\widehat{=}\; (z \mapsto u \uplus y \mapsto v)$$

1. Match left – fail
2. Match right

canonical right x h $(f : \text{ctns } x)$ $:=$ Ctns x $(h \uplus (\text{heap_of } f))$ ...

New unification problem for $?f'' : \text{ctns } y$

$$\text{heap_of } ?f'' \;\widehat{=}\; y \mapsto v$$

Example application of indomO

$$\text{heap_of } ?f \;\widehat{=}\; (z \mapsto u \uplus y \mapsto v)$$

1. Match left – fail
2. Match right

<u>canonical</u> right x h (f : ctns x) := Ctns x ($h \uplus$ (heap_of f)) ...

New unification problem for $?f''$: ctns y

$$\text{heap_of } ?f'' \;\widehat{=}\; y \mapsto v$$ succeed by found

Example application of indomO

$$\text{heap_of } ?f \ \widehat{=} \ (z \mapsto u \uplus y \mapsto v)$$

1. Match left – fail
2. Match right

canonical right x h $(f : \text{ctns } x) := \text{Ctns } x \ (h \uplus (\text{heap_of } f)) \ \ldots$

New unification problem for $?f'' : \text{ctns } y$

$$\text{heap_of } ?f'' \ \widehat{=} \ y \mapsto v$$

succeed by found

Result:

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

Implementation of indom: the truth revealed

When I gave the instances instances of ctns x:

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

canonical left x h $(f :$ ctns $x)$:= Ctns x ((heap_of f) $\uplus$ h) ...

canonical right x h $(f :$ ctns $x)$:= Ctns x (h $\uplus$ (heap_of f)) ...

I hid one problem...

Implementation of indom: the truth revealed

When I gave the instances instances of ctns x:

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

canonical left $x\ h\ (f : \text{ctns}\ x)\ := \text{Ctns}\ x\ ((\text{heap_of}\ f) \uplus h)\ \ldots$

canonical right $x\ h\ (f : \text{ctns}\ x)\ := \text{Ctns}\ x\ (h \uplus (\text{heap_of}\ f))\ \ldots$

I hid one problem...

Overlapping instances not allowed in Coq!

The "tagging" design pattern

Introduce a wrapper around the field we key on:

$$\underline{\text{structure}}\ \text{tagged_heap} := \text{Tag}\ \{\text{untag} : \text{heap}\}$$

Modify ctns x and indomO accordingly:

$$\begin{aligned}\underline{\text{structure}}\ \text{ctns}\ x := \text{Ctns}\ \{&\text{heap_of} : \text{tagged_heap};\\ &_ : \text{def}\ (\text{untag heap_of})\\ &\quad \to x \in \text{dom}\ (\text{untag heap_of})\}\end{aligned}$$

$$\begin{aligned}\text{indomO} : \forall x{:}\text{ptr}.\, \forall f{:}\text{ctns}\ x.\, &\text{def}\ (\text{untag}\ (\text{heap_of}\ f)) \to\\ &x \in \text{dom}\ (\text{untag}\ (\text{heap_of}\ f))\end{aligned}$$

. . . although we can make untag implicit, and thereby omit it.

Define synonyms for Tag and order them

This will also order the flow of control, as we shall see soon

right_tag h := Tag h
left_tag h := right_tag h
canonical found_tag h := left_tag h

} all synonyms of Tag

- One synonym of Tag for each canonical instance of ctns 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

Tag each instance of ctns x accordingly

No overlap anymore! Each instance has a different head constant.

<u>canonical</u> found $A\ x\ (v : A) :=$ Ctns x (found_tag $(x \mapsto v)$) ...

<u>canonical</u> left $x\ h\ (f :$ ctns $x) :=$
Ctns x (left_tag (untag (heap_of f) $\uplus$ h)) ...

<u>canonical</u> right $x\ h\ (f :$ ctns $x) :=$
Ctns x (right_tag (h $\uplus$ untag (heap_of f))) ...

Revisiting example application of indomO

We want to prove

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

applying

$$\mathrm{indomO} : \forall x{:}\mathrm{ptr}.\, \forall f{:}\mathrm{ctns}\ x.$$
$$\mathrm{def}\ (\mathrm{untag}\ (\mathrm{heap_of}\ f)) \to x \in \mathrm{dom}\ (\mathrm{untag}\ (\mathrm{heap_of}\ f))$$

Revisiting example application of indomO

We want to prove

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

applying

$\mathsf{indomO} : \forall x{:}\mathsf{ptr}.\ \forall f{:}\mathsf{ctns}\ x.$

$$\mathsf{def}\ (\mathsf{untag}\ (\mathsf{heap_of}\ f)) \to x \in \mathsf{dom}\ (\mathsf{untag}\ (\mathsf{heap_of}\ f))$$

getting the unification problem, for $?f : \mathsf{ctns}\ y$:

$$\mathsf{untag}\ (\mathsf{heap_of}\ ?f)\ \hat{=}\ (z \mapsto u \uplus y \mapsto v)$$

Revisiting example application of indomO

Now recall the tagged_heap structure and its instances:

$$\underline{\text{structure}}\ \text{tagged_heap} := \text{Tag}\ \{\text{untag} : \text{heap}\}$$

$$\left.\begin{array}{l}\text{right_tag}\ h := \text{Tag}\ h \\ \text{left_tag}\ h := \text{right_tag}\ h \\ \underline{\text{canonical}}\ \text{found_tag}\ h := \text{left_tag}\ h\end{array}\right\}\ \text{all synonyms of Tag}$$

Thus, the unification problem

$$\text{untag}\ (\text{heap_of}\ ?f)\ \widehat{=}\ (z \mapsto u \uplus y \mapsto v)$$

reduces to

$$\text{heap_of}\ ?f\ \widehat{=}\ \text{found_tag}\ (z \mapsto u \uplus y \mapsto v)$$

Revisiting example application of indomO

$$\text{heap_of } ?f \;\widehat{=}\; \text{found_tag } (z \mapsto u \uplus y \mapsto v)$$

Only instance of ctns x that applies:

canonical found $A\ x\ (v : A) := \text{Ctns } x\ (\text{found_tag } (x \mapsto v))\ \ldots$

Leading to:

$$?x \mapsto ?v \;\widehat{=}\; z \mapsto u \uplus y \mapsto v$$

Revisiting example application of indomO

$$\text{heap_of } ?f \ \widehat{=}\ \text{found_tag } (z \mapsto u \uplus y \mapsto v)$$

Only instance of ctns x that applies:

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

Leading to:

$$?x \mapsto ?v \ \widehat{=}\ z \mapsto u \uplus y \mapsto v$$

Failure! Our heap is not a singleton.
Coq will now try unfolding the definition of found_tag.

Example application of indomO

$$\text{heap_of } ?f \;\widehat{=}\; \text{left_tag } (z \mapsto u \uplus y \mapsto v)$$

Only instance of ctns x that applies:

$$\begin{array}{l} \underline{\text{canonical}}\ \text{left } x\ h\ (f : \text{ctns } x) \;:= \\ \quad \text{Ctns } x\ (\text{left_tag } (\text{untag } (\text{heap_of } f) \uplus h))\ \ldots \end{array}$$

Leading to:

$$\begin{array}{r} \text{untag } (\text{heap_of } ?f') \;\widehat{=}\; z \mapsto u \\ ?h \;\widehat{=}\; y \mapsto v \end{array}$$

Example application of indomO

$$\text{heap_of } ?f \ \widehat{=}\ \text{left_tag } (z \mapsto u \uplus y \mapsto v)$$

Only instance of ctns x that applies:

$$\underline{\text{canonical}}\ \text{left}\ x\ h\ (f : \text{ctns}\ x)\ := \\ \text{Ctns}\ x\ (\text{left_tag}\ (\text{untag}\ (\text{heap_of}\ f) \uplus h))\ \ldots$$

Leading to:

$$\text{untag (heap_of } ?f') \ \widehat{=}\ z \mapsto u \\ ?h \ \widehat{=}\ y \mapsto v$$

The first subproblem fails because $y \notin \text{dom}(z \mapsto u)$.

Unfold left_tag to right_tag.

Example application of indomO

To cut a long story short, unification succeeds with:

$$?f \ \widehat{=} \ \text{right } y \ (z \mapsto u) \ (\text{found } y \ v)$$

Other cool examples

Overloaded cancellation lemma for heaps

Applying the lemma cancelO on a heap equation

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

cancels the common terms to produce

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

- first logic program turn equations into abstract syntax trees
  - executed during type inference by the unification engine
  - equal variables turn into equal strings
- then a functional program cancels common terms
- requires only the tagging pattern

Overloaded version of noalias

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

$$\text{noaliasO} :$$
$$\forall x\ y : \text{ptr}.\, \forall s : \text{seq ptr}.\, \forall f : \text{scan}\ s.\, \forall g : \text{check}\ x\ y\ s.$$
$$\text{def}\ (\text{untag}\ (\text{heap_of}\ f)) \to x \mathrel{!=} \text{unpack}\ (\text{y_of}\ g)$$

- 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":

$$\text{bnd_write} : \text{verify}\ \overbrace{(x \mapsto v \uplus h)}^{\text{initial heap}}\ e\ \overbrace{q}^{\text{post-condition}} \to$$

$$\text{verify}\ (x \mapsto w \uplus h)\ \underbrace{(\text{write}\ x\ v; e)}_{\text{write } v \text{ in location } x \text{ and then run } e}\ q$$

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

$$G : \text{verify } (i_1 \uplus (x_1 \mapsto 1 \uplus x_2 \mapsto 2))\ (\text{write } x_2\ 4; e)\ q$$

we can apply bnd_writeO to reduce it to:

$$G : \text{verify } (i_1 \uplus (x_1 \mapsto 1 \uplus x_2 \mapsto 4))\ e\ q$$

Search-and-replace pattern: idea

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

Example: Turn $i_1 \uplus (x_1 \mapsto 1 \uplus x_2 \mapsto 2)$ into

$$f = \text{fun } k.\, i_1 \uplus (x_1 \mapsto 1 \uplus k)$$

Then the bnd_writeO lemma can be stated roughly as

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

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

$$H : \mathsf{verify}\ (i_1 \uplus (x_1 \mapsto 1 \uplus x_2 \mapsto 4))\ e\ q$$

we can apply $(\mathsf{bnd_writeO}\ (x := x_2)\ (w := 2))$ to it:

$$H : \mathsf{verify}\ (i_1 \uplus (x_1 \mapsto 1 \uplus x_2 \mapsto 2))\ (\mathsf{write}\ x_2\ 4; e)\ q$$

Note: this duality of use is not possible with tactics

Conclusions

Conclusions

Proof automation $\approx$ lemma overloading

overloading	:	infer code for a function based on arguments
proof automation	:	infer proof for a lemma based on arguments

Customizing type inference by logic programming

Improves robustness, generality, compositionality, when compared to tactics.

Comparison with Coq Type Classes

Coq Type Classes (CTC) introduced by Sozeau and Oury in 2008

- Broadly similar to canonical structures, which predated them
- Instance resolution for CTC guided by heuristic tactics, rather than by Coq unification
- Under development – semantics is in flux

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

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...

Thanks!