Teaching Your Rooster to Crow in C

Jason Gross

Coq Workshop 2018

Abstract

Coq's notation system is both extremely powerful and confusingly ad-hoc. While powerful enough to pretty-print abstract syntax trees in most domain-specific languages, how to do so does not seem to be common knowledge. Typical questions arising from such an endeavor might include "How do I pick notation levels?", "Why are these notations clashing?", "Which things should be marked as symbols?", "How do I use boxes in `format`?", and "How do I get parentheses to show up (only) where I want them to?" This interactive presentation aims to serve as a guide to these questions and more, by demonstrating and explaining how to pretty-print subsets of C using only Coq's `Notation` mechanism.

My goal in this presentation is to give the audience enough familiarity with `Notation`s to use them in their own projects for pretty-printing DSLs, or at least to know what sorts of experiments to execute to discover what they want. As such, I intend this to be a very interactive presentation. I plan to explain the basics of using `Notation`s, a selection of features, and present a number of tricks for getting things to work. The audience will be encouraged to ask questions and propose things they'd like to see me do with notations, and the presentation will take place in a prepared ProofGeneral buffer.

I expect to cover the following things, in roughly the following order, as time permits:

- Syntax of notations (e.g., `Infix "+" := add.`; `Notation "a + b" := (add a b).`)
  - Notation bodies must be parenthesized
- How to use levels and associativity, how to pick levels, how to discover levels of existing notations (`Print Grammar constr`)
- Controlling display: `format`, extra spaces, quoting symbols, boxes in `format` for display of indentation, newlines, and whitespace
- Recursive notations: `..` for pair- and λ-like things
- `Reserved Notation` and `only printing`, useful for ensuring that notations don't conflict

- Using single-variable only-printing notations for hiding identifiers
- Abbreviations vs notations (what it means when your `Notation` doesn't have quotes, and why you might want this)
- Notation scopes, useful for overloading notations and selecting the right one automatically (`Bind Scope`, `Delimit Scope`, `Open Scope`)
- Using binders for destructuring pairs
- Parenthesizing notations: when to use parentheses, and when to use levels
- Deliberate ordering of notations: how to express wildcard and negation in notation matching

Examples By the end of the presentation, I intend for the audience to understand, or at least feel comfortable experimenting with, notations that look like any of the following:

```
Infix "+" := Add.
Notation "T x = A ; b" := (LetIn (tx:=T) A (fun x => b))
  (at level 200, b at level 200, format "T  x  =  A ; '//' b").
Notation "T x = A ; 'return' ( b0 , b1 , .. , b2 )"
   := (LetIn (tx:=T) A (fun x => Pair .. (Pair b0%expr b1%expr) .. b2%expr))
  (at level 200, format "T  x  =  A ; '//' 'return'  ( b0 ,  b1 ,  .. ,  b2 )").
Notation "x" := (Var x) (only printing).
Notation "'slet' x .. y := A 'in' b"
  := (LetIn A%expr (fun x => .. (fun y => b%expr) .. )) : expr_scope.
Notation "( x * y )" := (Op (Mul _ _ _) (Pair x y))
  (format "( x  *  y )") : expr_scope.
Notation "( x * '(uint8_t)' y )"
 := (Op (Mul (TWord _) (TWord (S _)) (TWord 1)) (Pair x (Var y)))
  (at level 40, y at level 9, format "( x  *  '(uint8_t)' y )") : expr_scope.
```

Motivating Example A sort of "motivating example" driving this presentation, an answer to the question of "what can we do with this?", is turning a syntax tree that looks like

```
Abs (fun '(x8, x9, x7, x5, x3, (x16, x17, x15, x13, x11)) =>
LetIn (Op (Mul (TWord 6) (TWord 6) (TWord 7)) (Pair (Var x3) (Var x11)))
(fun x18 : var (TWord 7) => ...
```

into

```
 λ '(x8, x9, x7, x5, x3, (x16, x17, x15, x13, x11))%core,
 uint128_t x18 = ((uint128_t)x3 * x11);
 ...
```