Which simple types have a unique inhabitant?

Gabriel Scherer, Didier Rémy

Gallium – INRIA

September 1st, 2015
Many programs are fun to write. Some parts can be boring, though.

We get bored when there is no choice to make. The compiler should guess this code for you: code inference.
Many programs are fun to write. Some parts can be boring, though.

We get bored when there is no choice to make. The compiler should guess this code for you: code inference.

Some existing examples:
- Overloaded identifier disambiguation.
- Type classes, implicits.
- Proof assistants tactics.
Code inference

Many programs are fun to write. Some parts can be boring, though.

We get bored when there is no choice to make. The compiler should guess this code for you: code inference.

Some existing examples:
- Overloaded identifier disambiguation.
- Type classes, implicits.
- Proof assistants tactics.

We should infer any code uniquely determined by its type.
Which types have a unique inhabitant?

Uniquely inhabited typing \((\Gamma, A)\): inhabited \((\Gamma \vdash t : A)\) and

\[
\Gamma \vdash t : A \land \Gamma \vdash u : A \implies \Gamma \vdash t \simeq u : A
\]

\((\vdash)\) in a given type system (STLC with atoms, products and \textbf{sums})

\((\simeq)\) modulo some program equivalence (here, \(\beta\eta\))

Contribution: a decision procedure (algorithm) in this setting.
Killer example

The Monad instance for Exception $A \overset{\text{def}}{=} A + E$ is canonical.

\begin{align*}
\text{return: } & \quad X \quad \to \quad (X + E) \\
\text{bind: } & \quad X + E \quad \to \quad (X \to Y + E) \quad \to \quad Y + E
\end{align*}

Functor instance also canonical.

Applicative functor, two distinct choices.

\begin{align*}
\text{ap: } & \quad (X \to Y) + E \quad \to \quad X + E \quad \to \quad Y + E
\end{align*}

(Which argument to evaluate first?)
\( \beta \eta \)-equivalence

Type system for pure language: enforces strong normalization.

\[
(\lambda x. t) \ u \rightarrow_\beta t[u/x] \quad (t : A \rightarrow B) =_\eta \lambda x. t \ x
\]

\[
\pi_i (t_1, t_2) \rightarrow_\beta t_i \quad (t : A \ast B) =_\eta (\pi_1 t, \pi_2 t)
\]

\[
\text{match } (L \ t) \text{ with } \begin{cases} 
L \ x_1 \rightarrow u_1 \\
R \ x_2 \rightarrow u_2 \end{cases} \rightarrow_\beta u_1[t/x_1]
\]

\[
(t : A + B) =_\eta \text{ match } t \text{ with } \begin{cases} 
L \ x_1 \rightarrow L \ x_1 \\
R \ x_2 \rightarrow R \ x_2 \end{cases}
\]
\section*{$\beta\eta$-equivalence}

Type system for pure language: enforces strong normalization.

\begin{align*}
(\lambda x. \, t) \, u & \rightarrow_\beta t[u/x] & (t : A \rightarrow B) =_\eta \lambda x. \, t \, x \\
\pi_i (t_1, t_2) & \rightarrow_\beta t_i & (t : A \ast B) =_\eta (\pi_1 \, t, \pi_2 \, t) \\
\text{match} \, (L\, t) \, \text{with} & \begin{array}{ll}
L \, x_1 & \rightarrow u_1 \\
R \, x_2 & \rightarrow u_2
\end{array} & \rightarrow_\beta u_1[t/x_1] \\
(t : A + B) =_\eta \text{match} \, t \, \text{with} & \begin{array}{ll}
L \, x_1 & \rightarrow L \, x_1 \\
R \, x_2 & \rightarrow R \, x_2
\end{array} \\
\text{But:} & \begin{array}{ll}
(t, u) ? & \text{match} \, t \, \text{with} \\
& \begin{array}{ll}
L \, x_1 & \rightarrow (L \, x_1, u) \\
R \, x_2 & \rightarrow (R \, x_2, u)
\end{array}
\end{array}
\end{align*}
\[\beta\eta\text{-equivalence}\]

Type system for pure language: enforces strong normalization.

\[(\lambda x. t) \ u \rightarrow_\beta t[u/x]\quad (t : A \rightarrow B) =_\eta \lambda x. t \ x\]

\[\pi_i (t_1, t_2) \rightarrow_\beta t_i\quad (t : A \ast B) =_\eta (\pi_1 t, \pi_2 t)\]

\[\text{match } (L \ t) \text{ with } | L \ x_1 \rightarrow \ u_1 \quad \rightarrow_\beta \ u_1[t/x_1]\]

\[(t : A + B) =_\eta \text{match } t \text{ with } | L \ x_1 \rightarrow \quad L \ x_1\]

\[\text{match } t \text{ with } | R \ x_2 \rightarrow \quad R \ x_2\]

But:

\[(t, u) \overset{?}{=} \text{match } t \text{ with } | L \ x_1 \rightarrow (L \ x_1, u) \quad R \ x_2 \rightarrow (R \ x_2, u)\]

\[C[\Box] \overset{\text{def}}{=} (\Box, u)\]
\(\beta\eta\)-equivalence

Type system for pure language: enforces strong normalization.

\[
(\lambda x. t) \ u \rightarrow_\beta \ t[u/x]
\]

\[
(t : A \to B) =_\eta \lambda x. t \ x
\]

\[
\pi_i (t_1, t_2) \rightarrow_\beta \ t_i
\]

\[
(t : A \times B) =_\eta (\pi_1 \ t, \pi_2 \ t)
\]

\[
\text{match} \ (L \ t) \ \text{with} \ \\
\left| \begin{array}{c}
L \ x_1 \rightarrow \ u_1 \\
R \ x_2 \rightarrow \ u_2
\end{array} \right. \rightarrow_\beta \ u_1[t/x_1]
\]

\[
\forall C[\square : A + B],
\]

\[
C[t : A + B] =_\eta \text{match} \ t \ \text{with} \ \\
\left| \begin{array}{c}
L \ x_1 \rightarrow \ C[L \ x_1] \\
R \ x_2 \rightarrow \ C[R \ x_2]
\end{array} \right.
\]

5
\(\beta\eta\)-equivalence

Type system for pure language: enforces strong normalization.

\[
(\lambda x. \, t) \, u \rightarrow_\beta t[u/x] \quad (t : A \rightarrow B) =_\eta \lambda x. \, t \, x
\]

\[
\pi_i \, (t_1, t_2) \rightarrow_\beta t_i \quad (t : A \times B) =_\eta (\pi_1 \, t, \pi_2 \, t)
\]

\[
\text{match} \, (L \, t) \, \text{with} \quad \begin{align*}
L \, x_1 \rightarrow u_1 \\
R \, x_2 \rightarrow u_2
\end{align*} \rightarrow_\beta u_1[t/x_1]
\]

\[
\forall \, C[\emptyset : A + B], \\
C[t : A + B] =_\eta \text{match} \, t \, \text{with} \quad \begin{align*}
L \, x_1 \rightarrow C[L \, x_1] \\
R \, x_2 \rightarrow C[R \, x_2]
\end{align*}
\]

Unicity

A search process, enumerating distinct normal forms.

We know about proof search.

We know about program equivalence.
Unicity

A search process, enumerating distinct normal forms.

We know about proof search.

We know about program equivalence.

Is there a proof or type system that characterizes distinct programs? No duplicates.
the Graal of program equivalence

A type system for “normal forms” \( \Gamma \vdash_{nf} \nu : A \) that is

- **canonical**: syntactically distinct \( \Rightarrow \) semantically distinct
- **complete**: each STLC program is equivalent to a typable normal form

Unicity test by goal-directed search in this system.

\[ \Gamma \vdash_{nf} ? : A \]

Contribution: this, for simply-typed \( \lambda \)-calculus with sums.
\( \beta \)-short (weak)-\( \eta \)-long does not cut it

\[
f : (X \to Y + Y), x : X \vdash ? : X
\]
\(\beta\)-short (weak)-\(\eta\)-long does not cut it

\[ f : (X \rightarrow Y + Y), x : X \vdash ? : X \]

\(x\)
\( \beta \)-short (weak)-\( \eta \)-long does not cut it

\[ f : (X \rightarrow Y + Y), x : X \vdash ? : X \]

\[
\begin{array}{l}
\text{x} \\
\text{match } f \times \text{ with } \\
\text{  L } y_1 \rightarrow x \\
\text{  R } y_2 \rightarrow x \\
\end{array}
\]

In general: equivalent programs may differ by matching on the same subterm at different places. Need to quotient over that.
\(\beta\)-short (weak)-\(\eta\)-long does not cut it

\[ f : (X \to Y + Y), \quad x : X \vdash ? : X \]

\[ x \]

match \(f \times \) with

\[ \begin{align*}
    & L \ y_1 \to x \\
    & R \ y_2 \to x
\end{align*} \]

match \(f \times \) with

\[ \begin{align*}
    & L \ y_1 \to \text{match } f \times \text{ with} \\
    & R \ y_2 \to x
\end{align*} \]

...
\[ f : (X \to Y + Y), \, x : X \vdash ? : X \]

\[
x
\begin{array}{c|c|c}
\text{match } f \times \text{ with} & \text{L } y_1 \to x & \text{L } z_1 \to x \\
\text{match } f \times \text{ with} & \text{R } y_2 \to x & \text{R } z_2 \to x \\
\text{match } f \times \text{ with} & \text{L } z_1 \to \text{match } f \times \text{ with} & \text{L } z_1 \to x \\
\text{match } f \times \text{ with} & \text{R } y_2 \to x & \text{R } z_2 \to x
\end{array}
\]

In general: equivalent programs may differ by matching on the same subterm at different places.

Need to quotient over that.
Enforce sum elimination \textbf{as early as possible}.

During goal-directed search, we don’t know yet which sums will be useful. (Type system: maximally-early introduction is a non-local criterion)

Cannot enforce early elimination of all useful subterms.
Intuition

Enforce sum elimination as early as possible.

During goal-directed search, we don’t know yet which sums will be useful. (Type system: maximally-early introduction is a non-local criterion)

Cannot enforce early elimination of all useful subterms.

Just eliminate all possible sums: saturation.
Demo time

Implementation available:
https://gitlab.com/gasche/unique-inhabitant

\[ f : (X \to Y + Y), x : X \]

\[ \vdash \]

\[ ? : X \]
Demo time

Implementation available:
https://gitlab.com/gasche/unique-inhabitant

\[ f : (X \to Y + Y), x : X \]

\[ \vdash \]

\[ ? : X \]
Demo time

Implementation available:
https://gitlab.com/gasche/unique-inhabitant

\[ f : (X \to Y + Y), x : X \]

\[ \vdash \]

\[
\text{let } z^{Y+Y} = f \, x \text{ in } ? : X
\]
Demo time

Implementation available:
https://gitlab.com/gasche/unique-inhabitant

\[ f : (X \to Y + Y), x : X \]

\[ \vdash \]

\[
\text{let } z^{Y+Y} = f x \text{ in } ? : X
\]
Demo time

Implementation available:
https://gitlab.com/gasche/unique-inhabitant

\[ f : (X \rightarrow Y + Y), x : X \]

\[ \vdash \]

\[
\text{let } z^{Y+Y} = f \times \text{ in match } z \text{ with } \begin{array}{l}
\text{L } y_1^Y \rightarrow ? : X \\
\text{R } y_2^Y \rightarrow ? : X
\end{array}
\]

Final result: zero, one or two (distinct) terms.
Demo time

Implementation available:
https://gitlab.com/gasche/unique-inhabitant

\[ f : (X \rightarrow Y + Y), \ x : X \]

\[
\begin{align*}
\text{let } z^{Y+Y} &= f \ x \ \text{in match } z \ \text{with} \\
&\quad \begin{cases}
L \ y_1^Y \rightarrow ? : X \\
R \ y_2^Y \rightarrow ? : X
\end{cases}
\end{align*}
\]

Final result: zero, one or two (distinct) terms.
Demo time

Implementation available:
https://gitlab.com/gasche/unique-inhabitant

\[ f : (X \rightarrow Y + Y), x : X \]

\[
\begin{array}{l}
\hline
\text{let } z^{Y+Y} = f \ x \ \text{in match } z \ \text{with} \\
\hline
\text{L } y_1^{Y} \rightarrow ? : X \\
\text{R } y_2^{Y} \rightarrow ? : X
\end{array}
\]

Final result: zero, one or two (distinct) terms.
Demo time

Implementation available:
https://gitlab.com/gasche/unique-inhabitant

\[ f : (X \rightarrow Y + Y), x : X \]

\[
\begin{align*}
\text{let } z^{Y+Y} &= f \times \text{ in match } z \text{ with } &
\text{L } y_1^{Y} \rightarrow x \\
\text{R } y_2^{Y} \rightarrow ? : X
\end{align*}
\]

Final result: zero, one or two (distinct) terms.
Demo time

Implementation available:
https://gitlab.com/gasche/unique-inhabitant

\[ f : (X \rightarrow Y + Y), \ x : X \]

\[ \vdash \]

\[
\text{let } z^{Y+Y} = f x \text{ in match } z \text{ with }
\]

| \text{L } y_1^{Y} \rightarrow x |
| \text{R } y_2^{Y} \rightarrow x |
Demo time

Implementation available:
https://gitlab.com/gasche/unique-inhabitant

\[ f : (X \rightarrow Y + Y), x : X \]

\[ \vdash \]

```
let z^{Y+Y} = f x in match z with
            | L y_1^{Y \rightarrow x}
            | R y_2^{Y \rightarrow x}
```

Final result: zero, one or two (distinct) terms.
Saturation

We alternate goal-directed (backward) search and (forward) saturation.

Saturation of $\Gamma$: compute all possible neutral terms $\Gamma \vdash n : A + B$ and deconstruct (some of) them.
Saturation

We alternate goal-directed (backward) search and (forward) saturation.

Saturation of $\Gamma$: compute **all possible** neutral terms $\Gamma \vdash n : A + B$ and deconstruct (some of) them.

**freshness condition**: neutrals typeable in a strictly smaller $\Gamma$ are old, don’t deconstruct them again

$\Rightarrow$ canonicity
Saturation

We alternate goal-directed (backward) search and (forward) saturation.

Saturation of $\Gamma$: compute all possible neutral terms $\Gamma \vdash n : A + B$ and deconstruct (some of) them.

- **freshness condition**: neutrals typeable in a strictly smaller $\Gamma$ are old, don’t deconstruct them again

  $\Rightarrow$ canonicity

- **subformula property**: the sums $(A + B)$ that appear in $\Gamma$ suffice

- **two-or-more property**: at most two different neutrals of each type suffice

  $\Rightarrow$ termination
Conclusion

We build upon proof theory and logic programming – focusing (bidirectional typing, better).

Contribution: a focused saturating proof/type system, canonical and computationally complete for STLC with sums.
⇒ decidability of unique inhabitation
⇒ new insights on program equivalence (empty type?)

In the paper: other practical examples, detailed related work.
Conclusion

We build upon proof theory and logic programming – focusing (bidirectional typing, better).

**Contribution:** a focused **saturating** proof/type system, canonical and computationally complete for STLC with sums.

⇒ **decidability** of unique inhabitation
⇒ new insights on program equivalence (empty type?)

In the paper: other practical examples, detailed related work.

Future work: extend to polymorphism and... dependent types.
Conclusion

We build upon proof theory and logic programming – **focusing** (bidirectional typing, better).

Contribution: a focused **saturating** proof/type system, canonical and computationally complete for STLC with sums.
⇒ **decidability** of unique inhabitation
⇒ new insights on program equivalence (empty type?)

In the paper: other practical examples, detailed related work.

Future work: extend to polymorphism and... dependent types.

Thanks. Any question?