Lecture 19: Mechanizing Metatheory I

Chris Martens and Sam Stites

December 2nd, 2024

1 Introduction

In this lecture, we will begin a two-part development on *mechanizing metatheory*, which means taking much of what we've done in class thus far and writing it in a formal computer language that can be mechanically checked.

In class, we will develop a file called `IPPL.agda` and provide a link to it after class is over, so that you can reference it and try running it yourself if you so desire. If you have Agda already installed on your computer, you are welcome to follow along.

The lesson plan is as follows:

- Motivation/what is mechanizing metatheory?
- Introduction to Agda and dependent types
- Hutton's Razor encoding: defining types, statics, and dynamics
- Proving a simple correctness property

These will be our last two lectures for the course before student presentations, so we will also try to conclude with some general takeaways about our approach to PL in this class.

2 What is Mechanizing Metatheory?

Programming languages and logics, which we will refer to collectively under the term *deductive systems*, are examples of *theories*: the statements we can prove and the programs we can write according to the internal rules of these systems represent all the ways we can combine and transforming hypothetical knowledge within them.

Metatheory, then, is the practice of formalizing these systems and proving theorems *about* them (such as type safety or parametricity), within some usually-implicit meta-logic. *Mechanizing metatheory* is the idea that we want that metalogic to be *computational*: something like a programming language itself, wherein we can benefit from parsing and type checking to ensure well-formedness of our rules as well as *proof checking* and other *proof assistance* for stating and proving our metatheorems. There is a nice paper [Harper and Licata(2007)] offering a deep dive into the theory and practice of mechanizing metatheory (as of 2007, though the landcape of proof assistants has changed somewhat since then).

2.1 From Formal Systems to Formally Verified Systems

In class we've described a number of systems in terms of several components: syntax, static semantics, and dynamic semantics. This allows us to break down challenging problems, like reasoning about mutation, and construct soundness statements, like progress and preservation, which we can prove directly by induction over our inference rules. When you are developing new deductive systems, or modifying known ones, it can sometimes be challenging to develop these rules de novo. You may need to understand a deductive system and metatheoretic proof from someone else in detail without having access to a knowledgeable expert.

This is where *proof assistants* enter the picture: they are computer implementations of something approximating the implicit "meta-logic" in which we work on paper, and they permit us to write proofs and check their correctness, providing a way to *formally verify* correctness of deductive systems.

2.2 Terminology

In this module, we will be referring to "languages" at two levels, so it is helpful to distinguish between them: the **object language** is the deductive system we wish to describe, whereas the **metalanguage** is the language/proof assistant in which we do the describing. For these lectures, the object language will be HR+Bool (Hutton's Razor with booleans), and the metalanguage will be Agda.

Before we can state and prove formal properties about an object language, we need to tell Agda about it via an **encoding**. An encoding is a way of translating from object-level constructs to meta-level constructs.

Without getting into specifics, we can state a guiding principle for developing this encoding. An encoding $\ulcorner - \urcorner$ should *preserve truth* from our object language to our metalanguage. So if $\Gamma \vdash e : \tau$ in our object language, we should have translations $\ulcorner \Gamma \urcorner$, $\ulcorner e \urcorner$, and $\ulcorner \tau \urcorner$ in our metalanguage, and a metalanguage relation $R(-,-,-)$ encoding $\ulcorner \_ \vdash \_ : \_ \urcorner$, such that $R(\ulcorner \Gamma \urcorner, \ulcorner e \urcorner, \ulcorner \tau \urcorner)$. Additionally, whenever we have $R(G, E, T)$ in the metalanguage, this should correspond to some $\Gamma \vdash e : \tau$ in the object language. These two properties form the intuition behind designing an **adequate** encoding.

3 What is Agda?

Agda is a functional programming language that supports sum types, product types, functions, a limited form of recursive types and recursive definitions, and something new for this class called *dependent types*. It will be easier to explain what dependent types are in general once we see a couple of examples.

Here's how Agda represents the natural numbers:

```
data Nat :  Set where
      Zero :  Nat
      Succ :  Nat -> Nat
```

This is approximately the same as the type $\mu\alpha.('\mathsf{zero} : \mathbf{1}) + ('\mathsf{succ} : \alpha)$. The symbols `data` and `where` are keywords in Agda for declaring labeled datatypes like this, its `|` symbol between branches is similar to our $+$, its `Set` symbol is approximately what we would call a type, and the colon (`:`) declares a type for a new identifier being defined.

We can write functions over this data, like a function to compute whether a number is even:

```
even  :  Nat -> Bool
even  Zero = false
even  (Succ n) = not (even n)
```

But a cool "new" thing Agda gives us is the ability to use this same data declaration mechanism that we used to define our datatypes, to write *inference rules in code*, so we can actually give a relational definition of evenness like this:

```
data Even :  Nat -> Set where
      ev/z :  Even Zero

      ev/s :  forall {N : Nat}
              -> Even N
              -> Even (Succ (Succ N))
```

There are two ways to think about this new kind of data, `Even`, we've declared: as a *predicate* (unary relation) over natural numbers, or as a type *indexed by* a natural number. That's what its declaration `Nat -> Set` means: it's a kind of "function" from natural numbers into types (or, equivalently, logical propositions).

Now we probably need to talk about *kinds*.

3.1 Kinds

Just like types classify term-level expressions, *kinds* classify things that exist at the "type level". Previously, our only kind was type: the kind of types. (We used it for making sure types were well-formed in, e.g., System F.) But now we have type *families*, like `Even`, which aren't well-formed things that can classify terms until they're given an argument. That is, `Even` by itself isn't a type, but `Even Zero` is— and so is `Even (Succ Zero)`, although it is not an *inhabited* type.

In actuality, Agda's kinds go beyond this three-level hierarchy (terms–types–kinds), generalizing to an arbitrarily tall hierarchy using something called *universes*. But we won't need to explain that today! We just need to know that there are some type-like things that aren't really types until we apply them to arguments.

3.2 Derivations

Okay, so if `Even (Succ (Succ Zero))` is a type, what are the terms that inhabit it?

The answer is *proofs*, or specifically *derivations*, that the `Even` predicate holds of 2! And now that we've described data constructors that let us build that evidence, we can write it directly as a program:

```
ex0 :  Even 0
ex0 =  ev/z

ex2 :  Even 2
ex2 =  ev/s ev/z

ex4 :  Even 4
ex4 =  ev/s (ev/s ev/z)

ex5 :  Even 5
ex5 =  ?
```

Notice how we can only construct proofs to the `Even` predicate for even numbers. We can still *state* the final example `Even 5`, it is still a predicate, it just so happens to be a false. As a result, we can't construct a proof using the rules provided.

4 Mechanizing Hutton's Razor

For this development, we will follow a recipe which by now should be very familiar.

4.1 Type-Driven Language Design Recipe

1. Define the types of our system
2. Define syntax
3. Define statics (intro/elim rules, canonical forms)
4. Define dynamics (eval rules)
5. State and prove a safety property

Now we will switch to working primarily in Agda instead of including all of the code in the note here. Please see the supplemental Agda file for in-line commentary.

5 Recap: Encoding Deductive Systems in Agda

Here is the approach we have followed so far to designing adequate encodings in Agda:

Object Language	Metalanguage
Syntax	Inductive datatype
Inference rules	Inductively-defined relation
Derivation	Term with relation type
Theorems	Function types
Proofs	Functions whose types are theorems

5.1 Mechanization Effort

Mechanizing proofs is a lot of work! When is it worth it?

Sometimes, the *metatheory* part isn't the most important. Merely coming up with an encoding of your system into a language like Agda can go a long way in terms of making sure your definitions typecheck and pass tests. Also, *stating* a theorem, even without a proof, can be very useful. Describing the property we want out of a formal system is like the "how to design programs" methodology on the other side of types-as-propositions: it helps us figure out our intention and understand when we've succeeded. A mechanized version of that goes even further by giving us extra assurance about those desired properties, but we gain a little bit for every piece that we formalize: it's not all-or-nothing.

Mechanization can also save time in the long term if we plan to extend a deductive system, e.g. add a new construct to a programming language. It means we can add new cases to our proofs wherever they are needed and rely on the proof checker to tell us about any unexpected interactions with other parts of the language. For instance, if we were to prove a normalization theorem, and then add a fixed point operator, the proof checker would generate a new proof case for us that we would not be able to close.

References

[Harper and Licata(2007)] Robert Harper and Daniel R Licata. Mechanizing metatheory in a logical framework. *Journal of functional programming*, 17 (4-5):613–673, 2007.