Mining opportunities for unique inhabitants in dependent programs

Gabriel Scherer, PhD Student
under supervision of Didier Rémy

Gallium – INRIA

Claim – yet to prove

Types *with unique inhabitants* are a useful notion to write dependently typed programs.

In this talk:

1. What we mean by "type with a unique inhabitant", and how to use them.
2. Discussing usage opportunities in existing dependently typed code.

Definition

Pick a term language t, a type system $\Gamma \vdash t : \tau$ and a (sound) notion of program equivalence $\Gamma \vdash t \equiv t' : \tau$.

Under the environment Γ, a type τ has a *unique inhabitant* if:

$$\exists t, \qquad (\Gamma \vdash t : \tau) \ \wedge \ \forall t', (\Gamma \vdash t' : \tau) \implies (\Gamma \vdash t \equiv t' : \tau)$$

(we will say *singleton* for the rest of this talk)

We are interested in tuples of (term language, type system, equivalence relation) that make this notion interesting.

1. pure term languages
2. equivalence at least $\beta\eta$

Decision problem(s)

The structure of singletons is already interesting for the simply-typed lambda-calculus with sums – whose η-equivalence is tricky.

Decision problem(s)

The structure of singletons is already interesting for the simply-typed lambda-calculus with sums – whose η-equivalence is tricky.

So interesting that we don't have a decision procedure yet.

Decision problem(s)

The structure of singletons is already interesting for the simply-typed lambda-calculus with sums – whose η-equivalence is tricky.

So interesting that we don't have a decision procedure yet.

This talk requires some suspension of disbelief.
We will discuss what we could do *if we knew* how to detect singletons.
In dependently typed systems.

Joker

New language construct for your favorite language: $\Gamma \vdash ?! : \tau$
If τ is a singleton, infer a term, otherwise fail.

Term search can happen in a pure subset of the host language.
Or in use a richer type system (substructural types, more polymorphism or dependencies...).

Applicable (in thought experiments) to ML, Haskell, Coq, Agda...

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip = ?!
```

Intended use case: fill the boring glues around interesting program parts.

Dependent types help

In ML/Haskell, most programs fragments are not in singletons – except in typeful libraries.

```
List.map (fun (x,y) -> (y,x)) [(1,2); (3,4)]
```

Yet, singletons generalize erasable coercions (subtyping) and consistent type-class resolution.

Dependent types help

In ML/Haskell, most programs fragments are not in singletons – except in typeful libraries.

```
List.map (fun (x,y) -> (y,x)) [(1,2); (3,4)]
```

Yet, singletons generalize erasable coercions (subtyping) and consistent type-class resolution.

In dependently typed language, `List.fold` is in a singleton.

```
fold :: forall P, P nil ->
                (forall x xs, P xs -> P (cons x xs)) ->
                forall li, P li
fold init f nil = init
fold init f (cons x xs) = f x xs (fold init f xs)
```

You want to infer either the type or the term.

Where would it be useful?

Unicity of inhabitant is relevant to *program* construction rather than *proof* construction – where inhabitation is enough.
It adds value when it works, but also when it fails.

Where would it be useful?

Unicity of inhabitant is relevant to *program* construction rather than *proof* construction – where inhabitation is enough.
It adds value when it works, but also when it fails.

Split between two verified programming schools:

- "program then prove correct"

Where would it be useful?

Unicity of inhabitant is relevant to *program* construction rather than *proof* construction – where inhabitation is enough.
It adds value when it works, but also when it fails.

Split between two verified programming schools:

- "program then prove correct"
- "program correctly through types"; good for us!

Where would it be useful?

Unicity of inhabitant is relevant to *program* construction rather than *proof* construction – where inhabitation is enough.
It adds value when it works, but also when it fails.

Split between two verified programming schools:

- "program then prove correct"
- "program correctly through types"; good for us!

```
Fixpoint merge l1 l2 :=
  let fix merge_aux l2 :=
  match l1, l2 with
  | [], _ => l2
  | _, [] => l1
  | a1::l1', a2::l2' =>
      if a1 <=? a2
      then a1 :: merge l1' l2
      else a2 :: merge_aux l2'
  end
  in merge_aux l2.

Theorem Sorted_merge : forall l1 l2,
  Sorted l1 -> Sorted l2 -> Sorted (merge l1 l2).
Proof. ... Qed.
```

coq-8.3/theories/Sorting/Mergesort.v

```
emb :: Var Γ σ -> Tm Γ σ
emb vZ = top
emb (vS x τ) = emb x [ pop τ ]
```

James Chapman.
Type Theory should eat itself.
2008.

```
emb :: Var Γ σ -> Tm Γ σ
emb vZ = top
emb (vS x τ) = emb x [ pop τ ]
```

James Chapman.
Type Theory should eat itself.
2008.

```
Definition Sub E E' := ∀ t, Var E t -> Exp E' t.
Program Definition consSub {E E' t} (e:Exp E' t) (s:Sub E E')
: Sub (t::E) E' :=
  fun t' (v:Var (t::E) t') =>
  match v with
    | ZVAR _ _ => e
    | SVAR _ _ _ v' => s _ v'
  end.
```

Nick Benton, Chung-Kil Hur, Andrew Kennedy, and Conor McBride.
Strongly Typed Term Representation in Coq.
2009.

Two-level languages

LF family (Twelf, Beluga, VeriML. . .): two layers, an object language and a host language. Computation only happens at the host. It's natural to allow dependency on the object language.

VeriML: object language represents rich terms of higher-order logic (proofs and propositions). Useful to write tactics.

The "program then prove correct" style is not available!
Lots of opportunities for singleton types.

```
Inductive removed [T : Type] : List/[T] -> T -> List/[T] -> Prop :=
| removedHead : ∀hd tl, removed (cons hd tl) hd tl
| removedTail : ∀elm hd tl tl',
    removed tl elm tl' -> removed (cons hd tl) elm (cons hd tl') ;;

letrec min_list:
  ({ϕ : ctx}, {T : @Type}, cmp : (@T) -> (@T) -> bool) ->
  (l : @List) -> (min : @T) * (rest : @List) * hol(@removed l min rest)
= fun {ϕ T} cmp l =>
  let < @l' , @pfl' > = default_rewriter @l in
  let < @min, @rest, @pf > = holmatch @l' with
    | @nil -> error
    | @cons hd nil ->  < @hd , @nil , @removedHead ?  ?  >
    | @cons hd tl ->
        let < min', rem, pf > = min_list cmp @tl in
        if (cmp @hd @min' ) then
          < @hd , @tl, Exact @removedHead hd tl >
        else
          < @min', @cons hd rem, @removedTail pf >
  in < @min , @rest , {{ Auto }} > ;;
```

```
Inductive removed [T : Type] : List/[T] -> T -> List/[T] -> Prop :=
| removedHead : ∀hd tl, removed (cons hd tl) hd tl
| removedTail : ∀elm hd tl tl',
   removed tl elm tl' -> removed (cons hd tl) elm (cons hd tl') ;;

letrec min_list:
  ({ϕ : ctx}, {T : @Type}, cmp : (@T) -> (@T) -> bool) ->
  (l : @List) -> (min : @T) * (rest : @List) * hol(@removed l min rest)
= fun {ϕ T} cmp l =>
  let < @l' , @pfl' > = default_rewriter @l in
  let < @min, @rest, @pf > = holmatch @l' with
    | @nil -> error
    | @cons hd nil ->  < @hd , @nil , @removedHead ?  ?  >
    | @cons hd tl ->
       let < min', rem, pf > = min_list cmp @tl in
       if (cmp @hd @min' ) then
         < @hd , @tl, Exact @removedHead hd tl >
       else
         < @min', @cons hd rem, @removedTail pf >
  in < @min , @rest , {{ Auto }} > ;;
```

```
Inductive removed [T : Type] : List/[T] -> T -> List/[T] -> Prop :=
| removedHead : ∀hd tl, removed (cons hd tl) hd tl
| removedTail : ∀elm hd tl tl',
   removed tl elm tl' -> removed (cons hd tl) elm (cons hd tl') ;;

letrec min_list:
  ({ϕ : ctx}, {T : @Type}, cmp : (@T) -> (@T) -> bool) ->
  (l : @List) -> (min : @T) * (rest : @List) * hol(@removed l min rest)
= fun {ϕ T} cmp l =>
  let < @l' , @pfl' > = default_rewriter @l in
  let < @min, @rest, @pf > = holmatch @l' with
    | @nil -> error
    | @cons hd nil ->  ?!
    | @cons hd tl ->
       let < min', rem, pf > = min_list cmp @tl in
       if (cmp @hd @min' ) then
         < @hd , @tl, Exact @removedHead hd tl >
       else
         < @min', @cons hd rem, @removedTail pf >
  in < @min , @rest , {{ Auto }} > ;;
```

```
Inductive removed [T : Type] : List/[T] -> T -> List/[T] -> Prop :=
| removedHead : ∀hd tl, removed (cons hd tl) hd tl
| removedTail : ∀elm hd tl tl',
   removed tl elm tl' -> removed (cons hd tl) elm (cons hd tl') ;;

letrec min_list:
  ({ϕ : ctx}, {T : @Type}, cmp : (@T) -> (@T) -> bool) ->
  (l : @List) -> (min : @T) * (rest : @List) * hol(@removed l min rest)
= fun {ϕ T} cmp l =>
  let < @l' , @pfl' > = default_rewriter @l in
  let < @min, @rest, @pf > = holmatch @l' with
    | @nil -> error
    | @cons hd nil ->  ?!
    | @cons hd tl ->
        let < min', rem, pf > = min_list cmp @tl in
        if (cmp @hd @min' ) then
          < @hd , @tl, Exact @removedHead hd tl >
        else
          < @min', @cons hd rem, @removedTail pf >
  in < @min , @rest , {{ Auto }} > ;;
```

```
Inductive removed [T : Type] : List/[T] -> T -> List/[T] -> Prop :=
| removedHead : ∀hd tl, removed (cons hd tl) hd tl
| removedTail : ∀elm hd tl tl',
   removed tl elm tl' -> removed (cons hd tl) elm (cons hd tl') ;;

letrec min_list:
  ({φ : ctx}, {T : @Type}, cmp : (@T) -> (@T) -> bool) ->
  (l : @List) -> (min : @T) * (rest : @List) * hol(@removed l min rest)
= fun {φ T} cmp l =>
  let < @l' , @pfl' > = default_rewriter @l in
  let < @min, @rest, @pf > = holmatch @l' with
    | @nil -> error
    | @cons hd nil ->  ?!
    | @cons hd tl ->
        let < min', rem, pf > = min_list cmp @tl in
        if (cmp @hd @min' ) then
           < ?!, @tl, ?!  >
        else
          < @min', @cons hd rem, @removedTail pf >
  in < @min , @rest , {{ Auto }} > ;;
```

```
Inductive removed [T : Type] : List/[T] -> T -> List/[T] -> Prop :=
| removedHead : ∀hd tl, removed (cons hd tl) hd tl
| removedTail : ∀elm hd tl tl',
   removed tl elm tl' -> removed (cons hd tl) elm (cons hd tl') ;;

letrec min_list:
  ({ϕ : ctx}, {T : @Type}, cmp : (@T) -> (@T) -> bool) ->
  (l : @List) -> (min : @T) * (rest : @List) * hol(@removed l min rest)
= fun {ϕ T} cmp l =>
  let < @l' , @pfl' > = default_rewriter @l in
  let < @min, @rest, @pf > = holmatch @l' with
    | @nil -> error
    | @cons hd nil ->  ?!
    | @cons hd tl ->
       let < min', rem, pf > = min_list cmp @tl in
       if (cmp @hd @min' ) then
          < ?!, @tl, ?!  >
       else
         < @min', @cons hd rem, @removedTail pf >
  in < @min , @rest , {{ Auto }} > ;;
```

```
Inductive removed [T : Type] : List/[T] -> T -> List/[T] -> Prop :=
| removedHead : ∀hd tl, removed (cons hd tl) hd tl
| removedTail : ∀elm hd tl tl’,
   removed tl elm tl’ -> removed (cons hd tl) elm (cons hd tl’) ;;

letrec min_list:
  ({φ : ctx}, {T : @Type}, cmp : (@T) -> (@T) -> bool) ->
  (l : @List) -> (min : @T) * (rest : @List) * hol(@removed l min rest)
= fun {φ T} cmp l =>
  let < @l’ , @pfl’ > = default_rewriter @l in
  let < @min, @rest, @pf > = holmatch @l’ with
    | @nil -> error
    | @cons hd nil ->  ?!
    | @cons hd tl ->
       let < min’, rem, pf > = min_list cmp @tl in
       if (cmp @hd @min’ ) then
          < ?!, @tl, ?!  >
       else
         < ?!, @cons hd rem, ?!  >
  in < @min , @rest , {{ Auto }} > ;;
```

Conclusions so far

Bad: simpler inhabitation search is just as useful in a lot of cases.

Mixed: Our intuition about singletons needs more training.

Good: There is no confusion between intent-expressing types/code, and glue.

Good: There are opportunities for singleton types, when programming with rich types.