Unicity of type inhabitants; a Work in Progress

Gabriel Scherer

Gallium (INRIA Paris-Rocquencourt)

May 30, 2013

What? This talk is about a problem rather than a solution.

The question

Given a type T, does T have a *unique* inhabitant?
(modulo observational equivalence)

We need to fix a type system and a *pure* term language.

Let's start with the simply-typed lambda-calculus (STLC)
with arrows, products *and sums*.

Remark: (non-)relation with singleton types $\{= M\}$.

Why? Practical motivations

A *principal* approach to code inference.

Informal conjecture

When programmers feel bored even before writing the code, it's because *there are no choices to be made.*

Provide a feature to fill some hole (?), that fails if there are several possible choices.

```
val swap : 'a 'b 'c. ('a * 'b * 'c) -> ('a * 'c * 'b

let swap = ?
```

Code inference example

Most general form ($\Gamma \vdash \texttt{?} : \sigma$).
Default context choice ($\emptyset$), inferred type.

Code inference example

Most general form ($\Gamma \vdash \texttt{?} : \sigma$).
Default context choice ($\emptyset$), inferred type.

```
Type_variant (
  List.map (fun (name, name_loc, ctys, option, loc) ->
      name, List.map (fun cty -> cty.ctyp_type) ctys, option)
    cstrs
)
```

Code inference example

Most general form ($\Gamma \vdash ? : \sigma$).
Default context choice ($\emptyset$), inferred type.

```
Type_variant (
  List.map (fun (name, name_loc, ctys, option, loc) ->
      name, List.map (fun cty -> cty.ctyp_type) ctys, option)
    cstrs
)
```

```
Type_variant (
  List.map (? (List.map (fun cty -> cty.ctyp_type))) cstrs
)
```

Analysis of the `typing/` code. For 100 instances of `List.map (fun ...)`, about 30 of them could use code inference.

Uses of code inference

Non-interactive use:

- glue between trivial parts of the program
  I forgot the argument order... but only one type-correct choice.
- more ambitious: generic boilerplate
  Is there a type whose unique inhabitant is `List.map`? (next slide)
- re-expresses other code inference feature
  type classes, implicits...

Interactive use: program-assistant tactics?

Note: we're not using scoring/heuristics [recent C♯, Scala work].

Interaction between type and term inference. You can't do both at once, but they can cooperate.

What's a precise type for List.map?

$$\forall \alpha \beta . (\alpha \to \beta) \to (\mathit{List}\ \alpha \to \mathit{List}\ \beta) \qquad (?\ \texttt{f}\ \texttt{li})$$

What's a precise type for List.map?

$$\forall \alpha\beta.(\alpha \to \beta) \to (\mathit{List}\ \alpha \to \mathit{List}\ \beta) \qquad (\texttt{? f li})$$
$$\forall \alpha\beta.(\alpha \multimap \beta) \to (\mathit{List}\ \alpha \multimap \mathit{List}\ \beta) \qquad (\texttt{? f } \circ\!\!-\texttt{li})$$

What's a precise type for List.map?

$$
\begin{array}{lr}
\forall \alpha\beta.(\alpha \to \beta) \to (\mathit{List}\ \alpha \to \mathit{List}\ \beta) & (\texttt{?}\ \texttt{f}\ \texttt{li}) \\
\forall \alpha\beta.(\alpha \multimap \beta) \to (\mathit{List}\ \alpha \multimap \mathit{List}\ \beta) & (\texttt{?}\ \texttt{f}\ \circ\!\!-\texttt{li}) \\
\forall \alpha\beta.(\alpha \mathrel{-\!\!\triangleright} \beta) \to (\mathit{List}\ \alpha \mathrel{-\!\!\triangleright} \mathit{List}\ \beta) & (\texttt{?}\ \texttt{f}\ \triangleleft\!\!-\texttt{li})
\end{array}
$$

We are:

- using more expressive types than the host language ones
- producing purer terms

What's a precise type for List.map?

$$\begin{array}{lr}
\forall \alpha\beta.(\alpha \to \beta) \to (\mathit{List}\ \alpha \to \mathit{List}\ \beta) & (?\ \mathtt{f}\ \mathtt{li}) \\
\forall \alpha\beta.(\alpha \multimap \beta) \to (\mathit{List}\ \alpha \multimap \mathit{List}\ \beta) & (?\ \mathtt{f}\ \circ\!\!-\mathtt{li}) \\
\forall \alpha\beta.(\alpha \mathrel{-\!\!\triangleright} \beta) \to (\mathit{List}\ \alpha \mathrel{-\!\!\triangleright} \mathit{List}\ \beta) & (?\ \mathtt{f}\ \triangleleft\!\!-\mathtt{li})
\end{array}$$

We are:

- using more expressive types than the host language ones
- producing purer terms

For `fold`, need to move to dependent types; decreasing gains.

$$\begin{array}{ll}
\forall \alpha\beta, & \forall (A : \star)(P : \mathit{List}\ A \to \star), \\
\quad \beta \to & \quad P\ \mathtt{nil} \to \\
\quad (\alpha \to \beta \to \beta) \to & \quad (\forall (a : A)(l : \mathit{List}\ A), P\ l \to P\ (\mathtt{cons}\ a\ l)) \to \\
\quad \mathit{List}\ \alpha \to \beta & \quad \forall (l : \mathit{List}\ A),\ \ P\ l
\end{array}$$

Why? Theoretical motivations

It's fun: a question so simple to state must have interesting anwsers.

It's an excuse to look at the proof-search research with different eyes.
Look at *dynamic behavior*, rather than just yes/no inhabitation problems.

Caution required

Intuitionistic sequent calculi generally have a *contraction* rule

$$\frac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B} \qquad\qquad \frac{\Gamma, A, B \vdash C}{\Gamma, A * B \vdash C}$$

You can get rid of contraction if you preserve formulas at use site.

$$\frac{\Gamma, A * B, A, B \vdash C}{\Gamma, A * B \vdash C}$$

For sums and pairs, it is in fact not needed, but it is for arrows.

$$\frac{\Gamma, A \rightarrow B \vdash A \qquad \Gamma, B \vdash C}{\Gamma, A \rightarrow B \vdash C}$$

Dropping the arrow on the right is complete, but not dynamically so.

How? High-level directions

I recently started working on this. I will warmly welcome any suggestion.

Directions to explore in parallel

- Keep looking for related work.
  Diverse, hard to find, not well-connected.
- Enrich type systems to express more types with unique inhabitants.
  Substructural logics, polymorphic (parametricity), dependent types.
- Devise practical algorithms to check unicity.
  (Bulk of this talk)

Some related work

J. B. Wells and Boris Yakobowski.
Graph-based proof counting and enumeration with applications for program fragment synthesis.
In LOPSTR 2004.

Takahito Aoto.
Uniqueness of normal proofs in implicational intuitionistic logic.
Journal of Logic, Language and Information, 8:217–242, 1999.

Sabine Broda and Luís Damas.
On long normal inhabitants of a type.
J. Log. Comput., 15(3):353–390, 2005.

Pierre Boureau and Sylvain Salvati.
Game semantics and uniqueness of type inhabitance in the simply-typed λ-calculus.
Typed Lambda-Calculi and Applications, 2011.

A few words on [Yakobowski and Wells]

Consider the graph whose nodes are sequent, and edges are valid inference rules.

When context is a set, subformula property implies finiteness.

Can be seen as a "memoization" techniques: cycles in the graph can be dropped without hurting completeness.

(Idea of the paper: from this graph structure with set-contexts, deduce information about the infinite structure of multiset-contexts.)

Facing the Decision problem: Unicity for STLC

Obvious idea: enumerate proofs, check that there is only one.

Usual problem: irrelevant permutations allowed by the proof system

$$\dfrac{\dfrac{A, B, C, D \vdash E}{A, B, C * D \vdash E}}{A * B, C * D \vdash E} \qquad \dfrac{\dfrac{A, B, C, D \vdash E}{A * B, C, D \vdash E}}{A * B, C * D \vdash E}$$

Two approaches:

- do equivalence checks after enumeration to remove duplicates (simple, not fun, not efficient in general)
- change the proof system to remove those duplicates

Mandatory step towards duplicates-free systems: Focusing

Quotient by reordering of {non,}inversible proof steps.

$$\frac{\Gamma; \Delta, A \vdash B}{\Gamma; \Delta \vdash A \to B} \qquad \frac{\Gamma; \Delta, A, B \vdash C}{\Gamma; \Delta, A * B \vdash C} \qquad \frac{\Gamma; \Delta, A \vdash C \qquad \Gamma; \Delta, B \vdash C}{\Gamma; \Delta, A + B \vdash C}$$

$$\frac{\Gamma, X; \Delta \vdash C}{\Gamma; \Delta, X \vdash C} \qquad \frac{\Gamma \vdash [P]}{\Gamma; \emptyset \vdash P} \qquad \frac{\Gamma, [N] \vdash X}{\Gamma, N; \emptyset \vdash X} \qquad \frac{\Gamma, [N] \vdash P \qquad \Gamma; P \vdash Q}{\Gamma, N \vdash Q}$$

$$\frac{\Gamma \vdash [A] \qquad \Gamma \vdash [B]}{\Gamma \vdash [A * B]} \qquad \frac{\Gamma \vdash [A_i]}{\Gamma \vdash [A_1 + A_2]} \qquad \frac{\Gamma; \emptyset \vdash N}{\Gamma \vdash [N]}$$

$$\frac{}{\Gamma, [X] \vdash X} \qquad \frac{\Gamma, [N] \vdash A \to B \qquad \Gamma \vdash [A]}{\Gamma, [N] \vdash B}$$

Focused proofs correspond to β-normal, η-long terms. Good!

Shortcomings of Focusing

Too many proofs of $(X \to Y + Z) \to X \to X$.

```
fun f x -> ?
```

Shortcomings of Focusing

Too many proofs of $(X \to Y + Z) \to X \to X$.

```
fun f x -> ?
```

```
fun f x -> x
```

```
fun f x -> match f x with
  | L y -> ?
  | R z -> ?
```

Shortcomings of Focusing

Too many proofs of $(X \to Y + Z) \to X \to X$.

```
fun f x -> ?
```

```
fun f x -> x
```

```
fun f x -> match f x with
  | L y -> ?
  | R z -> ?
```

```
fun f x -> match f x with
  | L y -> x
  | R z -> x
```

```
fun f x -> match f x with
  | L y -> (match f x with
            | L y' -> ?
            | R z -> ?)
  | R z -> x
```

```
fun f x -> match f x with
  | L y -> x
  | R z -> (match f x with
            | L y -> ?
            | R z' -> ?)
```

```
fun f x -> match f x with
  | L y -> (match f x with L y' -> ? | R z  -> ?)
  | R z -> (match f x with L y  -> ? | R z' -> ?)
```

Shortcomings of Focusing

Too many proofs of $(X \to Y + Z) \to X \to X$.

```
fun f x -> ?
```

```
fun f x -> x
```

```
fun f x -> match f x with
  | L y -> ?
  | R z -> ?
```

```
fun f x -> match f x with
  | L y -> x
  | R z -> x
```

```
fun f x -> match f x with
  | L y -> (match f x with
           | L y' -> ?
           | R z -> ?)
  | R z -> x
```

```
fun f x -> match f x with
  | L y -> x
  | R z -> (match f x with
           | L y -> ?
           | R z' -> ?)
```

```
fun f x -> match f x with
  | L y -> (match f x with L y' -> ? | R z  -> ?)
  | R z -> (match f x with L y  -> ? | R z' -> ?)
```

Remark: $(Y + Z) \to X \to X$ would be fine.

η-equivalence for sum types

Weak, local equivalence:

```
e = match e with
    | L y -> L y
    | R z -> R z
```

In particular:

```
t = match e with
    | L y -> t
    | R y -> t
```

and...

Full, non-local, categorical equivalence

```
C[e] = match e with
       | L y -> C[L y]
       | R z -> C[R z]
```

```
match e with
| L y -> C1[y][match e with M1]
| R z -> C2[z][match e with M2]
```

= (strong η-sum)

```
match e with
| L y -> C1[y][match e with M1]
| R z -> C2[z][match e with M2]
```

= (strong η-sum)

```
match e with
| L y0 ->
  (match L y0 with
   | L y -> C1[y][match L y0 with M1]
   | R z -> C2[z][match L y0 with M2])
| R z0 ->
  (match R z0 with
   | L y -> C1[y][match R z0 with M1]
   | R z -> C2[z][match R z0 with M2])
```

= (β-sum)

```
match e with
| L y -> C1[y][match e with M1]
| R z -> C2[z][match e with M2]
```

= (strong η-sum)

```
match e with
| L y0 ->
  (match L y0 with
   | L y -> C1[y][match L y0 with M1]
   | R z -> C2[z][match L y0 with M2])
| R z0 ->
  (match R z0 with
   | L y -> C1[y][match R z0 with M1]
   | R z -> C2[z][match R z0 with M2])
```

= (β-sum)

```
match e with
| L y0 -> C1[y0][match L y0 with M1]
| R z0 -> C2[z0][match R z0 with M2]
```

Checking strong η-equivalence for sums

[Balat and Di Cosmo, 2004]; [Lindley, 2005]
General idea: move sum destructions *as early as possible*, then remove duplicates.

```
fun f g ...
  match ... with
    ...
    fun x y ...
      match ... with
      ...
       fun g z ...
          match f x with ...
```

Remark

$(\to)$ and $(+)$ are enemies in intuitionistic logic.
Both can be introduced reversibly, but not both at the same time.

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \qquad \frac{\Gamma \vdash A_i}{\Gamma \vdash A_1 + A_2}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash (A \to B), \Delta} \qquad \frac{\Gamma \vdash A_1, A_2, \Delta}{\Gamma \vdash (A_1 + A_2), \Delta}$$

(Remark in remark: intuitionistic focusing makes arbitrary choices. Related to various translations into linear logic [Chaudhuri and Miller].)

A general approach: saturation

Goal: integrate sum equivalence into proof search.

Our idea: Context saturation

Each time we introduce new things in the context, do *all possible destructions* that involve them *and* might get used in a proof term.

Saturation example

With saturation,

```
fun f x ->
  match f x with
    | L y -> (match f x with L y' -> ? | R z -> ?)
    | R z -> (match f x with L y -> ? | R z -> ?)
```

is ruled out. But for:

```
fun f x -> match f x with
  | L y -> x
  | R z -> x
```

it depends.
It would be ruled out as well if our proof search was sophisticated enough to notice that neither Y nor Z can help prove X.

Saturation Facts

Conjecture: a search calculus enforcing saturation solves the sum equivalence problem.

Danger: without clever ideas for checking “potential usefulness” of destructs, this method is impractical.

Hope: this approach allows to solve not only the -sum problem, but generalizes nicely to other constructors with tricky equalities.

Embarassing detail: no other example known, so generalization of little value; suggestions appreciated.

But: saturation is not obvious

A saturating calculus surprisingly hard to define.

Nave idea: at the end of each reversible phase (or incrementally during them), saturate the context. Focusing phases will only run with saturated contexts.

Context saturation operation $\mathrm{sat}(\Gamma)$?

$\mathrm{sat}(\Gamma; A \to B) = \mathrm{sat}(\Gamma, A \to B; B)$ when $\Gamma \vdash A$.
Problem when A of the form $B \to C$: re-saturation needed (recursively).

Termination? Practicality? We need something clever here.

Exploring the theorem proving countryside

Saturation seems costly in general, but sometimes it is *required* to solve inhabitation.

$$(X \to Y + Z) \to X \to Z + Y$$

Let's look at the automated theorem provign literature. Hopefully their techniques/optimizations have helpful semantic content.

Most research centered on classical logic – easy shortcuts due to arrow/sum permutation. But:

- The *inverse method* has been adapted to linear [Chaudhuri], intuitionistic logic. Sequent-saturation technique – may help for context saturation ?
- *Connection-based*, or Matrix-based calculi; horribly complicated, but probably helpful to avoid redundant work.

Presentation of the Inverse Method

Based on a termination argument that we can reuse for saturation: the subformula property.

Subformulas of $(X \to Y + Z) \to X \to X$

- (positively) X; $(X \to X)$; $((X \to Y + Z) \to X \to X)$
- (negatively) $(Y + Z)$; $(X \to Y + Z)$; X

Some rules:

$$\frac{X \text{ atom}}{X \vdash X} \qquad \frac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B} \qquad \frac{\Gamma_1 \vdash A \qquad \Gamma_2 \vdash B}{\Gamma_1, \Gamma_2 \vdash A * B}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \qquad \frac{\Gamma \vdash B \qquad A \notin \Gamma}{\Gamma \vdash A \to B}$$

Inverse Method: Pros and Cons

Note: already encoded some neededness information.

Can be refined with

- polarization
- focusing (derived constructors)

Has been used in practice to refute provability (Imogen, [McLaughlin and Pfenning, 2008]), so is practically able to perform saturation.

But: unclear how its inherent sharing/subsumption preserves the dynamic semantics of proof-terms.

Going further

Current idea : perform an inverse method to forward-explore the sequent space, then go backward to collect maximized proof.

Going on in parallel :

- "path calculi" are optimizations techniques on top of the inverse method that allow to further prune the search space [Degtyarev and Voronkov, 2001] and may help even further on "neededness" question.
- understand and integrate ideas from connection-based calculi [Galmiche and Méry, recent]

Thanks.

Any questions ?