Beyond Notations: Hygienic Macro Expansion for Theorem Proving Languages

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

[1] Karlsruhe Institute of Technology, Germany
sebastian.ullrich@kit.edu
[2] Microsoft Research, USA
leonardo@microsoft.com

Abstract. In interactive theorem provers (ITPs), extensible syntax is not only crucial to lower the cognitive burden of manipulating complex mathematical objects, but plays a critical role in developing reusable abstractions in libraries. Most ITPs support such extensions in the form of restrictive "syntax sugar" substitutions and other ad hoc mechanisms, which are too rudimentary to support many desirable abstractions. As a result, libraries are littered with unnecessary redundancy. Tactic languages in these systems are plagued by a seemingly unrelated issue: accidental name capture, which often produces unexpected and counterintuitive behavior. We take ideas from the Scheme family of programming languages and solve these two problems simultaneously by proposing a novel *hygienic macro system* custom-built for ITPs. We further describe how our approach can be extended to cover type-directed macro expansion resulting in a single, uniform system offering multiple abstraction levels that range from supporting simplest syntax sugars to elaboration of formerly baked-in syntax. We have implemented our new macro system and integrated it into the upcoming version (v4) of the Lean theorem prover. Despite its expressivity, the macro system is simple enough that it can easily be integrated into other systems.

1 Introduction

Mixfix notation systems have become an established part of many modern ITPs for attaching terse and familiar syntax to functions and predicates of arbitrary arity.

```
_⊢_:_ = Typing                                          Agda
Notation "Ctx ⊢ E : T" := (Typing Ctx E T).             Coq
notation typing ("_ ⊢ _ : _")                           Isabelle
notation Γ `⊢` e `:` τ := Typing Γ e τ                  Lean 3
```

As a further extension, all shown systems also allow *binding names* inside mixfix notations.

```
syntax ∃ A (λ x → P) = ∃[ x ∈ A ] P                          Agda
Notation "∃ x , P" := (exists (fun x ⇒ P)).                  Coq
notation exists (binder "∃")                                 Isabelle
notation `∃` binder `,` r:(scoped P, Exists P) := r          Lean 3
```

While these extensions differ in the exact syntax used, what is true about all of them is that at the time of the notation declaration, the system already, statically knows what parts of the term are bound by the newly introduced variable. This is in stark contrast to *macro systems* in Lisp and related languages where the expansion of a *macro* (a syntactic substitution) can be specified not only by a *template expression* with placeholders like above, but also by arbitrary *syntax transformers*, i.e. code evaluated at compile time that takes and returns a syntax tree[3]. As we move to more and more expressive notations and ideally remove the boundary between built-in and user-defined syntax, we argue that we should no more be limited by the static nature of existing notation systems and should instead introduce syntax transformers to the world of ITPs.

However, as usual, with greater power comes greater responsibility. By using arbitrary syntax transformers, we lose the ability to statically determine what parts of the macro template can be bound by the macro input (and vice versa). Thus it is no longer straightforward to avoid *hygiene* issues (i.e. accidental capturing of identifiers; [11]) by automatically renaming identifiers. We propose to learn from and adapt the macro hygiene systems implemented in the Scheme family of languages for interactive theorem provers in order to obtain more general but still well-behaved notation systems.

After giving a practical overview of the new, macro-based notation system we implemented in the upcoming version of Lean (Lean 4) in Section 2, we describe the issue of hygiene and our general hygiene algorithm, which should be just as applicable to other ITPs, in Section 3. Section 4 gives a detailed description of the implementation of this algorithm in Lean 4. In Section 5, we extend the use case of macros from mere syntax substitutions to type-aware elaboration. Finally, we have already encountered hygiene issues in the current version of Lean in a different part of the system: the tactic framework. We discuss how these issues are inevitable when implementing reusable tactic scripts and how our macro system can be applied to this hygiene problem as well in Section 6.

Contributions. We present a system for hygienic macros optimized for theorem proving languages as implemented[4] in the next version of the Lean theorem prover, Lean 4.

- We describe a novel, efficient hygiene algorithm to employ macros in ITP languages at large: a combination of a white-box, effect-based approach for detecting newly introduced identifiers and an efficient encoding of scope metadata.

[3] These two macro declaration styles are commonly referred to as *pattern-based* vs. *procedural*

[4] `https://github.com/leanprover/lean4/blob/IJCAR20/src/Init/Lean/Elab`

- We show how such a macro system can be seamlessly integrated into existing elaboration designs to support type-directed expansion even if they are not based on homogeneous source-to-source transformations.
- We show how hygiene issues also manifest in tactic languages and how they can be solved with the same macro system. To the best of our knowledge, the tactic language in Lean 4 is the first tactic language in an established theorem prover that is automatically hygienic in this regard.

2 The new macro system

Lean's current notation system as shown in Section 1 is still supported in Lean 4, but based on a much more general macro system; in fact, the `notation` keyword itself has been reimplemented as a macro, more specifically as a *macro-generating macro* making use of our tower of abstraction levels. The corresponding Lean 4 command[5] for the example from the previous section

```
notation Γ "⊢" e ":" τ ⇒ Typing Γ e τ
```

expands to the macro declaration

```
macro Γ:term "⊢" e:term ":" τ:term : term ⇒ `(Typing $Γ $e $τ)
```

where the *syntactic category* (`term`) of placeholders and of the entire macro is now specified explicitly. The right-hand side uses an explicit *syntax quasiquotation* to construct the syntax tree, with syntax placeholders (*antiquotations*) prefixed with `$`. As suggested by the explicit use of quotations, the right-hand side may now be an arbitrary Lean term computing a syntax object; in other words, there is no distinction between pattern-based and procedural macros in our system. We can now use this abstraction level to implement simple command-level macros, for example.

```
macro "defthunk" id:ident ":=" e:term : command ⇒
`(def $id:ident := Thunk.mk (fun _ ⇒ $e))
defthunk big := mkArray 100000 true
```

Syntactic categories can be specified explicitly for antiquotations as in `$id:ident` where otherwise ambiguous. `macro` itself is another command-level macro that, for our `notation` example, expands to two commands

```
syntax term "⊢" term ":" term : term
macro_rules
| `($Γ ⊢ $e : $τ) ⇒ `(Typing $Γ $e $τ)
```

that is, a pair of parser extension (which we will not further discuss in this paper) and syntax transformer. Our reason for ultimately separating these two concerns is that we can now obtain a well-structured syntax tree pre-expansion, i.e. a *concrete* syntax tree, and use it to implement source code tooling such

[5] All examples including full context can be found in the supplemental material at `https://github.com/Kha/macro-supplement`

as auto-completion, go-to-definition, and refactorings. Implementing even just the most basic of these tools for the Lean 3 frontend that combined parsing and notation expansion meant that they had to be implemented right inside the parser, which was not an extensible or even maintainable approach in our experience.

Both `syntax` and `macro_rules` are in fact further macros for regular Lean definitions encoding procedural metaprograms, though users should rarely need to make use of this lowest abstraction level explicitly. Both commands can only be used at the top level; we are not currently planning support for local macros.

There is no more need for the complicated `scoped` syntax since the desired translation can now be specified naturally, without any need for further annotations.

```
notation "∃" b "," P ⇒ Exists (fun b ⇒ P)
```

The lack of static restrictions on the right-hand side ensures that this works just as well with custom binding notations, even ones whose translation cannot statically be determined before substitution.

```
syntax "{" term "|" term "}" : term
macro_rules
| `({$x ∈ $s | $p}) ⇒ `(setOf (fun $x ⇒ $x ∈ $s ∧ $p))
| `({$b      | $p}) ⇒ `(setOf (fun $b ⇒ $p))

notation "⋃" b "," p ⇒ Union {b | p}
```

Here we explicitly make use of the `macro_rules` abstraction level for its convenient syntactic pattern matching syntax. `macro_rules` are "open" in the sense that multiple transformers for the same `syntax` declaration can be defined; they are tried in reverse declaration order by default up to the first match (though this can be customized using explicit priority annotations).

```
macro_rules
| `({$x ≤ $e | $p}) ⇒ `(setOf (fun $x ⇒ $x ≤ $e ∧ $p))
```

As a final example, we present a partial reimplementation of the arithmetic "bigop" notations found[6] in Coq's Mathematical Components library [12] such as

```
\sum_ (i <- [0, 2, 4] | i != 2) i
```

for summing over a filtered sequence of elements. The specific bigop notations are defined in terms of a single `\big_` fold operator; however, because Coq's notation system is unable to abstract over this new indexing syntax, every specific bigop notation has to redundantly repeat every specific index notation before delegating to `\big_`. In total, the 12 index notations for `\big_` are duplicated for 3 different bigops in the file.

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

[6] https://github.com/math-comp/math-comp/blob/master/mathcomp/ssreflect/bigop.v

In contrast, using our system, we can introduce a new syntactic category for index notations, interpret it once in `\big_`, and define new bigops on top of it without any redundancy.

```
declare_syntax_cat index
syntax ident "<-" term : index
syntax ident "<-" term "|" term : index
...

macro "Σ" "(" idx:index ")" F:term : term =>
`(\big_ [HasAdd.add, 0] ($idx:index) $F)
```

The full example is included as Appendix A and in the supplement.

3 Hygiene algorithm

In this section, we will give a mostly self-contained description of our algorithm for automatic hygiene applied to a simple recursive macro expander; we postpone comparisons to existing hygiene algorithms to Section 7.

Hygiene issues occur when transformations such as macro expansions lead to an unexpected capture (rebinding) of identifiers. For example, given the notation

```
notation "const" e ⇒ fun x ⇒ e
```

we would not expect the term `const x` to be closed because intuitively there is no `x` in scope at the argument position of `const`; that the implementation of the macro makes use of the name internally should be of no concern to the macro user.

Thus hygiene issues can also be described as a *confusion of scopes* when syntax parts are removed from their original context and inserted into new contexts, which makes name resolution strictly after macro expansion (such as in a compiler preceded by a preprocessor) futile. Instead we need to track scopes *as metadata* before and during macro expansion so as not to lose information about the original context of identifiers. Specifically,

1. when an identifier captured in a syntax quotation matches one or more top-level symbols[7], the identifier is annotated with a list of these symbols as *top-level* scopes to preserve its *extra-macro* context (which, because of the lack of local macros, can only contain top-level bindings), and
2. when a macro is expanded, all identifiers freshly introduced by the expansion are annotated with a new *macro* scope to preserve the *intra-macro* context. Macro scopes are appended to a list, i.e. ordered by expansion time. This full "history of expansions" is necessary to treat macro-producing macros correctly, as we shall see in Section 3.2.

Thus, the expansion of the above term is (an equivalent of) `fun x.1 ⇒ x` where `1` is a fresh macro scope appended to the macro-introduced `x`, preventing it

[7] Lean allows overloaded top-level bindings whereas local bindings are shadowing

from capturing the `x` from the original input. In general, we will style hygienic identifiers in the following as $\texttt{n.msc}_1\texttt{.msc}_2\texttt{.}\ldots\texttt{.msc}_n\texttt{\{tsc}_1\texttt{,}\ldots\texttt{,tsc}_n\texttt{\}}$ where `n` is the original name, `msc` are macro scopes, and `tsc` top-level scopes, eliding the braces if there are no top-level scopes as in the example above. We use the dot notation to suggest both the ordered nature of macro scopes and their eventual implementation in Section 4. We will now describe how to implement these operations in a standard macro expander.

3.1 Expansion algorithm

The macro expander described in this section bundles the execution of macros and insertion of their results with interspersed name resolution to track scopes and ensure hygiene of identifiers. As we shall see below, top-level scopes on binding names are always discarded by it. Thus we will define a *symbol* more formally as an identifier together with a list of macro scopes, such as `x.1` above.

Given a *global context* (a set of symbols), the expander does a conventional top-down expansion, keeping track of an initially-empty *local context* (another set of symbols). When a binding is encountered, the local context is extended with that symbol; top-level scopes on bindings are discarded since they are only meaningful on references. When a reference, i.e. an identifier not in binding position, is encountered, it is resolved according to the following rules:

1. If the local context has an entry for the same symbol, the reference binds to the corresponding local binding; any top-level scopes are ignored.
2. Otherwise, if the identifier is annotated with one or more top-level scopes or matches one or more symbols in the global context, it binds to all of these (to be disambiguated by the elaborator).
3. Otherwise, the identifier is unbound and an error is generated.

In the common incremental compilation mode of ITPs, every command is fully processed before subsequent commands. Thus, an expander for such a system will not extend the global context by itself, but pass the fully expanded command to the next compilation step before being called again with the next command's unexpanded syntax tree and a possibly extended global context.

Notably, our expander does not add macro scopes to identifiers by itself, either, much in contrast to other expansion algorithms. We instead delegate this task to the macro itself, though in a completely transparent way for all pattern-based and for many procedural macros. We claim that a macro should in fact be interpreted as an *effectful* computation since two expansions of the same identifier-introducing macro should not return the same syntax tree to avoid unhygienic interactions between them. Thus, as a *side effect*, it should apply a fresh macro scope to each captured identifier. In particular, a syntax quotation should not merely be seen as a datum, but implemented as an effectful value that obtains and applies this fresh scope to all the identifiers contained in it to immediately ensure hygiene for pattern-based macros. Procedural macros producing identifiers not originating from syntax quotations might need to obtain and make use of the fresh macro scope explicitly. We give a specific monad-based [14] implementation of effectful syntax quotations as a regular macro in Section 4.

3.2 Examples

Given the following input,

```
def x := 1
def e := fun y ⇒ x
notation "const" e ⇒ fun x ⇒ e
def y := const x
```

we incrementally parse, expand, and elaborate each declaration before advancing to the next one. For a first, trivial example, let us focus on the expansion of the second line. At this point, the global context contains the symbol `x` (plus any default imports that we will ignore here). Descending into the right-hand side of the definition, we first add `y` to the local context. The reference `x` does not match any local definitions, so it binds to the matching top-level definition.

In the next line, the built-in `notation` macro expands to the definitions

```
syntax "const" term : term
macro_rules
| `(const $e) ⇒ `(fun x ⇒ $e)
```

When a top-level macro application unfolds to multiple declarations, we expand and elaborate these incrementally as well to ensure that declarations are in the global context of subsequent declarations. When recursively expanding the `macro_rules` declaration (we will assume for this example that `macro_rules` itself is primitive) in the global context `{x, e}`, we first visit the syntax quotation on the left-hand side. The identifier `e` inside of it is in an antiquotation and thus not captured by the quotation. It is in binding position for the right-hand side, so we add `e` to the local context. Visiting the right-hand side, we find the quotation-captured identifier `x` and annotate it with the matching top-level definition of the same name; we do not yet know that it is in a binding position. When visiting the reference `e`, we see that it matches a local binding and do not add top-level scopes.

```
macro_rules
| `(const $e) ⇒ `(fun x{x} ⇒ $e)
```

Visiting the last line

```
def y := const x
```

with the global context `{x, e}`, we descend into the right-hand side. We expand the `const` macro given a fresh macro scope `2`, which is applied to any captured identifiers.

```
def y := fun x.2{x} ⇒ x
```

We add the symbol `x.2` (discarding the top-level scope `x`) to the local context and finally visit the reference `x`. The reference does not match the local binding `x.2` but does match the top-level binding `x`, so it binds to the latter.

```
def y := fun x.2 ⇒ x
```

Now let us briefly look at a more complex macro-macro example demonstrating use of the macro scopes stack:

```
macro "m" n:ident : command => `(
  def f := 1
  macro "mm" : command => `(def $n:ident := f    def f := $n:ident))
```

If we call `m f`, we apply a macro scope `1` to all captured identifiers, then incrementally process the two new declarations.

```
def f.1 := 1
macro "mm" : command => `(def f := f.1{f.1}    def f.1{f.1} := f)
```

If we call the new macro `mm`, we apply one more macro scope `2`.

```
def f.2 := f.1.2{f.1}    def f.1.2{f.1} := f.2
```

When processing these new definitions, we see that the scopes ensure the expected name resolution. In particular, we now have global declarations `f.1`, `f.2`, and `f.1.2` that show that storing only a single macro scope would have led to a collision.

4 Implementation

Syntax objects in Lean 4 are represented as an inductive type of *nodes* (or nonterminals), *atoms* (or terminals), and, as a special case of nonterminals, *identifiers*.

```
inductive Syntax
| node  (kind : Name) (args : Array Syntax)
| atom  (info : Option SourceInfo) (val : String)
| ident (info : Option SourceInfo) (rawVal : String) (val : Name)
    (preresolved : List (Nat × List String))
| missing
```

An additional constructor represents *missing* parts from syntax error recovery. Atoms and identifiers are annotated with source location metadata unless generated by a macro. Identifiers carry macro scopes inline in their `Name` while top-level scopes are held in a separate list. The additional `Nat` is an implementation detail of Lean's hierarchical name resolution.

The type `Name` of hierarchical names precedes the implementation of the macro system and is used throughout Lean's implementation for referring to (namespaced) symbols.

```
inductive Name
| anonymous : Name
| str : Name → String → Name
| num : Name → Nat    → Name
```

The syntax `` `a.b `` is a literal of type `Name` for use in meta-programs. The numeric part of `Name` is not accessible from the surface syntax and reserved for internal names; similar designs are found in other ITPs. By reusing `Name` for storing macro

scopes, but not top-level scopes, we ensure that the new definition of *symbol* from Section 3.1 coincides with the existing Lean type and no changes to the implementation of the local or global context are necessary for adopting the macro system.

A Lean 4 implementation of the expansion algorithm described in the previous section is given in Fig. 1; the full implementation including examples is included as Appendix B and in the supplement. As a generalization, syntax transformers have the type `Syntax` → `TransformerM Syntax` where the `TransformerM` monad gives access to the global context and a fresh macro scope per macro expansion. The expander itself uses an extended `ExpanderM` monad that also stores the local context and the set of registered macros. We use the Lean equivalent of Haskell's `do` notation [13] to program in these monads.

As usual, the expander has built-in knowledge of some "core forms" (lines 3-17) with special expansion behavior, while all other forms are assumed to be macros and expanded recursively (lines 20-22). Identifiers form one base case of the recursion. As described in the algorithm, they are first looked up in the local context (recall that the `val` of an identifier includes macro scopes), then as a fall back in the global context plus its own top-level scopes. `mkTermId : Name` → `Syntax` creates an identifier without source information or top-level scopes, which are not needed after expansion. `mkOverloadedConstant` implements the Lean special case of overloaded symbols to be disambiguated by elaboration; systems without overloading support should throw an ambiguity error instead in this case.

As an example of a core binding form, the expansion of a single-parameter `fun` is shown in lines 13-17 of Fig. 1. It recursively expands the given parameter type, then expands the body in a new local context extended with the value of `id`. Here `getIdentVal : Syntax` → `Name` in particular implements the discarding of top-level scopes from binders.

Finally, in the macro case, we fetch the syntax transformer for the given node kind, call it in a new context with a fresh current macro scope, and recurse.

Syntax quotations are given as one example of a macro: they do not have built-in semantics but transform into code that constructs the appropriate syntax tree (`expandStxQuot` in Fig. 2). More specifically, a syntax quotation will, at runtime, query the current macro scope `msc` from the surrounding `TransformerM` monad and apply it to all captured identifiers, which is done in `quoteSyntax`. `quoteSyntax` recurses through the quoted syntax tree, reflecting its constructors. Basic datatypes such as `String` and `Name` are turned into `Syntax` via the typeclass method `quote`. For antiquotations, we return their contents unreflected. In the case of identifiers, we resolve possible global references at compile time and reflect them, while `msc` is applied at runtime. Thus a quotation `` `(a + $b) `` inside a global context where the symbol `a` matches declarations `a.a` and `b.a` is transformed to the equivalent of

```
do msc ← getCurrMacroScope;
   pure (Syntax.node `plus
     [Syntax.ident none "a" (addMacroScope `a msc) [`a.a, `b.a],
      Syntax.atom none "+", b])
```

```
partial def expand : Syntax → ExpanderM Syntax
| stx ⇒ match_syntax stx with
  | `($id:ident) => do
    let val := getIdentVal id;
    gctx ← getGlobalContext;
    lctx ← getLocalContext;
    if lctx.contains val then
      pure (mkTermId val)
    else match resolve gctx val ++ getPreresolved id with
      | []        ⇒ throw ("unknown identifier " ++ toString val)
      | [(id, _)] ⇒ pure (mkTermId id)
      | ids       ⇒ pure (mkOverloadedIds ids)
  | `(fun ($id:ident : $ty) ⇒ $e) ⇒ do
    let val := getIdentVal id;
    ty ← expand ty;
    e ← withLocal val (expand e);
    `(fun ($(mkTermId val) : $ty) ⇒ $e)
  | ...  -- other core forms
  | _ => do
    t ← getTransformerFor stx.getKind;
    stx ← withFreshMacroScope (t stx);
    expand stx
```

Fig. 1. Abbreviated implementation of a recursive expander for our macro system

This implementation of syntax quotations itself makes use of syntax quotations for simplicity and thus is dependent on its own implementation in the previous stage of the compiler. Indeed, the helper variable `msc` must be renamed should the name already be in scope and used inside an antiquotation. Note that `quoteSyntax` is allowed to reference the same `msc` as `expandStxQuot` because they are part of the same macro call and the current macro scope is unchanged between them.

5 Integrating macros into elaboration

The macro system as described so far can handle most syntax sugars of Lean 3 except for ones requiring type information. For example, the *anonymous constructor* ⟨`e, ...`⟩ is sugar for `(c e ...)` if the expected type of the expression is known and it is an inductive type with a single constructor `c`. While trivial to parse, there is no way to implement this syntax as a macro if expansion is done strictly prior to elaboration. To the best of our knowledge, none of the ITPs listed in the introduction support hygienic elaboration extensions of this kind, but we will show how to extend their common elaboration scheme in that way in this section.

```
partial def quoteSyntax : Syntax → TransformerM Syntax
| Syntax.ident info rawVal val preresolved ⇒ do
  gctx ← getGlobalContext;
  let preresolved := resolve gctx val ++ preresolved;
  `(Syntax.ident none $(quote rawVal) (addMacroScope $(quote val) msc)
    $(quote preresolved))
| stx@(Syntax.node k args) ⇒
  if isAntiquot stx then pure (getAntiquotTerm stx)
  else do
    args ← args.mapM quoteSyntax;
    `(Syntax.node $(quote k) $(quote args))
| Syntax.atom info val ⇒ `(Syntax.atom none $(quote val))
| Syntax.missing ⇒ pure Syntax.missing

def expandStxQuot (stx : Syntax) : TransformerM Syntax := do
stx ← quoteSyntax (stx.getArg 1);
`(do msc ← getCurrMacroScope; pure $stx)
```

Fig. 2. Simplified syntax transformer for syntax quotations

Elaboration[8] can be thought of as a function `elabTerm : Syntax → ElabM Expr` in an appropriate monad `ElabM`[9] from a (concrete or abstract) surface-level syntax tree type `Syntax` to a fully-specified core term type `Expr` [15]. We have presented the (concrete) definition of `Syntax` in Lean 4 in Section 4; the particular definition of `Expr` is not important here. While such an elaboration system could readily be composed with a type-insensitive macro expander such as the one presented in Section 3, we would rather like to *intertwine* the two to support type-sensitive but still hygienic-by-default macros (henceforth called *elaborators*) without having to reimplement macros of the kind discussed so far. Indeed, these can automatically be adapted to the new type given an adapter between the two monads, similarly to the adaption of macros to *expanders* in [6]:

```
def transformerToElaborator (m : Syntax → TransformerM Syntax) :
    Syntax → ElabM Expr :=
fun stx ⇒ do stx' ← (transformerMToElabM m) stx; elabTerm stx'
```

Because most parts of our hygiene system are implemented by the expander for syntax quotations, the only changes to an elaboration system necessary for supporting hygiene are storing the current macro scope in the elaboration monad (to be passed to the expansion monad in the adapter) and allocating a fresh macro scope in `elabTerm` and other recursion points, which morally now represent the starting point of a macro's expansion. Thus elaborators immediately benefit from hygiene as well whenever they use syntax quotations to construct unelaborated helper syntax objects to pass to `elabTerm`. In order to support syntax quotations

[8] at the term level; other levels work analogously but with different output types
[9] or some other encoding of effects

in these two and other monads, we generalize their implementation to a new monad typeclass implemented by both monads.

```
class MonadQuotation (m : Type → Type) :=
(getCurrMacroScope : m MacroScope)
(withFreshMacroScope {α : Type} : m α → m α)
```

The second operation is not used by syntax quotations directly, but can be used by procedural macros to manually enter new macro call scopes.

As an example, the following is a simplified implementation of the anonymous constructor syntax mentioned above.

```
@[termElab anonymousCtor]
def elabAnonymousCtor (stx : Syntax) : ElabM Expr :=
match_syntax stx with
| `(⟨$args*⟩) => do
  expectedType ← getExpectedType;
  match Expr.getAppFn expectedType with
  | Expr.const constName _ _ => do
    ctors ← getCtors constName;
    match ctors with
    | [ctor] => do
      stx ← `($(mkCTermId ctor) $(getSepElems args)*);
      elabTerm stx
...  -- error handling
```

The `[termElab]` attribute registers this elaborator for the given syntax node kind. `$args*` is an antiquotation *splice* that extracts/injects a syntactic sequence of elements into/from an `Array Syntax`. The array by default includes separators such as "," as `Syntax.atom`s in order to be lossless, which we here filter out using `getSepElems`. `mkCTermId : Name → Syntax` synthesizes a hygienic reference to the given constant name by storing it as a top-level scope and applying a reserved macro scope to the constructed identifier.

This implementation fails if the expected type is not yet sufficiently known at this point. The actual implementation[10] of this elaborator extends the code by *postponing* elaboration until the expected type is known.

6 Tactic hygiene

Lean 3 includes a tactic framework that, much like macros, allows users to write custom automation either procedurally inside a `Tactic` monad (renamed to `TacticM` in Lean 4) or "by example" using tactic language quotations, or in a mix of both [9]. For example, Lean 3 uses a short tactic block to prove injection lemmas for data constructors.

```
def mkInjEq : Tactic Unit :=
`[intros; apply propext; apply Iff.intro; ...]
```

[10] https://github.com/leanprover/lean4/blob/IJCAR20/src/Init/Lean/Elab/BuiltinNotation.lean#L43

Unfortunately, this code unexpectedly broke in Lean 3 when used from a library for homotopy type theory that defined its own `propext` and `Iff.intro` declarations[11]; in other words, Lean 3 tactic quotations are unhygienic and required manual intervention in this case. Just like with macros, the issue with tactics is that binding structure in such embedded terms is not known at declaration time. Only at tactic run time do we know all local variables in the current context that preceding tactics may have added or removed, and therefore the scope of each captured identifier.

Arguably, the Lean 3 implementation also exhibited a lack of hygiene in the handling of tactic-introduced identifiers: it did not prevent users from referencing such an identifier outside of the scope it was declared in.

```
def myTac : Tactic Unit := `[intro h]
lemma triv (p : Prop) : p → p := begin myTac; exact h end
```

Coq's similar Ltac tactic language [5] exhibits the same issue and users are advised not to introduce fixed names in tactic scripts but to generate fresh names using the `fresh` tactic first[12], which can be considered a manual hygiene solution.

Lean 4 instead extends its automatically hygienic macro implementation to tactic scripts by allowing regular macros in the place of tactic invocations.

```
macro "myTac" : tactic ⇒ `(intro h; exact h)
theorem triv (p : Prop) : p → p := begin myTac end
```

By the same hygiene mechanism described above, introduced identifiers such as `h` are renamed so as not to be accessible outside of their original scope, while references to global declarations are preserved as top-level scope annotations. Thus Lean 4's tactic framework resolves both hygiene issues discussed here without requiring manual intervention by the user. Expansion of tactic macros in fact does not precede but is integrated into the *tactic evaluator* `evalTactic : Syntax → TacticM Unit` such that recursive macro calls are expanded lazily.

```
syntax "repeat" tactic : tactic
macro_rules
| `(tactic| repeat $t) => `(tactic| try ($t; repeat $t))
```

Here the *quotation kind* `tactic` followed by a pipe symbol specifies the parser to use for the quotation, since tactic syntax may otherwise overlap with term syntax. `macro` automatically infers it from the given syntax category, but cannot be used here because the parser for `repeat` would not yet be available in the right-hand side. When `$t` eventually fails, the recursion is broken without visiting and expanding the subsequent `repeat` macro call. The `try` tactical is used to ignore this eventual failure.

While we believe that macros will cover most use cases of tactic quotations in Lean 3, their use within larger `TacticM` metaprograms can be recovered by passing such a quotation to `evalTactic`:

[11] https://github.com/leanprover/lean/pull/1913
[12] https://github.com/coq/coq/issues/9474

```
def myTac2 : TacticM Unit :=
do stx ← `(tactic|intro h; exact h); evalTactic stx
```

`TacticM` implements the `MonadQuotation` typeclass for this purpose.

7 Related work

The main inspiration behind our hygiene implementation was Racket's new *Sets of Scopes* [10] hygiene algorithm. Much like in our approach, Racket annotates identifiers both with scopes from their original context as well as with additional macro scopes when introduced by a macro expansion. However, there are some significant differences: Racket stores both types of scopes in a homogeneous, unordered set and does name resolution via a maximum-subset check. For both simplicity of implementation and performance, we have reduced scopes to the bare minimal representation using only strict equality checks, which we can easily encode in our existing `Name` implementation. In particular, we only apply scopes to matching identifiers and only inside syntax quotations. This optimization is of special importance because top-level declarations in Lean and other ITPs are not part of a single, mutually recursive scope as in Racket, but each open their own scope over all subsequent declarations, which would lead to a total number of scope annotations quadratic in the number of declarations using the Sets of Scopes algorithm. Finally, Racket detects macro-introduced identifiers using a "black-box" approach without the macro's cooperation following the marking approach of [11]: a fresh macro scope is applied to all identifiers in the macro input, then inverted on the macro output. While elegant, a naive implementation of this approach can result in quadratic runtime compared to unhygienic expansion and requires further optimizations in the form of lazy scope propagation [7], which is difficult to implement in a pure language such as Lean. Our "white-box" approach based on the single primitive of an effectful syntax quotation, while slightly easier to escape from in procedural syntax transformers, is simple to implement, incurs minimal overhead, and is equivalent for pattern-based macros.

The idea of automatically handling hygiene in the macro, and not in the expander, was introduced in [4], though only for pattern-based macros. MetaML [18] refined this idea by tying hygiene more specifically to syntax quotations that could be used in larger metaprogram contexts, which Template Haskell [17] interpreted as effectful (monadic) computations requiring access to a fresh-names generator, much like in our design. However, both of the latter systems should perhaps be characterized more as metaprogramming frameworks than Scheme-like macro systems: there are no "macro calls" but only explicit splices and so only built-in syntax with known binding semantics can be captured inside syntax quotations. Thus the question of which captured identifiers to rename becomes trivial again, just like in the basic notation systems discussed in Section 1.

While the vast majority of research on hygienic macro systems has focused on S-expression-based languages, there have been previous efforts on marrying that research with non-parenthetical syntax, with different solutions for combining syntax tree construction and macro expansion. The Dylan language requires

macro syntax to use predefined terminators and eagerly scans for the end of a macro call using this knowledge [2], while in Honu [16] the syntactic structure of a macro call is discovered during expansion by a process called "enforestation". The Fortress [1] language strictly separates the two concerns into grammar extensions and transformer declarations, much like we do. Dylan and Fortress are restricted to pattern-based macro declarations and thus can make use of simple hygiene algorithms while Honu uses the full generality of the Racket macro expander. On the other hand, Honu's authors "explicitly trade expressiveness for syntactic simplicity" [16]. In order to express the full Lean language and desirable extensions in a macro system, we require both unrestricted syntax of macros and procedural transformers.

Many theorem provers such as Coq, Agda, Idris, and Isabelle not already based on a macro-powered language provide restricted syntax extension mechanisms, circumventing hygiene issues by statically determining binding as seen in Section 1. Extensions that go beyond that do not come with automatic hygiene guarantees. Agda's macros[13], for example, operate on the De Bruijn index-based core term level and are not hygienic[14]. The ACL2 prover in contrast uses a subset of Common Lisp as its input language and adapts the hygiene algorithm of [7] based on renaming [8]. The experimental Cur [3] theorem prover is a kind of dual to our approach: it takes an established language with hygienic macros, Racket, and extends it with a dependent type system and theorem proving tools. ACL2 does not support tactic scripts, while their potential support in Cur is unclear at the time of writing[15].

8 Conclusion

We have proposed a new macro system for interactive theorem provers that enables syntactic abstraction and reuse far beyond the usual support of mixfix notations. Our system is based on a novel hygiene algorithm designed with a focus on minimal runtime overhead as well as ease of integration into pre-existing codebases, including integration into standard elaboration designs to support type-directed macro expansion. Despite that, the algorithm is general enough to provide a complete hygiene solution for pattern-based macros and provides flexible hygiene for procedural macros. We have also demonstrated how our macro system can address unexpected name capture issues that haunt existing tactic frameworks. We have implemented our method in the upcoming version (v4) of the Lean theorem prover; it should be sufficiently attractive and straightforward to implement to be adopted by other interactive theorem proving systems as well.

[13] `https://agda.readthedocs.io/en/v2.6.0.1/language/reflection.html#macros`
[14] `https://github.com/agda/agda/issues/3819`
[15] `https://github.com/wilbowma/cur/issues/104`

Acknowledgments. We are very grateful to the anonymous reviewers, David Thrane Christiansen, Gabriel Ebner, Matthew Flatt, Sebastian Graf, Alexis King, Daniel Selsam, and Max Wagner for extensive comments, corrections, and advice.

References

1. Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessen, J.W., Ryu, S., Steele Jr, G.L., Tobin-Hochstadt, S., Dias, J., Eastlund, C., et al.: The Fortress language specification. Sun Microsystems **139**(140), 116 (2005)
2. Bachrach, J., Playford, K., Street, C.: D-expressions: Lisp power, Dylan style. Style DeKalb IL (1999)
3. Chang, S., Ballantyne, M., Turner, M., Bowman, W.J.: Dependent type systems as macros. Proceedings of the ACM on Programming Languages **4**(POPL), 1–29 (2019)
4. Clinger, W., Rees, J.: Macros that work. In: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 155–162 (1991)
5. Delahaye, D.: A tactic language for the system Coq. In: Logic for Programming and Automated Reasoning, 7th International Conference, LPAR 2000, Proceedings. pp. 85–95 (2000)
6. Dybvig, R.K., Friedman, D.P., Haynes, C.T.: Expansion-passing style: Beyond conventional macros. In: Proceedings of the 1986 ACM conference on LISP and functional programming. pp. 143–150 (1986)
7. Dybvig, R.K., Hieb, R., Bruggeman, C.: Syntactic abstraction in Scheme. Lisp and symbolic computation **5**(4), 295–326 (1993)
8. Eastlund, C., Felleisen, M.: Hygienic macros for ACL2. In: International Symposium on Trends in Functional Programming. pp. 84–101. Springer (2010)
9. Ebner, G., Ullrich, S., Roesch, J., Avigad, J., de Moura, L.: A metaprogramming framework for formal verification. Proc. ACM Program. Lang. **1**(ICFP) (Sep 2017). https://doi.org/http://dx.doi.org/10.1145/3110278
10. Flatt, M.: Binding as sets of scopes. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 705–717. POPL '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2837614.2837620, `http://doi.acm.org/10.1145/2837614.2837620`
11. Kohlbecker, E., Friedman, D.P., Felleisen, M., Duba, B.: Hygienic macro expansion. In: Proceedings of the 1986 ACM conference on LISP and functional programming. pp. 151–161 (1986)
12. Mahboubi, A., Tassi, E.: Mathematical components, `https://math-comp.github.io/mcb/`
13. Marlow, S., et al.: Haskell 2010 language report (2010), `https://www.haskell.org/onlinereport/haskell2010`
14. Moggi, E.: Notions of computation and monads. Information and computation **93**(1), 55–92 (1991)
15. de Moura, L., Avigad, J., Kong, S., Roux, C.: Elaboration in dependent type theory (2015)
16. Rafkind, J., Flatt, M.: Honu: syntactic extension for algebraic notation through enforestation. In: ACM SIGPLAN Notices. vol. 48, pp. 122–131. ACM (2012)
17. Sheard, T., Jones, S.P.: Template meta-programming for Haskell. In: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell. pp. 1–16. ACM (2002)

18. Taha, W., Sheard, T.: MetaML and multi-stage programming with explicit annotations. Theoretical computer science **248**(1-2), 211–242 (2000)

Appendix A Bigop example

```
-- preliminaries
def Seq (α : Type) := List α

def BigBody (β α) :=  α × (β → β → β) × Bool × β

def applyBig {α β : Type} (body : BigBody β α) (x : β) : β :=
let (_, op, b, v) := body;
if b then op v x else x

def reducebig {α β : Type} (idx : β) (r : Seq α) (body : α → BigBody
    β α) : β :=
r.foldr (applyBig ∘ body) idx

def bigop := @reducebig

def iota : Nat → Nat → List Nat
| m, 0   => []
| m, n+1 => m :: iota (m+1) n

def index_iota (m n : Nat) := iota m (n - m)

class Enumerable (α : Type) :=
(elems : List α)

instance : Enumerable Bool :=
{ elems := [false, true] }

instance {α β} [Enumerable α] [Enumerable β]: Enumerable (α × β) :=
{ elems := do a ← Enumerable.elems α; b ← Enumerable.elems β; pure
    (a, b) }

def finElemsAux (n : Nat) : forall (i : Nat), i < n → List (Fin n)
| 0,   h => [⟨0, h⟩]
| i+1, h => ⟨i+1, h⟩ :: finElemsAux i (Nat.ltOfSuccLt h)

def finElems : forall (n : Nat), List (Fin n)
| 0     => []
| (n+1) => finElemsAux (n+1) n (Nat.ltSuccSelf n)

instance {n} : Enumerable (Fin n) :=
{ elems := (finElems n).reverse }

instance {n} : HasOfNat (Fin (Nat.succ n)) :=
⟨Fin.ofNat⟩

-- The new, macro-based frontend is not the default yet, so we
    activate it explicitly for the remaining file
```

```
new_frontend

-- Declare a new syntax category for "indexing" big operators
-- declare_syntax_cat
declare_syntax_cat index
syntax ident "<-" term : index
syntax ident "<-" term "|" term : index
-- end
syntax term:50 "≤" ident "<" term : index
syntax term:50 "≤" ident "<" term "|" term : index
-- Primitive notation for big operators
syntax "\\big_" "[" term "," term "]" "(" index ")" term : term

-- We define how to expand `\big` with the different kinds of index
macro_rules
| `(\big_ [$op, $idx] ($i <- $r) $F) => `(bigop $idx $r (fun $i:ident
    => ($i:ident, $op, true, $F)))
| `(\big_ [$op, $idx] ($i <- $r | $p) $F) => `(bigop $idx $r (fun
    $i:ident => ($i:ident, $op, $p, $F)))
| `(\big_ [$op, $idx] ($lower ≤ $i < $upper) $F) => `(bigop $idx
    (index_iota $lower $upper) (fun $i:ident => ($i:ident, $op, true,
    $F)))
| `(\big_ [$op, $idx] ($lower ≤ $i < $upper | $p) $F) => `(bigop $idx
    (index_iota $lower $upper) (fun $i:ident => ($i:ident, $op, $p,
    $F)))

-- Sum
macro "Σ" "(" idx:index ")" F:term : term =>
`(\big_ [HasAdd.add, 0] ($idx:index) $F)
-- end

-- We can already use `Σ` with the different kinds of index.
#check Σ (i <- [0, 2, 4] | i != 2) i
#check Σ (10 ≤ i < 20 | i != 5) i+1
#check Σ (10 ≤ i < 20) i+1

-- Define `Π`
macro "Π" "(" idx:index ")" F:term : term => `(\big_ [HasMul.mul, 1]
    ($idx:index) $F)

-- The examples above now also work for `Π`
#check Π (i <- [0, 2, 4] | i != 2) i
#check Π (10 ≤ i < 20 | i != 5) i+1
#check Π (10 ≤ i < 20) i+1

-- We can extend our grammar for the syntax category `index`.
syntax ident "|" term : index
syntax ident ":" term : index
syntax ident ":" term "|" term : index
-- Add new rules
```

```
macro_rules
| `(\big_ [$op, $idx] ($i:ident : $type) $F) => `(bigop $idx
    (Enumerable.elems $type) (fun $i:ident => ($i:ident, $op, true,
    $F)))
| `(\big_ [$op, $idx] ($i:ident : $type | $p) $F) => `(bigop $idx
    (Enumerable.elems $type) (fun $i:ident => ($i:ident, $op, $p, $F)))
| `(\big_ [$op, $idx] ($i:ident | $p) $F) => `(bigop $idx
    (Enumerable.elems _) (fun $i:ident => ($i:ident, $op, $p, $F)))

-- The new syntax is immediately available for all big operators that
    we have defined
def myPred (x : Fin 10) : Bool := true
#check Σ (i : Fin 10) i+1
#check Σ (i : Fin 10 | i != 2) i+1
#check Σ (i | myPred i) i+i
#check Π (i : Fin 10) i+1
#check Π (i : Fin 10 | i != 2) i+1
#check Π (i | myPred i) i+i

-- We can easily create alternative syntax for any big operator.
macro "Σ" idx:index "," F:term : term => `(Σ ($idx:index) $F)

#check Σ 10 ≤ i < 20, i+1

-- Finally, we create a command for automating the generation of big
    operators.
syntax "def_bigop" str term:max term:max : command
-- Antiquotations can be nested as in `$$F`, which expands to `$F`,
    i.e. the value of
-- `F` is inserted only in the second expansion, the expansion of the
    new macro `$head`.
macro_rules
| `(def_bigop $head $op $unit) =>
   `(macro $head:strLit "(" idx:index ")" F:term : term => `(\big_
    [$op, $unit] ($$idx:index) $$F))

def_bigop "SUM" Nat.add 0
#check SUM (i <- [0, 1, 2]) i+1
```

Appendix B Example expander

```
import Init.Lean
namespace Lean
namespace Expander

-- Result of name resolution. As in the paper, we will ignore the
    second component here.
abbrev NameRes := Name × List String
-- We model the global context more precisely as a mapping from
    symbols to qualified symbols,
-- e.g. (`a ↦ [`ns1.a, `ns2.a])
abbrev GlobalContext := Name → List NameRes

-- the simplified transformer monad
structure TransformerM.Context :=
(gctx : GlobalContext)
(currMacroScope : MacroScope)

abbrev TransformerM := ReaderT TransformerM.Context Id
abbrev Transformer := Syntax → TransformerM Syntax

-- support syntax quotations in transformers
instance : MonadQuotation TransformerM := {
  getCurrMacroScope   := do ctx ← read; pure ctx.currMacroScope,
  withFreshMacroScope := fun _ x => x,  -- dummy impl, unused
  -- The actual implementation also adds the current module name to
    macro scopes for global uniqueness,
  -- which we can ignore in this single-file example.
  getMainModule       := pure `Expander
}

-- the expander extension of the transformer monad
structure ExpanderM.Context extends TransformerM.Context :=
(lctx : NameSet)
(macros : Name → Option Transformer)

abbrev ExpanderM := ReaderT ExpanderM.Context $ StateT MacroScope $
    ExceptT String $ Id

instance MonadQuotation : MonadQuotation ExpanderM := {
  getCurrMacroScope   := do ctx ← read; pure ctx.currMacroScope,
  withFreshMacroScope := fun α x => do
    fresh ← modifyGet (fun n => (n, n + 1));
    adaptReader (fun (ctx : ExpanderM.Context) => {ctx with
    currMacroScope := fresh}) x,
  getMainModule       := pure `Expander
}
```

```
-- implicitly coerce transformer monad into expander monad
instance {α : Type} : HasCoe (TransformerM α) (ExpanderM α) := {
  coe := fun t ctx => pure $ t {..ctx}
}

-- simplified: ignore the module name parameter
def addMacroScope (n : Name) (scp : MacroScope) : Name :=
Lean.addMacroScope `Expander n scp

def getGlobalContext : TransformerM GlobalContext := do
ctx ← read;
pure ctx.gctx

def getLocalContext : ExpanderM NameSet := do
ctx ← read;
pure ctx.lctx

def resolve (gctx : GlobalContext) (n : Name) : List NameRes :=
gctx n

-- slightly more meaningful name
def getIdentVal : Syntax → Name := Syntax.getId

def getPreresolved : Syntax → List NameRes
| Syntax.ident _ _ _ preresolved => preresolved
| _                              => []

def mkOverloadedIds (cs : List NameRes) : Syntax :=
Syntax.node choiceKind (cs.toArray.map (mkTermId ∘ Prod.fst))

def withLocal (l : Name) : ExpanderM Syntax → ExpanderM Syntax :=
adaptReader (fun (ctx : ExpanderM.Context) => {ctx with lctx :=
    ctx.lctx.insert l})

def getTransformerFor (k : SyntaxNodeKind) : ExpanderM (Syntax →
    ExpanderM Syntax) := do
ctx ← read;
match ctx.macros k with
| some t => pure (fun stx => t stx)
| none   => throw ("unknown macro " ++ toString k)

-- expand
partial def expand : Syntax → ExpanderM Syntax
| stx ⇒ match_syntax stx with
  | `($id:ident) => do
    let val := getIdentVal id;
    gctx ← getGlobalContext;
    lctx ← getLocalContext;
    if lctx.contains val then
      pure (mkTermId val)
```

```
    else match resolve gctx val ++ getPreresolved id with
      | []        ⇒ throw ("unknown identifier " ++ toString val)
      | [(id, _)] ⇒ pure (mkTermId id)
      | ids       ⇒ pure (mkOverloadedIds ids)
  | `(fun ($id:ident : $ty) ⇒ $e) ⇒ do
    let val := getIdentVal id;
    ty ← expand ty;
    e ← withLocal val (expand e);
    `(fun ($(mkTermId val) : $ty) ⇒ $e)
-- end
  -- more core forms
  | `(fun $id:ident ⇒ $e) ⇒ do
    e ← withLocal (getIdentVal id) (expand e);
    `(fun $id:ident ⇒ $e)
  | `($num:num) => `($num:num)
  | `($str:str) => `($str:str)
  | `($n:quotedName) => `($n:quotedName)
  | `($fn $args*) => do
    fn ← expand fn;
    args ← args.mapM expand;
    `($fn $args*)
  | `(def $id := $e) => do
    e ← expand e;
    `(def $id := $e)
  -- syntax: keep as-is
  | `(syntax [$cat] $args* : $kind) => `(syntax [$cat] $args* : $kind)
  -- macro_rules: expand rhs (but not lhs) to exercise syntax
    quotation macro
  | `(macro_rules | $lhs => $rhs) => do
    let vars := Lean.Elab.Term.Quotation.getPatternVars lhs;
    rhs ← vars.foldr (fun var ex => withLocal (var.getIdAt 0) ex)
    (expand rhs);
    `(macro_rules | $lhs => $rhs)
  | _ => do
    -- expansion consists of multiple commands => yield and get called
    back per command
    if stx.isOfKind nullKind then pure stx else do
    t ← getTransformerFor stx.getKind;
    stx ← withFreshMacroScope (t stx);
    expand stx

open Lean.Elab.Term.Quotation
-- quoteSyntax
partial def quoteSyntax : Syntax → TransformerM Syntax
| Syntax.ident info rawVal val preresolved ⇒ do
  gctx ← getGlobalContext;
  let preresolved := resolve gctx val ++ preresolved;
  `(Syntax.ident none $(quote rawVal) (addMacroScope $(quote val) msc)
    $(quote preresolved))
| stx@(Syntax.node k args) ⇒
```

```
  if isAntiquot stx then pure (getAntiquotTerm stx)
  else do
    args ← args.mapM quoteSyntax;
    `(Syntax.node $(quote k) $(quote args))
| Syntax.atom info val ⇒ `(Syntax.atom none $(quote val))
| Syntax.missing ⇒ pure Syntax.missing

def expandStxQuot (stx : Syntax) : TransformerM Syntax := do
stx ← quoteSyntax (stx.getArg 1);
`(do msc ← getCurrMacroScope; pure $stx)
-- end

-- two more, simple macros
def expandDo (stx : Syntax) : TransformerM Syntax :=
match_syntax stx with
| `(do $id:ident ← $val; $body:term) => `(HasBind.bind $val (fun
    $id:ident => $body))
| _                                  => pure Syntax.missing

def expandParen (stx : Syntax) : TransformerM Syntax :=
match_syntax stx with
| `(($e)) => pure e
| _       => pure Syntax.missing

-- custom Syntax pretty printer for our core forms that uses the
    paper's notation for hygienic identifiers
open Lean.Format
def ppIdent (n : Name) : Format :=
let v := extractMacroScopes n;
fmt $ v.scopes.foldl mkNameNum v.name

-- flip to make output more readable
def hideMacroRulesRhs := false

partial def pp : Syntax → Format
| stx => match_syntax stx with
  | `($id:ident) => match getPreresolved id with
    | [] => ppIdent id.getId
    | ps => ppIdent id.getId ++ bracket "{" (joinSep (ps.map (fmt ∘
    Prod.fst)) ", ") "}"
  | `(fun ($id : $ty) => $e) => paren $ "fun " ++ paren (pp id ++ " :
    " ++ pp ty) ++ " ⇒ " ++ pp e
  | `(fun $id => $e) => paren $ "fun " ++ pp id ++ " => " ++ pp e
  | `($num:numLit) => fmt (num.isNatLit?.getD 0)
  | `($str:strLit) => repr (str.isStrLit?.getD "")
  | `($fn $args*) => paren $ pp fn ++ " " ++ joinSep (args.toList.map
    pp) line
  | `(def $id:ident := $e) => "def " ++ ppIdent id.getId ++ " := " ++
    pp e
```

```
  | `(syntax [$cat] $args* : $kind) => "syntax ..."  -- irrelevant for
    this example
  | `(macro_rules | $lhs => $rhs) => "macro_rules |" ++
    lhs.reprint.getD "" ++ " => " ++ (if hideMacroRulesRhs then "..."
    else pp rhs)
  | _ => "<not a core form: " ++ toString stx ++ ">"

-- integrate example expander into frontend, between parser and
    elaborator. Not pretty.
section Elaboration
open Lean.Elab
open Lean.Elab.Frontend

def runCommandElabM' (x : Command.CommandElabM Syntax) : FrontendM
    Syntax :=
fun ctx => do
  cmdPos ← liftIOCore! $ ctx.cmdPosRef.get;
  let cmdCtx : Command.Context := { cmdPos := cmdPos, stateRef :=
    ctx.commandStateRef, fileName := ctx.inputCtx.fileName, fileMap :=
    ctx.inputCtx.fileMap };
  EIO.catchExceptions (x cmdCtx) (fun _ => pure Syntax.missing)

-- run expander: adapt global context and set of macro from Environment
def expanderToFrontend (ref : Syntax) (e : ExpanderM Syntax) :
    FrontendM Syntax := runCommandElabM' $ do
st ← get;
match e {
  gctx := fun n => (match st.env.find? n with
    | some _ => [(n, [])]
    | none   => [] : List NameRes),
  lctx := {},
  currMacroScope := st.nextMacroScope,
  macros := fun k =>
    -- our hardcoded example macros
    if k == `Lean.Parser.Term.stxQuot then some expandStxQuot
    else if k == `Lean.Parser.Term.do then some expandDo
    else if k == `Lean.Parser.Term.paren then some expandParen
    -- `notation`, `macro`, and macros generated at runtime
    else
      let table := (macroAttribute.ext.getState st.env).table;
      match table.find? k with
      | some (t::_) => some (fun stx ctx => match t stx {mainModule :=
    `Expander, ..ctx} 0 with EStateM.Result.ok stx s => stx | _ =>
    Syntax.missing)
      | _           => none
} (st.nextMacroScope + 1) with
| Except.ok (stx, nextMacroScope) => do
  modify (fun st => {st with nextMacroScope := nextMacroScope});
  pure stx
| Except.error e => do
```

```
    logError ref e;
    pure Syntax.missing

  partial def processCommand : Syntax → FrontendM Unit
  | cmd => do
    cmd' ← expanderToFrontend cmd (expand cmd);
    if cmd'.isOfKind nullKind then
      -- expander returned multiple commands => process in turn
      cmd'.getArgs.forM processCommand
    else do
      runCommandElabM $ logInfo cmd (pp cmd');
      elabCommandAtFrontend cmd'

  partial def processCommands : Unit → FrontendM Unit
  | () => do
    cmdState    ← getCommandState;
    parserState ← getParserState;
    inputCtx    ← getInputContext;
    (cmd, ps, messages) ← pure (Parser.parseCommand cmdState.env
      inputCtx parserState cmdState.messages);
    setParserState ps;
    setMessages messages;
    if Parser.isEOI cmd then do
      pure ()
    else do
      processCommand cmd;
      processCommands ()

  def process (input : String) (env : Environment) (opts : Options)
      (fileName : Option String := none) : IO (Environment ×
      MessageLog) := do
  let fileName   := fileName.getD "<input>";
  let inputCtx   := Parser.mkInputContext input fileName;
  parserStateRef ← IO.mkRef { Parser.ModuleParserState . };
  cmdStateRef    ← IO.mkRef $ Command.mkState env {} opts;
  ps ← parserStateRef.get;
  cmdPosRef ← IO.mkRef ps.pos;
  EIO.adaptExcept (fun (ex : Empty) => Empty.rec _ ex) $
    processCommands () { commandStateRef := cmdStateRef, parserStateRef
      := parserStateRef, cmdPosRef := cmdPosRef, inputCtx := inputCtx };
  cmdState ← cmdStateRef.get;
  pure (cmdState.env, cmdState.messages)

  def run (input : String) : MetaIO Unit := do
  env  ← MetaIO.getEnv;
  opts ← MetaIO.getOptions;
  (env, messages) ← liftM $ process input env opts;
  messages.forM $ fun msg => IO.println msg;
  pure ()
```

```
end Elaboration

-- examples
-- see also `hideMacroRulesRhs` above

#eval run "
def x := 1
def e := fun (y : Nat) ⇒ x
notation \"const\" e ⇒ fun (x : Nat) ⇒ e
def y := const x
"

#eval run "
macro \"m\" n:ident : command => `(
  def f := 1
  macro \"mm\" : command => `(def $n:ident := f    def f := $n:ident))
m f
mm
mm
"

end Expander
end Lean
```