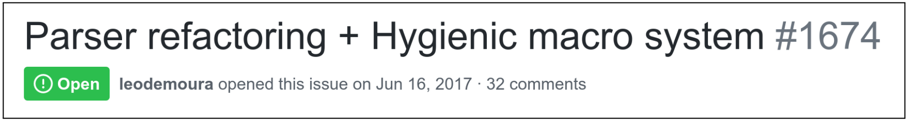
Beyond Notations: Hygienic Macro Expansion for Theorem Proving Languages

Sebastian Ullrich[1], Leonardo de Moura[2]

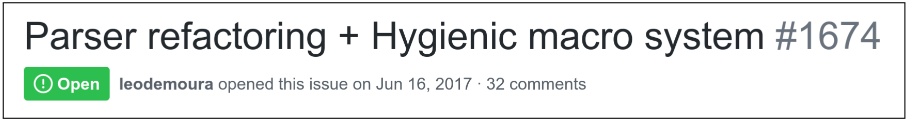
[1]KIT, Germany [2]Microsoft Research, USA

It's been a long time coming...


Parser refactoring + Hygienic macro system #1674

Open leodemoura opened this issue on Jun 16, 2017 · 32 comments

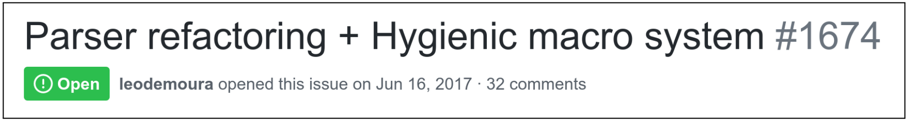
It's been a long time coming...


Parser refactoring + Hygienic macro system #1674

leodemoura opened this issue on Jun 16, 2017 · 32 comments

"We should really refactor the elaborator as well"

It's been a long time coming...

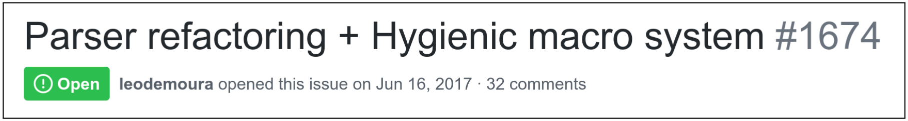

Parser refactoring + Hygienic macro system #1674

Open leodemoura opened this issue on Jun 16, 2017 · 32 comments

"We should really refactor the elaborator as well"

"If we rewrite the frontend, we should do that in Lean"

It’s been a long time coming...


Parser refactoring + Hygienic macro system #1674

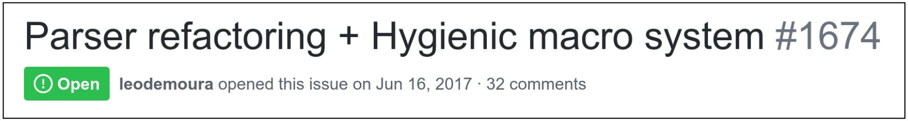
leodemoura opened this issue on Jun 16, 2017 · 32 comments

“We should really refactor the elaborator as well”

“If we rewrite the frontend, we should do that in Lean”

“We first need a capable Lean compiler for that...”

It's been a long time coming...


Parser refactoring + Hygienic macro system #1674

Open leodemoura opened this issue on Jun 16, 2017 · 32 comments

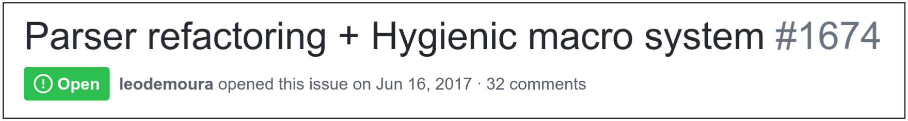
"We should really refactor the elaborator as well"

"If we rewrite the frontend, we should do that in Lean"

"We first need a capable Lean compiler for that..."

Thus the Lean 4 project was born.

It's been a long time coming...


Parser refactoring + Hygienic macro system #1674

Open leodemoura opened this issue on Jun 16, 2017 · 32 comments

"We should really refactor the elaborator as well"

"If we rewrite the frontend, we should do that in Lean"

"We first need a capable Lean compiler for that..."

Thus the Lean 4 project was born.

What issues could be that important?

Issues with existing syntax sugar systems

- Restricted to term level

```
notation Γ `⊢` e `:` τ := Typing Γ e τ
```

Issues with existing syntax sugar systems

- Restricted to term level

```
notation Γ `⊢` e `:` τ := Typing Γ e τ
```

- Restricted inputs

```
notation `∃` binder `,` r:(scoped P, Exists P) := r
```

Issues with existing syntax sugar systems

- Restricted to term level

```
notation Γ `⊢` e `:` τ := Typing Γ e τ
```

- Restricted inputs

```
notation `∃` binder `,` r:(scoped P, Exists P) := r
```

```
Notation "∃ x , P" := (exists (fun x => P)).
```

Issues with existing syntax sugar systems

- Restricted to term level

```
notation Γ `⊢` e `:` τ := Typing Γ e τ
```

- Restricted inputs

```
notation `∃` binder `,` r:(scoped P, Exists P) := r
```

```
Notation "\sum_ ( i <- r ) F"         := (\big[addn/0]_(i <- r) F).
Notation "\sum_ ( i <- r | P ) F"     := (\big[addn/0]_(i <- r | P) F).
Notation "\sum_ ( m <= i < n | P ) F" := (\big[addn/0]_(m <= i < n | P) F).
...
Notation "\mul_ ( i <- r ) F"         := (\big[muln/0]_(i <- r) F).
Notation "\mul_ ( i <- r | P ) F"     := (\big[muln/0]_(i <- r | P) F).
Notation "\mul_ ( m <= i < n | P ) F" := (\big[muln/0]_(m <= i < n | P) F).
...
```

Issues with existing syntax sugar systems

- Restricted to term level

```
notation Γ `⊢` e `:` τ := Typing Γ e τ
```

- Restricted inputs

```
notation `∃` binder `,` r:(scoped P, Exists P) := r
```

- Low-level might exist, but separate system!

```
@[user_notation] meta def format_macro (_ : parse $ tk "format!") (s : string) :
  parser pexpr := ...
```

A unified frontend system

A unified frontend system

Notations *Term* $\rightarrow$ *Term*

```
notation "∃" b "," P => Exists (fun b => P)
```

A unified frontend system

Notations	*Term* $\rightarrow$ *Term*	`notation "∃" b "," P => Exists (fun b => P)`
$\Downarrow$ Macros	*Surf* $\rightarrow$ *Surf*	`macro "∃" b:term "," P:term : term =>` `` `(Exists (fun $b => $P)) ``

A unified frontend system

Notations $\quad Term \to Term$

```
notation "∃" b "," P => Exists (fun b => P)
```

$\Downarrow$

Macros $\quad Surf \to Surf$

```
macro "∃" b:term "," P:term : term =>
`(Exists (fun $b => $P))
```

$\Downarrow$

Elaborators $\quad Surf \to Core$

```
elab "∃" b:term "," P:term : term =>
`(Exists (fun $b => $P)) >>= elabTerm
```

A unified frontend system

Notations *Term* $\rightarrow$ *Term*

```
notation "∃" b "," P => Exists (fun b => P)
```

$\Downarrow$

Macros *Surf* $\rightarrow$ *Surf*

```
macro "∃" b:term "," P:term : term =>
`(Exists (fun $b => $P))
```

$\Downarrow$

Elaborators *Surf* $\rightarrow$ *Core*

```
elab "∃" b:term "," P:term : term =>
`(Exists (fun $b => $P)) >>= elabTerm
```

Equal hygiene guarantees for all levels

Hygiene

```
notation "const" e => fun x => e
```

“Of course” *e* may not capture *x*

Hygiene

```
notation "const" e => fun x => e
```

“Of course” *e* may not capture *x*

```
macro "elab" ... => do
  ...;
  `(@[$elabAttr] def elabFn (stx : Syntax) : $type := match_syntax stx with ...)
```

“Of course” *elabFn* may not be captured from outside

Hygiene

```
notation "const" e => fun x => e
```

"Of course" *e* may not capture *x*

```
macro "elab" ... => do
  ...;
  `(@[$elabAttr] def elabFn (stx : Syntax) : $type := match_syntax stx with ...)
```

"Of course" *elabFn* may not be captured from outside

⇒ Hygienic macros introduce scopes!

Hygiene system

Main inspiration: *Binding as Sets of Scopes*, Matthew Flatt, POPL'16

Hygiene system

Main inspiration: *Binding as Sets of Scopes*, Matthew Flatt, POPL'16

Streamlined & optimized for slightly simpler macro system:
no local macros, no mutual recursion between decls and macros

Hygiene system

In essence:

1. *Remember* the surrounding scope in syntax quotations

```
`(def elabFn{} (stx{} : Syntax{Lean.Syntax}) ...)
```

Hygiene system

In essence:

1. *Remember* the surrounding scope in syntax quotations

```
`(def elabFn{} (stx{} : Syntax{Lean.Syntax}) ...)
```

2. *Tag* names introduced by macros

```
def elabFn.23{} (stx.23{} : Syntax.23{Lean.Syntax}) ...
```

(sequences of tags become important in macro-macros!)

Hygiene system

In essence:

1. *Remember* the surrounding scope in syntax quotations

```
`(def elabFn{} (stx{} : Syntax{Lean.Syntax}) ...)
```

2. *Tag* names introduced by macros

```
def elabFn.23{} (stx.23{} : Syntax.23{Lean.Syntax}) ...
```

(sequences of tags become important in macro-macros!)

Both actually implemented inside the `` `(...) `` macro!

Adapted name resolution

1. If tagged name is in local context, use it

```
... (stx.23{} : ...) := match_syntax stx.23{} ...
```

Implementation unchanged from basic name resolution

Adapted name resolution

1. If tagged name is in local context, use it

```
... (stx.23{} : ...) := match_syntax stx.23{} ...
```

   Implementation unchanged from basic name resolution

2. Otherwise, check global scopes and remembered names if any

```
Syntax.23{Lean.Syntax}
```

Adapted name resolution

1. If tagged name is in local context, use it

```
... (stx.23{} : ...) := match_syntax stx.23{} ...
```

Implementation unchanged from basic name resolution

2. Otherwise, check global scopes and remembered names if any

```
Syntax.23{Lean.Syntax}
```

3. Otherwise fail

Examples: Lean 4 elaborator

Examples: Lean 4 elaborator

```
syntax "if" optIdent term "then" term "else" term : term
macro_rules
| `(if $h : $cond then $t else $e) => `(dite $cond (fun $h => $t) (fun $h => $e))
| `(if $cond then $t else $e)      => `(ite $cond $t $e)
```

Examples: Lean 4 elaborator

$$term ::= ...|\langle term, ..., term\rangle$$

Examples: Lean 4 elaborator

$$\mathit{term} ::= \ldots \mid \langle \mathit{term}, \ldots, \mathit{term} \rangle$$

$$\frac{\Gamma \vdash \tau \equiv I\,\bar{p} \qquad c \text{ is single constructor of } I \qquad \Gamma \vdash c\,\bar{t} \Leftarrow \tau \rightsquigarrow t'}{\Gamma \vdash \langle \bar{t} \rangle \Leftarrow \tau \rightsquigarrow t'}$$

Examples: Lean 4 elaborator

$$\mathit{term} ::= \ldots \,|\, \langle \mathit{term}, \ldots, \mathit{term} \rangle$$

$$\frac{\Gamma \vdash \tau \equiv I\ \overline{p} \qquad c \text{ is single constructor of } I \qquad \Gamma \vdash c\ \overline{t} \Leftarrow \tau \rightsquigarrow t'}{\Gamma \vdash \langle \overline{t} \rangle \Leftarrow \tau \rightsquigarrow t'}$$

```
elab "⟨" args:(sepBy term ", ") "⟩" : term <= τ => do
  τ ← whnf τ;
  match τ.getAppFn with
  | Expr.const I _ _ => do
    ctors ← getCtors I;
    match ctors with
    | [c] => do
      stx ← `($(mkCTermId c) $(getSepElems args.getArgs)*);
      elabTerm stx τ
  ... -- error handling
```

Examples: Lean 4 elaborator

$$\mathit{term} ::= \ldots \mid \langle \mathit{term}, \ldots, \mathit{term} \rangle$$

$$\frac{\Gamma \vdash \tau \equiv I\ \bar{p} \qquad c \text{ is single constructor of } I \qquad \Gamma \vdash c\ \bar{t} \Leftarrow \tau \rightsquigarrow t'}{\Gamma \vdash \langle \bar{t} \rangle \Leftarrow \tau \rightsquigarrow t'}$$

```
elab "⟨" args:(sepBy term ", ") "⟩" : term <= τ => do
  τ ← whnf τ;
  match τ.getAppFn with
  | Expr.const I _ _ => do
    ctors ← getCtors I;
    match ctors with
    | [c] => do
      stx ← `($(mkCTermId c) $(getSepElems args.getArgs)*);
      elabTerm stx τ
  ... -- error handling
```

Examples: Lean 4 elaborator

$$\textit{term} ::= \ldots \mid \langle \textit{term}, \ldots, \textit{term} \rangle$$

$$\frac{\Gamma \vdash \tau \equiv I\, \overline{p} \qquad c \text{ is single constructor of } I \qquad \Gamma \vdash c\, \overline{t} \Leftarrow \tau \rightsquigarrow t'}{\Gamma \vdash \langle \overline{t} \rangle \Leftarrow \tau \rightsquigarrow t'}$$

```
elab "⟨" args:(sepBy term ", ") "⟩" : term <= τ => do
  τ ← whnf τ;
  match τ.getAppFn with
  | Expr.const I _ _ => do
    ctors ← getCtors I;
    match ctors with
    | [c] => do
      stx ← `($(mkCTermId c) $(getSepElems args.getArgs)*);
      elabTerm stx τ
  ... -- error handling
```

Examples: Lean 4 elaborator

$$term ::= \ldots \mid \langle term, \ldots, term \rangle$$

$$\frac{\Gamma \vdash \tau \equiv I\,\overline{p} \qquad c \text{ is single constructor of } I \qquad \Gamma \vdash c\,\overline{t} \Leftarrow \tau \rightsquigarrow t'}{\Gamma \vdash \langle \overline{t} \rangle \Leftarrow \tau \rightsquigarrow t'}$$

```
elab "⟨" args:(sepBy term ", ") "⟩" : term <= τ => do
  τ ← whnf τ;
  match τ.getAppFn with
  | Expr.const I _ _ => do
    ctors ← getCtors I;
    match ctors with
    | [c] => do
      stx ← `($(mkCTermId c) $(getSepElems args.getArgs)*);
      elabTerm stx τ
  ... -- error handling
```

Examples: Lean 4 elaborator

$$\mathit{term} ::= \ldots \mid \langle \mathit{term}, \ldots, \mathit{term} \rangle$$

$$\frac{\Gamma \vdash \tau \equiv I\,\overline{p} \qquad c \text{ is single constructor of } I \qquad \Gamma \vdash c\,\overline{t} \Leftarrow \tau \rightsquigarrow t'}{\Gamma \vdash \langle \overline{t} \rangle \Leftarrow \tau \rightsquigarrow t'}$$

```
elab "⟨" args:(sepBy term ", ") "⟩" : term <= τ => do
  τ ← whnf τ;
  match τ.getAppFn with
  | Expr.const I _ _ => do
    ctors ← getCtors I;
    match ctors with
    | [c] => do
      stx ← `($(mkCTermId c) $(getSepElems args.getArgs)*);
      elabTerm stx τ
  ... -- error handling
```

Examples: simple web server

```
import Webserver

GET / => redirect "/greet/stranger"

GET /greet/{name} => write
  <html>
    <h1>Hello, {name}!</h1>
  </html>

def main : IO Unit := do
  hIn ← IO.stdin;
  hOut ← IO.stdout;
  Webserver.run hIn hOut
```

https://leanprover.github.io/talks/PLDI20

Examples: simple web server

```
import Webserver

GET / => redirect "/greet/stranger"

GET /greet/{name} => write
  <html>
    <h1>Hello, {name}!</h1>
  </html>

def main : IO Unit := do
  hIn ← IO.stdin;
  hOut ← IO.stdout;
  Webserver.run hIn hOut
```

https://leanprover.github.io/talks/PLDI20

Tactic Hygiene

Lean 3 helper for proving injectivity of constructors:

```
def mk_inj_eq : tactic unit :=
`[intros, apply propext, apply iff.intro, ...]
```

Passable because no-one would ever redefine `iff.intro` ... right?

Tactic Hygiene

Lean 3 helper for proving injectivity of constructors:

```
def mk_inj_eq : tactic unit :=
`[intros, apply propext, apply iff.intro, ...]
```

Passable because no-one would ever redefine `iff.intro` ... right?

```
namespace hott
...
@[hott] def iff.intro : ...
...
inductive ... -- breaks!
```

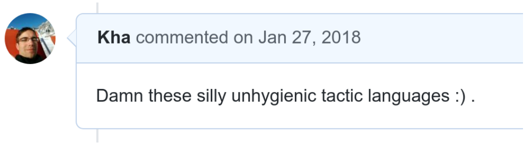
Tactic Hygiene

Lean 3 helper for proving injectivity of constructors:

```
def mk_inj_eq : tactic unit :=
`[intros, apply propext, apply iff.intro, ...]
```

Passable because no-one would ever redefine `iff.intro` ... right?

```
namespace hott
...
@[hott] def iff.intro : ...
...
inductive ... -- breaks!
```



Kha commented on Jan 27, 2018

Damn these silly unhygienic tactic languages :) .

Tactic Hygiene

Lean 4:

```
macro mkInjEq : tactic =>
`(intros; apply propext; apply Iff.intro; ...)
```

Tactic macros expanded on the fly by a new tactic interpreter
Same hygiene guarantees as with other macros

Tactic Hygiene

Lean 4:

```
macro mkInjEq : tactic =>
`(intros; apply propext; apply Iff.intro; ...)
```

Tactic macros expanded on the fly by a new tactic interpreter
Same hygiene guarantees as with other macros

```
macro introH : tactic => `(intro h)
lemma ... by introH; exact h  -- breaks!
```

Tactic Hygiene

Lean 4:

```
macro mkInjEq : tactic =>
`(intros; apply propext; apply Iff.intro; ...)
```

Tactic macros expanded on the fly by a new tactic interpreter
Same hygiene guarantees as with other macros

```
macro introH : tactic => `(intro h)
lemma ... by introH; exact h  -- breaks!
```

```
macro introH : tactic => `(intro $(mkIdent `h))
lemma ... by introH; exact h  -- works!
```

Conclusion

- A tower of abstractions from notations down to elaborators
- A simple, non-invasive but effective macro hygiene system
- The first hygienic tactic system of its kind

Conclusion

- A tower of abstractions from notations down to elaborators
- A simple, non-invasive but effective macro hygiene system
- The first hygienic tactic system of its kind

Thank you!