An experience report on writing usable DSLs in Coq

Clément Pit-Claudel
MIT CSAIL
Cambridge, Massachusetts, USA
cpitcla@csail.mit.edu

Thomas Bourgeat
MIT CSAIL
Cambridge, Massachusetts, USA
bthom@csail.mit.edu

Abstract

Features added to Coq over the last 5 years have made it possible to create drastically more usable domain-specific languages (DSLs). We report on our experience building and using a hardware-description language embedded within Coq, highlighting how recent Coq improvements make it possible to solve longstanding pain points with Coq DSLs.

1 Introduction

The following snippets are valid Coq programs representing two deeply embedded Kôika [2] terms:

```
Definition mul₃ {sz: nat} : IntFun R Sigma := <{
  fun mul₃ (bs: bits_t sz) : bits_t sz =>
    (bs << 1'b0) + bs }>.
```

```
Program Definition multiply : rule R Sigma := <{
  let v := read₁(r₀) in
  let odd := v[0] in
  if odd then write₁(r₀, mul₃(v) + 1)
  else fail }>.
```

A common way to design and study programming languages in Coq [8] is to embed them by leveraging Coq's advanced notation system. In the past, this approach didn't scale well: limitations of Coq's notations made the user experience too rough to allow writing useful programs instead of small examples.

By leveraging Coq features introduced in the last few years and a few recently discovered tricks, we found that we could design a deeply embedded language offering an almost-decent user experience. Specifically, we report on the design of a Coq frontend for Kôika [2], a rule-based hardware description language embedded in Coq. We have written dozens of programs covering thousands of lines in this DSL, including a simple pipelined RISCV core. We think that we have reached a point where writing Kôika programs is possible with little to no knowledge of Coq.

This abstract walks the reader through the main techniques that we used to create Kôika's two DSLs (Kôika programs can be written in a simple untyped language and typechecked, or written directly in an intrinsically typed DSL [6] that guarantees well-formedness and well-typedness using dependent types; the syntax is the same).

2 Experimenting with Custom Entries

Custom Entries [7], a Coq feature introduced in Coq 8.9, allow users to use multiple independent grammars within one Coq file. In Kôika files, when Coq encounters `{{...}}` (for untyped Kôika programs) or `<{...}>` (for dependently-typed ones), it forgets about everything about Coq's grammar and starts parsing Kôika's grammar.

Custom entries are very powerful: with them, it is possible to parse a broad range of grammars that were difficult or impossible to parse neatly with Coq's original `Notation` system. For example, custom entries make it trivial to define complex recursive notations, without using Coq's limited built-in support: one can define a notation for associative maps like `{# a -> 1; b -> 2 #}` by defining a custom entry delimited by `{# ... #}`, and a notation "`_ -> _; _`" exclusive to that custom entry.

In fact, custom entries are so powerful that the practical limitations that we encountered when using them popped up not in notations but in editing tools. Indeed, introducing new syntax can cause issues with indentation, syntax highlighting, and even sentence parsing.

Instead, we found it best to match Coq's existing syntax as much as possible when designing Kôika (this is the opposite of the usual advice for standard notations: normally, one tries to make up notations that do not conflict with Coq's built-in ones). For example, the following construct adds a parsing rule in the `koika` custom entry to handle what looks like a standard Coq "let" construct, but is in fact parsed as a Kôika AST when encountered within Kôika delimiters.

```
Notation "'let' a ':=' b 'in' c" :=
  (UBind a b c) (in custom koika  ...).
```

The same can be done for conditionals, matches, anonymous functions, and record construction: this results in a pleasant editing experience.

Custom entry delimiters act as a form of quoting. A pattern that we found practically useful was to provide an antiquoting mechanism: a way to exit the Kôika parser to reenter the Coq one (`` `...` `` in Kôika). This enables us to macro-generate subterms of Kôika ASTs using Gallina.

3 Deep-embedded binders

A key part of designing a DSL is to choose a binder representation strategy [1, 3–5]. Shallow and mixed embeddings typically use native Coq binders, while deep embeddings commonly use strings for variables. Unfortunately, this choice leads to unpleasant syntax, along the lines of the following example:

```
{{ let "x" := 1 in let "y" := "x" + 1 in "x" + "y" }}.
```

By combining notations, custom entries, tactics in terms, and Ltac2's reflection capabilities, we can do better. First, we create a notation `ident_to_string a` which, given an unbound identifier, returns the corresponding string. Then, we use a custom entry so that the default interpretation of a plain symbol in the `koika` scope resolves to that notation:

```
Check (ident_to_string CoqPL). => "CoqPL"
Notation "a" := (UVar (ident_to_string a)) ...
```

The result is a decluttered and readable syntax, which resolves to a deeply embedded AST with variable names `"x"` and `"y"`.

```
{{ let x := 1 in let y := x + 1 in x + y }}.
```

The key challenge is the implementation of the function `ident_to_string`. We plan to release it as a standalone library before the CoqPL meeting, so users don't need to know the following — admittedly unpalatable — details. First, we need a way to convert a bound identifier to a Gallina string: `ident_to_constr : ident -> constr`. Ltac2 already supplies `Ident.to_string: ident -> string`, so it's simply a matter of converting an Ltac2 string to a Gallina string using Ltac2's limited APIs. We have found that using a lookup table to translate Ltac2 `char`s to Gallina `ascii`s was the most efficient. Next, we need a way to capture an unbound identifier from Ltac2, before Coq realizes that the identifier is unbound. This is done using notations and tactics in terms, as shown below.

```
Inductive __Ltac2_IdentMarker := __Ltac2_Mark.

Ltac serialize_ident_in_context :=
  ltac2:(match! goal with
  | [ h: __Ltac2_IdentMarker |- _ ] =>
    let s := ident_to_constr h in exact ($s)
  end).

Notation ident_to_string a :=
  (match __Ltac2_Mark return string with
   | a => ltac:(serialize_ident_in_context)
   end) (only parsing).
```

The `match` in `ident_to_string` uses the name passed to the notation as a binder in a match. Because the match has a single case, it reduces automatically, leaving no trace in the parsed term (in contrast, using a let binding or a lambda abstraction would pollute the resulting term). Then, `serialize_ident_in_context` captures this binding and converts it to a string using Ltac2.

4 Improved error messages

Programs written in Kôika's untyped AST have to be typechecked, using a function that originally returned either a typed AST, or an error message. Unfortunately, error messages without locations are near-useless in large terms: our users spend their time looking at error messages like `4'b0 has type bits 4 but should have type bits 5`.

We solved this problem by annotating AST nodes with positions. A frontend build on a standard parser could instantiate these locations with a file name and lines and column numbers, but within Coq we do not have access to that location information.

Instead, our notations generate an unannotated AST, which we then annotate by augmenting each node with a representation of the path leading to it. When the typechecker detects an error, it reports that path; we use it to traverse the original (unannotated) AST and wrap the corresponding node into another type of annotation, indicating the location of the error. Concretely:

```
Check (tc {{ let x := y in y + 1 }}).

Error: Tactic failure:
## In term: {{ "y" }}
## Type error: (UnboundVariable "y")
## Context: {{ let "x" := >>> "y" <<< in "y" + 1 }}
```

5 Type classes for DeBruijn index inference

Kôika has two sets of ASTs, (typed and untyped), but we use the same notations for both (in different custom entries). One difficulty arises for variable references in typed term. Consider the following snippet:

```
let x := 4'b0 in y := x + 1
```

The corresponding AST is this one, where `?m: member "x" ["x"]` is a dependently typed DeBruijn index witnessing that "x" belongs to the list of variables in scope (here, just `["x"]`):

```
Bind "x" (Const 0) (Plus (Var "x" ?m) (Const 1))
```

We use a type class trick to infer `?m`. First, we define a type class whose only field is a proof that a variable "k" belongs to a given context "sig":

```
Class VarRef k sig := vr_m : member k sig.
Hint Mode VarRef + + : typeclass_instances.
Hint Extern 1 (VarRef ?k ?sig) =>
  exact (must (assoc k sig)) : typeclass_instances.
```

Then, we set the notation for variables to produce `(Var "x" (_: VarRef "x" _))`. Coq's unification fills in the hole for `sig`, then type class inference kicks in to resolve `VarRef "x" ["x"]`. An appropriate instance is inferred by the tactic called through `Hint Extern`.

(Note that tactics in terms would not work here, because the tactic would be called before inferring the local context `sig`, and we would not be able to infer the dependent index `m` without that information).

Conclusion

Coq has come a long way since the dark days of Coq 8.4. Such a long way, in fact, that writing moderately-large programs in deep embedded DSLs within Coq can be made reasonably pleasant. Our talk will walk the audience through these tricks and explain the implementation interactively.

References

[1] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. 2005. Mechanized Metatheory for the Masses: The PoplMark Challenge. In *Theorem Proving in Higher Order Logics*, Joe Hurd and Tom Melham (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 50–65.

[2] Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. 2020. The Essence of Bluespec: A Core Language for Rule-Based Hardware Design. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 243–257. https://doi.org/10.1145/3385412.3385965

[3] Arthur Charguéraud. 2012. The Locally Nameless Representation. *J. Autom. Reason.* 49, 3 (2012), 363–408. https://doi.org/10.1007/s10817-011-9225-2

[4] Adam Chlipala. 2008. Parametric higher-order abstract syntax for mechanized semantics. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, James Hook and Peter Thiemann (Eds.). ACM, 143–156. https://doi.org/10.1145/1411204.1411226

[5] N. G de Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)* 75, 5 (1972), 381–392. https://doi.org/10.1016/1385-7258(72)90034-0

[6] Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. 2018. Intrinsically-typed definitional interpreters for imperative languages. *Proceedings of the ACM on Programming Languages* 2, POPL (2018), 16:1–16:34. https://doi.org/10.1145/3158104

[7] The Coq Development Team. 2019. *The Coq Proof Assistant, version 8.9.0.* https://doi.org/10.5281/zenodo.2554024

[8] The Coq Development Team. 2020. *The Coq Proof Assistant, version 8.11.0.* https://doi.org/10.5281/zenodo.3744225