For each project I include some suggested core goals, and some further possibilities. All of these though are suggestions — other directions or choices of content can be fine

Reading suggestions are in several cases incomplete — feel free for more suggestions as necessary (either if they're missing, or if the given suggestions aren't to your taste). In longer references, I've aimed to give pointers to specific chapters/sections; where these are missing, again please ask!

For any implementation projects: Use any programming language you know and like; generally I recommend Haskell, OCaml, SML, or similar, as well-suited to these purposes. Or, of course, use a proof assistant — Coq, Agda, Lean…

1 Simply typed λ-calculus

Project (CCC semantics)**.** Cartesian closed category ("CCC") semantics of simple type theory.

Core goals: definition of CCC's; syntax forms a CCC; syntax can be interpreted in any CCC. Further possibilities: (2-)initiality of the syntactic category.

Background: a little category theory.

Literature: Awodey 2006; Harper 2016; Lambek and P. J. Scott 1986

Project (Stack machines)**.** Stack machine operational semantics of simple type theory.

Core goals: definition of the stack machine and execution, and termination proof for a small core theory. Further possibilities: extension to larger theories; comparison with other semantics ,

Literature: Harper 2016.

Project (Tait computability beyond $\to$)**.** Normalisation using Tait-style hereditary normalisability, for systems with more type constructors than given in class.

Core goal: proof of (weak) normalisation for extension of type theory with products, maybe a little more. Further possibilities: discussion/analysis; extension to strong normalisation; more constructors?

Literature: Harper 2022, Angiuli 2015

Project (Normalisation strategies)**.** Call-by-name vs call-by-value normalisation strategies, for simple type theory.

Core goals: define these reduction strategies; informally compare, with examples. Further possibilities: some precise comparisons.

Reading: Harper 2016

Project (Implementation)**.** A computer-implementation of a core simple type theory, in some programming language.

Core goals: Define the syntax, including substitution, and some examples demonstrating it. Further possibilities: A normaliser/evaluator; an interpreter

into your host language; a parser for input (warning — parsers have a steep learning curve, that will be a lot of work if you haven't done one before).

Background: Some programming experience.

Reading: Pierce 2002, Sitnikovski 2019

2 Untyped λ-calculus

Project (Domain semantics)**.** Domain semantics for untyped λ-calculus (or some other similar system, eg PCF).

Core goal: Definition of Scott domains; function domain; construction of Scott's D_∞; interpretation of some core syntax. Further possibilities: connection with the C-monoid analysis (see below) and/or CCC semantics (see above).

Background: A little topology.

Reading: D. S. Scott 1970, Barendregt 2012, p. V.18.2, Pitts 2012, §8, Lambek and P. J. Scott 1986, p. II.18

Project (Equivalence with combinatory logic)**.** Equivalence between untyped λ-calculus and SKI combinator logic ("CL").

Core goal: Definition of the combinatory system CL; proof of the translations between it and λ-calculus, and that they are suitable mutually inverse. Further possibilities: more comparison of the two systems, advantages of each; historical context; relationship to C-monoid semantics (see below).

Reading: Barendregt 2012, p. II.7, Smullyan 1985 (puzzle book!)

Project (C-monoid semantics)**.** C-monoid semantics for untyped λ-calculus.

Core goal: Definition of C-monoids; interpretation of λ-calculus in a C-monoid; sketch of some example. Further possibilities: Detailed presentation of some example (e.g. the domain D_∞, see above); connection with the CCC semantics (see above).

Reading: Lambek and P. J. Scott 1986, I.15–18

Project (Implementation)**.** A computer-implementation of a untyped λ-calculus, in some programming language (any language you prefer, but I recommend Haskell, OCaml, SML, or similar).

Core goals: Define the syntax, including substitution, and some examples demonstrating it. Further possibilities: A normaliser/evaluator; an interpreter into your host language; a parser for input (warning — parsers have a steep learning curve, that will be a lot of work if you haven't done one before).

Background: Some programming experience.

Reading: Pierce 2002, Sitnikovski 2019

3 Dependent type theory

Project (Impredicative universes)**.** Impredicative universes in dependent type theory — applications, problems, connections.

Core goals: Definition of an impredicative set universe, and at least one of the further options. Further possibilities: impredicative encoding of inductive types; history in Martin-Löf type theory and elsewhere; Girard's paradox; Hurkens' paradox; translation from System F (see below).

Reading: Martin-Löf 1984, Werner 1994.

Project (Formalisation)**.** Formalisation in dependent type theory of some mathematical topic.

Core goals: Very broad topic! Pick a proof assistant (recommended: Coq, Agda, Lean); choose some topic; and develop some material in that topic, either using an existing library as a base or possibly (if your topic is very elementary) starting from scratch. Ask me for topic recommendations depending on your interests! Further possibilities: Contribute your formalisation to the library you've used, if it's on material not yet in there.

Experience: Some programming helpful.

Reading: Online documentation + community resources for chosen proof assistant.

4 Other

Project. System F Girard's polymorphic type thery, "system F".

Core goal: Definition of system F. Further possibilities: Impredicative encoding of inductive types; comparison with untyped λ-calculus; comparison with dependent type theory using an impredicative universe (see above); some semantics.

Reading: Girard, Taylor, and Lafont 1989.

References

Angiuli, Carlo (2015). "How to prove that STLC is normalizing". unpublished note. URL: https://www.cs.cmu.edu/~rwh/courses/chtt/pdfs/angiuli.pdf (visited on 04/15/2023).

Awodey, Steve (2006). *Category theory*. Vol. 49. Oxford Logic Guides. The Clarendon Press, Oxford University Press, New York, pp. xii+256. ISBN: 0-19-856861-4; 978-0-19-856861-2. DOI: 10.1093/acprof:oso/9780198568612.001.0001.

Barendregt, Henk P. (2012). *The lambda calculus, its syntax and semantics*. Vol. 40. Studies in Logic (London). [Reprint of the 1984 revised edition, MR0774952], With addenda for the 6th imprinting, Mathematical Logic and Foundations. College Publications, London, xvi+622+E16. ISBN: 978-1-84890-066-0.

Girard, Jean-Yves, Paul Taylor, and Yves Lafont (1989). *Proofs and types*. Vol. 7. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, pp. xii+176. ISBN: 0-521-37181-3.

Harper, Robert (2016). *Practical Foundations for Programming Languages*. 2nd. USA: Cambridge University Press. ISBN: 1107150302.

Harper, Robert (2022). “Kripke-Style Logical Relations for Normalization”. URL: https://www.cs.cmu.edu/~rwh/courses/chtt/pdfs/kripke.pdf (visited on 04/15/2023).

Lambek, J. and P. J. Scott (1986). *Introduction to higher order categorical logic*. Vol. 7. Cambridge Studies in Advanced Mathematics. Cambridge University Press, Cambridge, pp. x+293. ISBN: 0-521-24665-2.

Martin-Löf, Per (1984). *Intuitionistic type theory. notes by Giovanni Sambin*. Vol. 1. Studies in Proof Theory. Lecture Notes. Naples: Bibliopolis, pp. iv+91. ISBN: 88-7088-105-9.

Pierce, Benjamin C. (2002). *Types and Programming Languages*. 1st. The MIT Press. ISBN: 0262162091.

Pitts, Andrew (2012). URL: https://www.cl.cam.ac.uk/teaching/1112/DenotSem/dens-notes-bw.pdf (visited on 04/29/2023).

Scott, Dana S. (Nov. 1970). *Outline of a Mathematical Theory of Computation*. Tech. rep. PRG–2.

Sitnikovski, Boro (Mar. 15, 2019). blog post. URL: https://bor0.wordpress.com/2019/03/15/writing-a-simple-evaluator-and-type-checker-in-haskell/ (visited on 04/29/2023).

Smullyan, Raymond (1985). *To Mock a Mockingbird*. Alfred A. Knopf.

Werner, Benjamin (May 1994). “Une théorie des constructions inductives”. PhD thesis. Université Paris 7 (Denis Diderot).