## Contents

0 Introduction 5  
  0.1 Metaprogramming 5  
  0.2 Scala, Quickly 6  
  0.3 Further Reading 7  

1 Interpretation 9  
  1.1 Programs as Data / Data as Programs 9  
     1.1.1 Programs as Data 9  
     1.1.2 Data as Programs 9  
  1.2 Interpreters 10  
  1.3 Meta-Circularity 10  
     1.3.1 Meta-Interpreters in Lisp 10  
     1.3.2 Meta-Interpreters in Prolog 10  
  1.4 Further Reading 10  

2 Reflection 11  
  2.1 Reification / Reflection 11  
  2.2 Reflective Towers of Interpreters 11  
  2.3 Metaobject Protocols 11  
  2.4 Reflection vs. Abstraction 11  
  2.5 Further Reading 12  

3 Compilation 13  
  3.1 Partial Evaluation 13  
     3.1.1 Futamura projections 13  
  3.2 Multi-Stage Programming 14  
  3.3 Turning Interpreters into Compilers 14  
  3.4 Collapsing Towers of Interpreters 15  
  3.5 Further Reading 15  

4 Embedding 17  
  4.1 Domain-Specific Languages 17  
  4.2 Finally-Tagless 17  
  4.3 Examples 17
CONTENTS

4.3.1 Non-determinism ........................................ 17
4.3.2 Relational programming ............................... 17
4.3.3 Probabilistic programming ............................ 17
4.4 Further Reading ........................................... 17

5 Synthesis .................................................. 19

5.1 Further Reading ........................................... 19

6 Explore! ................................................... 21

6.1 Unconventional Models of Computation ............... 21
6.2 Relaxing from Symbolic to Neural .................... 21
6.3 Reversible Computing .................................... 21
6.4 Further Reading ........................................... 21
Chapter 0

Introduction

0.1 Metaprogramming

Metaprogramming is writing programs that manipulate programs, as data or processes. For example,

- an interpreter takes a program as input data and then turns this description into a process;
- a compiler takes a program as input data and then produces an equivalent program as output data – translating the description from one language to another.

Metaprogramming is an art, even just to keep all the levels straight. This course gives some recipes for a more principled approach to meta-programming, from art to science.

Why is metaprogramming relevant?

1. Meta-computation is at the heart of computation, theoretically and practically even in the possibility of a universal machine and the realization of a computer.

2. A useful way of programming is starting with a domain, and building a little language to express a problem and a solution in that domain. Metaprogramming gives the tooling.

3. Thinking principally about metaprogramming will give you new approaches to programming. For example, you will have a recipe to mechanically turn an interpreter into a compiler. You will acquire new principles, new shortcuts or lenses or tools for thought and implementation if you will.

Beyond traditional meta-programs such as interpreters and compilers, metaprogramming taken more broadly encompasses programming programming, and manipulating processes that manipulate processes. Reflection comes into play.
Also, control – for example, a high-level logic search strategy controlling a low-level SMT solver is a form of metaprogramming. Ultimately, we want to abstract over abstractions. Metaprogramming gives us a way of thinking about higher-order abstractions and meta-linguistic abstractions.

In this course, we will cover techniques around interpretation, reflection, compilation, embedding, synthesis, and exploration.

0.2 Scala, Quickly

Scala has implicit conversions. They enable promoting one type to another, and also enable the “pimp your library” pattern.

case class Complex(r: Int, i: Int) {
  ...
}

implicit def fromInt(r: Int) = Complex(r, 0)

Scala has implicit parameters. They support type classes as well as automatically providing information based on context.

// type classes, e.g. Numeric[T], Ordering[T]
def foo[T: Numeric](x: T) = {
  val num = implicitly[Numeric[T]]
  num.plus(x, x)
}

// equivalently:
def foo2[T](x: T)(implicit num: Numeric[T]) {
  num.plus(x, x)
}

// for example, we could define Numeric[Complex]
implicit def numComplex: Numeric[Complex] = ...

Scala has case classes. They make it easy to define algebraic data types (ADTs). Here are ADTs for the lambda calculus, and an interpreter from Terms to Values. We can use trait mixin composition to share the Const nodes between Terms and Values. We use sealed classes to get help with exhaustive pattern matching. The @ syntax enables us to precisely name part of the pattern match. Here, it is useful because we’ve been using precise types (not just the ADT union type) in some parts (for example, a closure contains a lambda).

sealed abstract trait Term
sealed abstract trait Val
case class Var(x: String) extends Term
case class Lam(p: Var, b: Term) extends Term
case class App(a: Term, b: Term) extends Term
type Env = Map[Var, Val]
case class Clo(l: Lam, m: Env) extends Val
case class Const(n: Int) extends Term with Val
def ev(e: Term, m: Env): Val = e match {
  case v@Var(x) => m(v)
  case c@Const(n) => c
  case l@Lam(p, b) => Clo(l, m)
  case App(a, b) => ev(a, m) match {
    case Clo(Lam(param, body), clo_env) =>
      val arg = ev(b, m)
      ev(body, clo_env + (param -> arg))
  }
}

A case class provides a few helpers in the companion object. Here is how the desugaring works.

case class Foo(a: Int)
// produces companion object
object Foo {
  def apply(a: Int): Foo = Foo(a)
  def unapply(x: Foo): Some[Int] = Some(x.a)
}

For more, see https://github.com/namin/metaprogramming/blob/master/lectures/0-scala-intro/intro_done.scala

0.3 Further Reading

We will be using the Scala programming language for assignments. There are many books that could be useful but none are required. I recommend Odersky [2014] as well as the book by Odersky et al. [2016].

TAPL [Pierce, 2002], SICP [Abelson and Sussman, 1996], PAIP [Norvig, 1992], Art of Prolog [Sterling and Shapiro, 1994], Building Problem Solvers [Forbus and Kleer, 1993], Test Driven Development by Example [Beck, 2002], Expert F# [Cistermino et al., 2008], Valentino Braitenberg’s Vehicles [Braitenberg, 1986], the Reasoned Schemer [Friedman et al., 2005, 2018] — these are a few of my favorite books.
Chapter 1

Interpretation

1.1 Programs as Data / Data as Programs

1.1.1 Programs as Data

We can encode data in terms of the lambda-calculus. For example, we can define our own notion of pair constructor cons and destructors car and cdr.

\[(\text{car} \ (\text{cons} \ A \ B)) \equiv A\]
\[(\text{cdr} \ (\text{cons} \ A \ B)) \equiv B\]

\[(\text{define} \ \text{my-cons}
\quad (\lambda \ (a \ b)
\quad \quad (\lambda \ (msg)
\quad \quad \quad (\text{if} \ (\text{eq?} \ \text{msg} \ '\text{car}) \ a) \ b)))\]

\[(\text{define} \ \text{my-car}
\quad (\lambda \ (p)
\quad \quad (p \ '\text{car})))\]

\[(\text{define} \ \text{my-cdr}
\quad (\lambda \ (p)
\quad \quad (p \ '\text{cdr})))\]

1.1.2 Data as Programs

Interpreters reflect data as programs, while quoting mechanisms reify programs as data.
1.2 Interpreters

1.3 Meta-Circularity

1.3.1 Meta-Interpreters in Lisp
See https://github.com/namin/metaprogramming/tree/master/lectures/1-lisp

1.3.2 Meta-Interpreters in Prolog
See https://github.com/namin/metaprogramming/tree/master/lectures/1-prolog

1.4 Further Reading

SICP [Abelson and Sussman, 1996], PAIP [Norvig, 1992], Art of Prolog [Sterling and Shapiro, 1994] all have examples of interpreters.

Seminal papers on definitional interpreters and meta-circularity are those of McCarthy [1960] and Reynolds [1972].


Piumarta [2011a] shows how to bootstrap a flexible core interpreter in a simple and elegant serie of stepping stones.

Interpretation can lead to abstract interpretation quite naturally [Codish and Sondergaard, 2002, Darais et al., 2017].

Meta-circularity poses punny questions of trust [Thompson, 1984]. Yet CakeML is a proven-correct compiler that can bootstrap itself [Kumar et al., 2014, Kumar, 2016].
Chapter 2

Reflection

2.1 Reification / Reflection

Reification turns a program structure into data that can be reasoned or acted upon.

Reflection turns data back into a program structure so that it can affect the computation.

2.2 Reflective Towers of Interpreters

Reflective towers of interpreters have a long history, starting with Brian Cantwell Smith’s LISP-3, then brown, blond and black.

Each level (including the user level) has a meta level. So each level can be reasoned about and acted upon.

You can play with the Black reflective tower here: http://io.livecode.ch/learn/readevalprintlove/black

2.3 Metaobject Protocols

Metaobject protocols provide an object-oriented system with a landscape of design options rather than focusing on a single point in the design space.

2.4 Reflection vs. Abstraction

See examples in Java, where reflection breaks basic expectations: https://github.com/namin/metaprogramming/tree/master/lectures/2-reflection-dangers
2.5 Further Reading

Wand [1998] gives meaning to FEXPR and shows that it precludes generally admissible optimizations.


A Meta-Object Protocol (MOP [Kiczales et al., 1993]) opens up the design of the language to the user. Bracha and Ungar [2004] advocates a pluggable form of such reflection. Piumarta and Warth [2008] shows how to design and bootstrap a small flexible object system in steps.

Fun systems of interests because of their malleability and bootstrapping capabilities include Fisher [1970], which opens up all control structures up to defining parsing as a problem in control; Smalltalk, where the MOP in Smalltalk-72 [Goldberg and Kay, 1976] was so flexible that some of its power was revoked later in Smalltalk-80 [Goldberg and Robson, 1983]. Systems with syntax malleability include IMP [Irons, 1970], Lisp-70 [Tester et al., 1973] and Meta II [Schorre, 1964]. Self-hosting systems to study include ichibins [Bacon, 2007] and Maru [Piumarta, 2011b].

Sobel and Friedman [1996] give an early account of monadic reflection. A modern incarnation can be found in Meta-F* [Martínez et al., 2018], where reflection as a principled effect enables sound mixing of proofs by tactics and by automatic discharge to an SMT solver. This approach is related to “proof by reflection”, a common technique in proof assistants such as Coq [Bertot and Castéran, 2004, Chlipala, 2013], Agda [Danielsson, 2013, Stump, 2016, van der Walt and Swierstra, 2012] and HOL [Fallenstein and Kumar, 2015]. Harrison [1995] surveys proof by reflection, and favors the LCF approach even for proving in the large.


Pitrat [1995] (in French) argues for reflection, or change of abstraction level, to solve problems.
Chapter 3

Compilation

3.1 Partial Evaluation

Partial evaluation optimizes a program by specializing to some known arguments. The known arguments are static, while unknown arguments are dynamic. Binding-Time Analysis (BTA) decides whether an expression in the program is static or dynamic by propagation. BTA can be done offline (in advance) or online (while specializing).

3.1.1 Futamura projections

Notation from Jones et al. [1990].
Let \( L \) be a meta function from a program to the function it computes. Let \( S \) and \( T \) be programming languages.

Equation for partial evaluator mix:

\[
(P) \ L \ p \ [d_1, d_2] = L \ (L \ mix \ [p, d_1]) \ d_2
\]

Equation for an \( S \)-interpreter int written in \( L \):

\[
(I) \ S \ pgm \ data = L \ int \ [pgm, \ data]
\]

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

\[
(C) \ S \ pgm \ data = T \ (L \ comp \ pgm) \ data
\]

Futamura projections:

\[
(1) \ L \ mix \ [int, \ pgm] = target
\]

\[
(2) \ L \ mix \ [mix, \ int] = \ compiler
\]

\[
(3) \ L \ mix \ [mix, \ mix] = \ compiler \ generator
\]
The equations are easily verified using the equations for mix \((P)\), and for interpreters \((I)\) and compilers \((C)\) above.

Verify (1):

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

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

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

Verify (2):

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

\[
L \text{ mix [int, pgm] } = L(L \text{ mix [mix, int]}) \text{ pgm by (P)}
\]

Therefore, \((L \text{ mix [mix, int]})\) acts as a compiler.

Verify (3):

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

\[
L \text{ mix [mix, int] } = L(L \text{ mix [mix, mix]}) \text{ int by (P)}
\]

Therefore, \((L \text{ mix [mix, mix]})\) acts as a compiler generator.

### 3.2 Multi-Stage Programming

Multi-stage programming explicitly separates a program, turned into a program generator, into stages — “now” / static / code generator stage vs “later” / dynamic / generated code stage. This distinction can be done syntactically, as in MetaOcaml, or driven by types, as in LMS. In contrast to partial evaluation, the binding times are thus explicit and manual.

See https://scala-lms.github.io/tutorials/index.html for an introduction to LMS.

### 3.3 Turning Interpreters into Compilers

An interpreter can be mechanically turned into a (naive) compiler using staging by making the program static and only the input to the program dynamic.

For example, a staged regular expression matcher makes the regular expression static and the matched string dynamic, generating code specialized to one regular expression. See https://github.com/namin/metaprogramming/tree/master/lectures/3-regexp as a starting point in LMS-verify [Amin and Rompf, 2017b].
3.4 Collapsing Towers of Interpreters

Collapsing towers of interpreters can be achieved through stage polymorphism [Amin and Rompf, 2017a].

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. See http://popl18.namin.net.

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. See https://github.com/namin/lms-black.

3.5 Further Reading

The book by Jones et al. [1993] is the bible of Partial Evaluation. The Futamura projections were first introduced by Futamura [1971, 1999]. The tutorial by ? is a good starting point, building a small partial evaluator in Haskell.

The SQL to C compiler by Rompf and Amin [2015] is good starting point for learning about Lightweight Modular Staging (LMS [Rompf and Odersky, 2012]) and using staging to turn an interpreter into a compiler.

Kiselyov [2018] teaches modular multi-stage programming with MetaOCaml.
Chapter 4

Embedding

4.1 Domain-Specific Languages

4.2 Finally-Tagless

4.3 Examples

4.3.1 Non-determinism

4.3.2 Relational programming

4.3.3 Probabilistic programming

4.4 Further Reading

Examples of embedding logic programming into a functional host include Spivey and Seres [1999], Byrd et al. [2012], Hemann and Friedman [2013]. Embedding a functional interpreter into a relational (purely logical) host results in interesting applications [Byrd et al., 2017].

Probabilities form a monad plus [Ramsey and Pfeffer, 2002, Cisternino et al., 2008]. Hansei [Kiselyov and Shan, 2009] is a language that sports weighted non-determinism.

Chapter 5

Synthesis

5.1 Further Reading

Gulwani et al. [2015] review inductive programming.

Chapter 6

Explore!

6.1 Unconventional Models of Computation

6.2 Relaxing from Symbolic to Neural

6.3 Reversible Computing

6.4 Further Reading

Examples of relaxing from symbolic to neural include Differentiable Forth [Bošnjak et al., 2016] and Differentiable ILP (Inductive Logic Programming) [Evans and Grefenstette, 2018].

Janus [Yokoyama and Glück, 2007] is a language for exploring reversible computing.
Acknowledgment

I thank Oliver Bračevac, William E. Byrd, Mistral Contrastin, Samuel Gruetter, Alan Kay, Oleg Kiselyov, Fengyun Liu, François-René Rideau, Tiark Rompf, Sandro Stucki, Alessandro Warth, Jeremy Yallop and the participants of the IFIP Working Groups on Language Design and on Functional Programming for many insightful discussions and pointers.
Bibliography


BIBLIOGRAPHY


