Metaprogramming Lecture Notes

Nada Amin
(namin@seas.harvard.edu)

Initial Version: October 2018, University of Cambridge, UK
Minor Update: January 2025
Working Draft: June 2025

Contents

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 metaprogramming, 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 `Term`s to `Val`ues. We can use trait mixin composition to share the `Const` nodes between `Term`s and `Val`ues. 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
```

```scala
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.

```scala
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# [Cisternino 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`.

```
;(car (cons A B)) == A
;(cdr (cons A B)) == B

(define my-cons
  (lambda (a b)
    (lambda (msg)
      (if (eq? msg 'car) a b))))

(define my-car
  (lambda (p)
  (p 'car)))

(define my-cdr
  (lambda (p)
    (p '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].

McCarthy [1981] gives a fun history of lisp.

Ferguson [1981] gives a fun introduction to Prolog.

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.

Reflective towers of interpreters started with Smith [1984], then continued with Brown [Wand and Friedman, 1986], Blond [Danvy and Malmkjaer, 1988], Black [Asai et al., 1996, Asai, 2014], Pink & Purple [Amin and Rompf, 2017a].

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 ichbins [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.

The idea of proof by reflection, changing theorem proving in the theory into evaluation in the metatheory, dates back to FOL [Weyhrauch, 1980].

Demers and Malenfant [1995] surveys reflection in logic, functional and object-oriented programming. Tanter [2009] reviews reflection for open implementation. Costantini [2002] surveys meta-reasoning, discussing seminal approaches such as FOL and 3-LISP.

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:

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

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:

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

Futamura projections:

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

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

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

The equations are easily verified using the equations for mix (P), and for interpreters (I) and compilers (C) above.

Verify (1):

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

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

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

Verify (2):

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

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

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

Verify (3):

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

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

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

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 Cook and Lämmel [2011] 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.

DSLs abound. Some fun examples include Halide [Ragan-Kelley et al., 2013] for image processing, Pan [Elliott, 2003, Elliott et al., 2003] for functional imaging well-suited for fractals Jones [2004], and OMeta/Ohm [Warth, 2018] for parsing.

Chapter 5

Synthesis

5.1 Further Reading

Gulwani et al. [2015] review inductive programming.

Sketching is an approach to program synthesis [Solar-Lezama, 2008] based on specification and holes. Osera and Zdancewic [2015] propose a type-and-example directed approach.

Chapter 6

Explore!

6.1 Languages as Interfaces: Innovations Above and Below

6.1.1 e.g.: Probabilistic programming

6.1.2 e.g.: Satisfiability Modulo Theories (SMT)

6.2 Unconventional Models of Computation

6.3 Relaxing from Symbolic to Neural

6.4 Reversible Computing

6.5 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]. My own explorations of relaxed machines are at `https://github.com/namin/relaxed-machines`

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

Acknowledgment

I thank Oliver Braçevac, 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

Martin Odersky. Scala by example, 2014. URL https://www.scala-lang.org/docu/files/ScalaByExample.pdf.

Martin Odersky, Lex Spoon, and Bill Venners. Programming in Scala: Updated for Scala 2.12. Artima Incorporation, USA, 3rd edition, 2016. ISBN 0981531687, 9780981531687.

Benjamin C. Pierce. Types and Programming Languages. The MIT Press, 1st edition, 2002. ISBN 0262162091, 9780262162098. URL https://www.cis.upenn.edu/~bcpierce/tapl/.

Harold Abelson and Gerald J. Sussman. Structure and Interpretation of Computer Programs. MIT Press, Cambridge, MA, USA, 2nd edition, 1996. ISBN 0262011530. URL https://mitpress.mit.edu/sites/default/files/sicp/full-text/book/book.html.

Peter Norvig. Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992. ISBN 1558601910.

Leon Sterling and Ehud Shapiro. The Art of Prolog (2Nd Ed.): Advanced Programming Techniques. MIT Press, Cambridge, MA, USA, 1994. ISBN 0-262-19338-8.

Kenneth D. Forbus and Johan De Kleer. Building Problem Solvers. MIT Press, Cambridge, MA, USA, 1993. ISBN 0262560720.

Beck. Test Driven Development: By Example. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 0321146530.

A. Cisternino, A. Granicz, and D. Syme. Expert F#. Apress, 2008. ISBN 9781430214786.

Valentino Braitenberg. Vehicles: Experiments in Synthetic Psychology. Bradford Books. MIT Press, 1986. ISBN 9780262521123. Alan Kay suggests Robot Odyssey, rewired at https://www.robotodyssey.online/.

Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. The Reasoned Schemer. The MIT Press, 2005. ISBN 0262562146.

Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. The Reasoned Schemer. The MIT Press, 2nd edition, 2018. ISBN 9780262535519.

John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. Commun. ACM, 3(4):184–195, April 1960. ISSN 0001-0782. doi: 10.1145/367177.367199. URL http://www-formal.stanford.edu/jmc/recursive.pdf.

John C. Reynolds. Definitional interpreters for higher-order programming languages. In Proceedings of the ACM Annual Conference - Volume 2, ACM '72, pages 717–740, New York, NY, USA, 1972. ACM. doi: 10.1145/800194.805852. URL http://homepages.inf.ed.ac.uk/wadler/papers/papers-we-love/reynolds-definitional-interpreters-1972.pdf.

John McCarthy. History of lisp. In Richard Wexelblat, editor, History of Programming Languages. Academic Press, 1981. URL http://jmc.stanford.edu/articles/lisp/lisp.pdf.

Ron Ferguson. Prolog: A step towards the ultimate computer language, 1981. URL http://tinlizzie.org/~awarth/prolog-ferguson81.pdf.

Ian Piumarta. Open, extensible composition models. In Proceedings of the 1st International Workshop on Free Composition, FREECO '11, pages 2:1–2:5, New York, NY, USA, 2011a. ACM. ISBN 978-1-4503-0892-2. doi: 10.1145/2068776.2068778. URL http://piumarta.com/freeco11/freeco11-piumarta-oecm.pdf.

Michael Codish and Harald Sondergaard. Meta-circular abstract interpretation in prolog. In David Schmidt Torben Mogensen and I. Hal Sudburough, editors, The Essence of Computation: Complexity, Analysis, Transformation, volume 2566 of Lecture Notes in Computer Science, pages 109–134. Springer-Verlag, 2002. URL https://www.cs.bgu.ac.il/~mcodish/Tutorial/.

David Darais, Nicholas Labich, Phúc C. Nguyen, and David Van Horn. Abstracting definitional interpreters (functional pearl). Proc. ACM Program. Lang., 1(ICFP):12:1–12:25, August 2017. ISSN 2475-1421. doi: 10.1145/3110256. URL https://plum-umd.github.io/abstracting-definitional-interpreters/.

Ken Thompson. Reflections on trusting trust. Commun. ACM, 27(8):761–763, August 1984. ISSN 0001-0782. doi: 10.1145/358198.358210. URL http://doi.acm.org/10.1145/358198.358210.

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. Cakeml: A verified implementation of ml. In Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, pages 179–191, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2544-8. doi: 10.1145/2535838.2535841. URL https://cakeml.org/popl14.pdf.

Ramana Kumar. Self-compilation and self-verification. PhD thesis, University of Cambridge, Compter Laboratory, 2016. URL http://www.sigplan.org/Awards/Dissertation/2017_kumar.pdf.

Mitchell Wand. The theory of fexprs is trivial. LISP and Symbolic Computation, 10(3):189–199, May 1998. ISSN 1573-0557. doi: 10.1023/A:1007720632734. URL http://www.ccs.neu.edu/home/wand/papers/fexprs.ps.

Brian Cantwell Smith. Reflection and semantics in lisp. In Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL '84, pages 23–35, New York, NY, USA, 1984. ACM. ISBN 0-89791-125-3. doi: 10.1145/800017.800513. URL http://doi.acm.org/10.1145/800017.800513.

Mitchell Wand and Daniel P. Friedman. The mystery of the tower revealed: A non-reflective description of the reflective tower. In Proceedings of the 1986 ACM Conference on LISP and Functional Programming, LFP '86, pages 298–307, New York, NY, USA, 1986. ACM. ISBN 0-89791-200-4. doi: 10.1145/319838.319871. URL http://doi.acm.org/10.1145/319838.319871.

Olivier Danvy and Karoline Malmkjaer. Intensions and extensions in a reflective tower. In Proceedings of the 1988 ACM Conference on LISP and Functional Programming, LFP '88, pages 327–341, New York, NY, USA, 1988. ACM. ISBN 0-89791-273-X. doi: 10.1145/62678.62725. URL http://doi.acm.org/10.1145/62678.62725.

Kenichi Asai, Satoshi Matsuoka, and Akinori Yonezawa. Duplication and partial evaluation. LISP and Symbolic Computation, 9(2):203–241, May 1996. ISSN 1573-0557. doi: 10.1007/BF01806113. URL https://github.com/readevalprintlove/black.

Kenichi Asai. Compiling a reflective language using metaocaml. In Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences, GPCE 2014, pages 113–122, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3161-6. doi: 10.1145/2658761.2658775. URL http://doi.acm.org/10.1145/2658761.2658775.

Nada Amin and Tiark Rompf. Collapsing towers of interpreters. Proc. ACM Program. Lang., 2(POPL):52:1–52:33, December 2017a. ISSN 2475-1421. doi: 10.1145/3158140. URL http://popl18.namin.net.

Gregor Kiczales, J. Michael Ashley, Luis Rodriguez, Amin Vahdat, and Daniel G. Bobrow. Metaobject protocols: Why we want them and what else they can do, 1993. URL http://cseweb.ucsd.edu/~vahdat/papers/mop.pdf.

Gilad Bracha and David Ungar. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. In Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '04, pages 331–344, New York, NY, USA, 2004. ACM. ISBN 1-58113-831-8. doi: 10.1145/1028976.1029004. URL http://bracha.org/mirrors.pdf.

Ian Piumarta and Alessandro Warth. Open, extensible object models. In Self-Sustaining Systems, pages 1–30. Springer, 2008. URL http://piumarta.com/software/cola/objmodel2.pdf.

David A Fisher. Control structures for programming languages. PhD thesis, CMU, 1970.

Adele Goldberg and Alan Kay. Smalltalk-72: Instruction Manual. XEROX Parc, 1976. URL http://bitsavers.informatik.uni-stuttgart.de/pdf/xerox/parc/techReports/Smalltalk-72_Instruction_Manual_Mar76.pdf.

Adele Goldberg and David Robson. Smalltalk-80: The Language and Its Implementation. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983. ISBN 0-201-11371-6. URL http://stephane.ducasse.free.fr/FreeBooks/BlueBook/Bluebook.pdf.

Edgar T. Irons. Experience with an extensible language. Commun. ACM, 13 (1):31–40, January 1970. ISSN 0001-0782. doi: 10.1145/361953.361966. URL http://doi.acm.org/10.1145/361953.361966.

Lawrence G Tester, Horace J Enea, and David C Smith. The lisp70 pattern matching system. Proceedings of the Third International Joint Conference on Artificial Intelligence, 1973. URL https://www.ijcai.org/Proceedings/73/Papers/073.pdf.

D. V. Schorre. Meta II: a syntax-oriented compiler writing language, 1964. URL http://www.ibm-1401.info/Meta-II-schorre.pdf.

Darius Bacon. Ichbins: a self-hosting compiler of a lisp dialect to c in 6 pages of code, 2007. URL https://github.com/darius/ichbins.

Ian Piumarta. Maru: a symbolic expression evaluator that can compile its own implementation language, 2011b. URL http://piumarta.com/software/maru/.

Jonathan M Sobel and Daniel P Friedman. An introduction to reflection-oriented programming. In Reflection, 1996. URL http://www2.parc.com/csl/groups/sda/projects/reflection96/docs/sobel/sobel.pdf.

G. Martínez, D. Ahman, V. Dumitrescu, N. Giannarakis, C. Hawblitzel, C. Hritcu, M. Narasimhamurthy, Z. Paraskevopoulou, C. Pit-Claudel, J. Protzenko, T. Ramananandro, A. Rastogi, and N. Swamy. Meta-F*: Proof Automation with SMT, Tactics, and Metaprograms. ArXiv e-prints, March 2018. URL https://arxiv.org/abs/1803.06547.

Yves Bertot and Pierre Castéran. * Proof by Reflection, pages 433–448. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. ISBN 978-3-662-07964-5. doi: 10.1007/978-3-662-07964-5_16. URL https://doi.org/10.1007/978-3-662-07964-5_16.

Adam Chlipala. Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant, chapter 15. Proof by Reflection. The MIT Press, 2013. ISBN 0262026651, 9780262026659. URL http://adam.chlipala.net/cpdt/html/Cpdt.Reflection.html.

Nils Anders Danielsson. A very simple theorem prover, using the technique of proof by reflection, 2013. URL http://www.cse.chalmers.se/edu/year/2013/course/afp/lectures/lecture12/html/Proof-by-reflection.html.

Aaron Stump. Verified Functional Programming in Agda, chapter 6.3 Proof by Reflection. Association for Computing Machinery and Morgan & Claypool, New York, NY, USA, 2016. ISBN 978-1-97000-127-3.

Paul van der Walt and Wouter Swierstra. Engineering Proof by Reflection in Agda. In Ralf Hinze, editor, IFL - 24th International Symposium on Implementation and Application of Functional Languages, volume 8241 of Lecture Notes in Computer Science, pages 157–173, Oxford, United Kingdom, August 2012. Springer. doi: 10.1007/978-3-642-41582-1_10. URL https://hal.inria.fr/hal-00987610.

Benja Fallenstein and Ramana Kumar. Proof-producing reflection for hol. In International Conference on Interactive Theorem Proving, pages 170–186. Springer, 2015. URL https://cakeml.org/itp15-reflection.pdf.

John Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK, 1995. URL http://www.cl.cam.ac.uk/~jrh13/papers/reflect.pdf.

Richard W Weyhrauch. Prolegomena to a theory of mechanized formal reasoning. Artificial intelligence, 13(1-2):133–170, 1980. Try GETFOL: https://io.livecode.ch/learn/namin/GETFOL.

François-Nicola Demers and Jacques Malenfant. Reflection in logic, functional and object-oriented programming: a short comparative study. In In IJCAI '95 Workshop on Reflection and Metalevel Architectures and their Applications in AI, pages 29–38, 1995.

Eric Tanter. Reflection and open implementations, 2009. URL https://www.dcc.uchile.cl/TR/2009/TR_DCC-20091123-013.pdf.

Stefania Costantini. Meta-reasoning: A survey. In Computational Logic: Logic Programming and Beyond: Essays in Honour of Robert A. Kowalski Part II, pages 253–288. Springer, 2002. URL https://people.disim.univaq.it/stefcost/pubbls/metarev.pdf.

Jacques Pitrat. Des métaconnaissances pour des sysèmes intelligents. In Quaderni, volume 25, pages 29–42, 1995. URL https://www.persee.fr/doc/quad_0987-1381_1995_num_25_1_1110.

N. D. Jones, C. K. Gomard, A. Bondorf, O. Danvy, and T. A. Mogensen. A self-applicable partial evaluator for the lambda calculus. In Proceedings. 1990 International Conference on Computer Languages, pages 49–58, March 1990. doi: 10.1109/ICCL.1990.63760.

Nada Amin and Tiark Rompf. Lms-verify: abstraction without regret for verified systems programming. In Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017, pages 859–873, 2017b. URL http://github.com/namin/lms-verify.

Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. Partial Evaluation and Automatic Program Generation. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993. ISBN 0-13-020249-5. URL https://www.itu.dk/~sestoft/pebook/jonesgomardsestoft-letter.pdf.

Yoshihiko Futamura. Partial evaluation of computation process–an approach to a compiler-compiler. Systems Computers Controls, 2(5), 1971.

Yoshihiko Futamura. Partial evaluation of computation process, revisited. Higher Order Symbol. Comput., 12(4):377–380, December 1999. ISSN 1388-3690. doi: 10.1023/A:1010043619517. URL https://doi.org/10.1023/A:1010043619517.

William R Cook and Ralf Lämmel. Tutorial on online partial evaluation. arXiv preprint arXiv:1109.0781, 2011. URL http://www.cs.utexas.edu/~wcook/tutorial/.

Tiark Rompf and Nada Amin. Functional pearl: A sql to c compiler in 500 lines of code. In Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, pages 2–9, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3669-7. doi: 10.1145/2784731.2784760. URL https://www.cs.purdue.edu/homes/rompf/papers/rompf-icfp15.pdf.

Tiark Rompf and Martin Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled dsls. Commun. ACM, 55(6):121–130, June 2012. ISSN 0001-0782. doi: 10.1145/2184319.2184345. URL http://scala-lms.github.io.

Oleg Kiselyov. Reconciling abstraction with high performance: A metaocaml approach. Foundations and Trends® in Programming Languages, 5(1):1–101,

2018. ISSN 2325-1107. doi: 10.1561/2500000038. URL http://okmij.org/ftp/meta-programming/tutorial/.

J Michael Spivey and Silvija Seres. Embedding prolog in haskell. In Haskell, 1999. URL http://www.silvija.net/0000OxfordPublications/seres_haskell99.pdf.

William E Byrd, Eric Holk, and Daniel P Friedman. minikanren, live and untagged. In Scheme Workshop, 2012. URL http://webyrd.net/quines/quines.pdf.

Jason Hemann and Daniel P. Friedman. µkanren: A minimal functional core for relational programming. In Scheme Workshop, 2013. URL http://webyrd.net/scheme-2013/papers/HemannMuKanren2013.pdf.

William E. Byrd, Michael Ballantyne, Gregory Rosenblatt, and Matthew Might. A unified approach to solving seven programming problems (functional pearl). Proc. ACM Program. Lang., 1(ICFP):8:1–8:26, August 2017. ISSN 2475-1421. doi: 10.1145/3110252. URL http://io.livecode.ch/learn/gregr/icfp2017-artifact-auas7pp.

Norman Ramsey and Avi Pfeffer. Stochastic lambda calculus and monads of probability distributions. In Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '02, pages 154–165, New York, NY, USA, 2002. ACM. ISBN 1-58113-450-9. doi: 10.1145/503272.503288. URL https://www.cs.tufts.edu/~nr/pubs/pmonad.pdf.

Oleg Kiselyov and Chung-chieh Shan. Embedded probabilistic programming. In Walid Mohamed Taha, editor, Domain-Specific Languages, pages 360–384, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-03034-5. URL http://okmij.org/ftp/kakuritu/Hansei.html.

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2014-6. doi: 10.1145/2491956.2462176. URL http://people.csail.mit.edu/jrk/halide-pldi13.pdf.

Conal Elliott. Functional images. In The Fun of Programming, "Cornerstones of Computing" series. Palgrave, March 2003. URL http://conal.net/papers/functional-images/.

Conal Elliott, Sigbjørn Finne, and Oege de Moor. Compiling embedded languages. Journal of Functional Programming, 13(2), 2003. URL http://conal.net/papers/jfp-saig/. Updated version of paper by the same name that appeared in SAIG '00 proceedings.

Mark P. Jones. Functional pearl composing fractals. Journal of Functional Programming, 14(6):715–725, 2004. doi: 10.1017/S0956796804005167. URL http://web.cecs.pdx.edu/~mpj/pubs/composing-fractals.pdf.

Alessandro Warth. Experimenting with Programming Languages. PhD thesis, UCLA, 2018. URL http://www.vpri.org/pdf/tr2008003_experimenting.pdf.

Sumit Gulwani, José Hernández-Orallo, Emanuel Kitzelmann, Stephen H. Muggleton, Ute Schmid, and Benjamin Zorn. Inductive programming meets the real world. Commun. ACM, 58(11):90–99, October 2015. ISSN 0001-0782. doi: 10.1145/2736282. URL http://doi.acm.org/10.1145/2736282.

Armando Solar-Lezama. Program Synthesis by Sketching. PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 2008. URL https://people.csail.mit.edu/asolar/papers/thesis.pdf. AAI3353225.

Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '15, pages 619–630, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3468-6. doi: 10.1145/2737924.2738007. URL https://www.cis.upenn.edu/~stevez/papers/OZ15.pdf.

Matko Bošnjak, Tim Rocktäschel, Jason Naradowsky, and Sebastian Riedel. Programming with a differentiable forth interpreter. arXiv preprint arXiv:1605.06640, 2016. URL https://github.com/uclmr/d4.

Richard Evans and Edward Grefenstette. Learning explanatory rules from noisy data. Journal of Artificial Intelligence Research, 61:1–64, 2018.

Tetsuo Yokoyama and Robert Glück. A reversible programming language and its invertible self-interpreter. In Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM '07, pages 144–153, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-620-2. doi: 10.1145/1244381.1244404. URL https://topps.diku.dk/pirc/janus-playground/.