Metaprogramming: Multi-stage programming and SMT as a backend

— Nada Amin

Lecture 3 - *June 28, 2025*

1 Outline

Idea: Turning "towers of interpreters" into compilers through staging.

2 Meta-theory Compilers

Goal: How can we turn a tower of interpreters into a one-pass compiler using staging? The idea of what to do with one of the levels is based in partial evaluation, namely the Futamara projections.

2.1 Futamura projections

Generally, the three *Futamura projections* formalize the connection between interpreters and compilers through the lens of *specialization*. More specifically, the Futamura projections (first described by Yoshihiko Futamura in 1971) are as follows[1]:

1. 1st projection: specializing an interpreter to a given program yields a compiled version of the program in the language of the interpreter.
2. 2nd projection: a process that can specialize a given interpreter to any program is equivalent to a compiler.
3. 3rd projection: a process that can take any interpreter and turn it into a compiler is a compiler generator (or *cogen*).

[1]Taken from "Collapsing Towers of Interpreters," Amin and Rompf's paper from POPL 2018.

Notes directly from the lecturer[2], which formalizes things a bit.

Let **L** be a meta function from a program to the function it computes. Let **S** and **T** be programming languages. The following equations define `mix`, an **S**-interpreter `int`, and an **S**-to-**T**-compiler `comp`.

Equation for partial evaluator mix:

$$(P)\ L\ p\ [d_1, d_2] = L\ (L\ \text{mix}\ [p, d_1])\ d_2$$

Equation for an S-interpreter int written in L:

$$(I)\ S\ \text{pgm data}\ = L\ \text{int}\ [\text{pgm},\ \text{data}]$$

Equation for an S-to-T-compiler comp written in L:

$$(C)\ S\ \text{pgm}\ \ \text{data}\ = T\ (L\ \text{comp pgm})\ \text{data}$$

The following equations define the Futamura projections, which state that given some partial evaluator `mix` and an interpreter `int`, we can compile programs and even generate standalone compilers and compiler generators by self-applying `mix`:

$$(1)\ L\ \text{mix}\ [\text{int},\ \text{pgm}] = \ \textbf{target}$$

$$(2)\ L\ \text{mix}\ [\text{mix},\ \text{int}] = \ \textbf{compiler}$$

$$(3)\ L\ \text{mix}\ [\text{mix},\ \text{mix}] = \ \textbf{compiler generator}$$

These equations can be verified using the equations for `mix` (P), and for interpreters (I) and compilers (C) above. Specifically, we can do the following verification:

Verify (1):

$$S\ \text{pgm data}\ = L\ \text{int}\ [\text{pgm},\ \text{data}]\ \textit{by}\ (I)$$
$$L\ \text{int}\ [\text{pgm},\ \text{data}] = L(L\ \text{mix}\ [\text{int}, \text{pgm}])\ \text{data}\ \textit{by}\ (P)$$

Therefore, $(L\ \text{mix}\ [\text{int},\ \text{pgm}])$ acts as **target**.

Verify (2):

$$L\ \text{mix}\ [\text{int},\ \text{pgm}] = \ \textbf{target}\ \ \textit{by}\ (1)$$
$$L\ \text{mix}\ [\text{int}, \text{pgm}] = L(L\ \text{mix}\ [\text{mix},\ \text{int}])\ \text{pgm}\ \ \textit{by}\ (P)$$

[2] https://github.com/namin/metaprogramming-lecture-notes/blob/main/3-compilation.tex

Therefore, (L mix [mix, int]) acts as a **compiler**.

Verify (3):

$$L \text{ mix } [\text{mix, int}] = \textbf{ compiler } by\ (2)$$
$$L \text{ mix } [\text{mix, int}] = L(L \text{ mix } [\text{mix, mix}]) \text{ int } by\ (P)$$

Therefore, (L mix [mix, mix]) acts as a **compiler generator**.

The projection is based in the idea of creating a generalized `mix` procedure that takes a source input and an interpreter and yields an output in a target language. In a sense, we are "specializing" the program to some symbolic input—this generates a compiler that will then run on concrete input, without the overhead ofinterpretation.

2.2 Question: "What's binding-time analysis?"

Response: Say you have

```
f(x) = x + x + (2 + 3)
```

and you want to optimize it to

```
f(x) = x^2 + 5
```

to do this, one could annotate the program in such a way that annotates what its type will be in future stages, e.g. in Scala `x : Rep[Int]` tells us that `x` will be of type `Int` in a future stage – figuring out whether or not the type is determined dynamically is called "binding-time analysis."

In general, binding-time analysis *annotates* a program, differentiating between parts that are "eliminable" (and thus computed during partial evaluation) and those that are "residual" (i.e. the remaining parts that are not eliminable). The eliminable parts are then interpreted normally, while the residual parts generate code that ends up in the residual program. See "A Self-Applicable Partial Evaluator for the Lambda Calculus" (Gomard) for further details.

2.3 LMS: Lightweight Modular Staging in Scala

Based on MetaML (and partly on MetaOCaml, an OCaml dialect for multi-stage programming). In practice, most meta theory compilers are implemented in two stages, in order to limit the amount

of reasoning that is required across multiple levels (although it is certainly possible to have more than two stages). At a high level, the idea of staged programming is to allow the programmer to control the *order* (in other words, the timing) of evaluation of terms.

Traditionally, staging is done using the quasiquoting facilities of a language like Scheme or Lisp; in MetaOCaml and Scala's LMS, staging is done instead at the *type level*, i.e. using the distinction between the types `Int` and `Rep[Int]` (as mentioned above).

Using LMS, we can control in which stage our objects are created, e.g. `new Array[Int]` creates one in the first stage but `NewArray[Int]` creates the object in the second stage. We can also define statically known data through `staticData(s)`. More about Scala LMS can be learned through the tutorial[3]. The goal here is to write code that looks like an interpreter but is actually a compiler as it generates code as a side effect.

This pipeline of turning an interpreter into a compiler yields a naive compiler (not necessarily an optimized one), but it is a good starting point when developing a larger system, in terms of removing some of the overhead of abstraction.

2.4 Can this be fully static?

This can be done by encoding lifting and lowering between stages as actual operators in our language.

The `Lift` operator allows us to lift things like closures into lambdas (and similar for other operations). This is typically referred to as normalization through evaluation. Broadly, this can be described as collapsing our tower of interpreters[4] using a multi-stage evaluator. Collapsing towers of interpreters can be achieved through stage polymorphism.[5]

The more dynamic approach relies on a stage-polymorphic VM, where operations are lifted or not by dynamic dispatched, based on the dynamic types of the arguments.

The more static approach relies on stage polymorphism driven by types and optimizations in LMS. Any code, even generated code, can be instantiated for interpretation or compilation.

2.5 Stage Polymorphism

Stage polymorphism (or *binding-time* polymorphism) is essentially the idea that we should be able to write *generic code* that is parameterized over the *timing* of when certain pieces of code are

[3] `https://scala-lms.github.io/tutorials/index.html`

[4] `https://www.cs.purdue.edu/homes/rompf/papers/amin-popl18.pdf`

[5] For a more detailed description of the multi-level core language $\lambda_{\uparrow\downarrow}$ in which all of this is done, see (again) "Collapsing Towers of Interpreters" from Amin and Rompf at POPL 2018.

generated. For example, in $\lambda_{\uparrow\downarrow}$, the multi-level core language that includes the `Lift` operator, we would parameterize over `Lift` itself by replacing `lift` with calls to a parameter `maybe-lift`:

```
(define matches (lambda (maybe-lift) (lambda (r) (lambda (s)
  (if (null? r) (maybe-lift #t) (if (null? s) (maybe-lift #f)
  (if (eq? (lift (car r)) (car s)) ((matches (cdr r)) (cdr s)) (maybe-lift #f))))))))
```

This would then allow us to have two versions of the `matches` function, one generic and one specialized, in parameterized form:

```
define matches-spec (matches (lambda (e) (lift e))))
(define matches-gen (matches (lambda (e) e)))
```

3 SMT as a backend

Follows the idea that you can use the first stage of a staging interpreter to generate code. Can we use multi-stage programming to generate SMT constraints to solve programming puzzles?

Specifically, Holey is a Python framework that allows for the generation of SMTLib directly from a Python expression. The mapping is straightforward between SMTLib and Python functions.

An experimentation with Holey was to see if an LLM can do better than SMT if we give it hints. At the moment, the conclusion is that LLMs can't reason about SMTLib constraints, but they can solve the problem decently through example-driven prompts. The Holey repository contains specific metrics, along with source code.

3.1 Verification with SMT

Typically when using SMT solvers to verify designs, people tend to use the theory of Hoare logic. Basically, the user annotates a program with pre-conditions (defined constraints on inputs), post-conditions (defined expectations about results), and invariants (defined constraints on loop iterations). These are then used to generate *verification conditions* (VCs), which are going to be the actual queries we make to the SMT solver. VCs tend to be of the form `PRE -> (BODY & POST)`.

In order to simplify the transformation, one can perform *Weakest Precondition computations*, which walk inversely and build out the weakest possible precondition (WP) that satisfies the current post-conditions and body and then the SMT query becomes `WP -> PRE`.

Holey shows the general elegance of the staged programming technique: we are able to create a decent SMT query generator with relatively few lines of code.

Summary Thinking in stages can be a very helpful device when reasoning about code. Staging itself can be used to define a very precise notion of code being executed in multiple phases, even though it looks like a monolithic chunk of code. Helps distinguish between things like static and dynamic evaluation.

That said, there are some difficulties with this kind of multi-stage programming, as pointed out during the lecture: *"It's very hard to debug at the high-level when your problem is at the low-level"*. Since multi-stage programming involves moving between different stages (in both directions, upwards and downwards), it can be difficult to pinpoint at what stage a bug is occurring.