Monthly Report: October 2025

Samuel Laaksonen

October 2025

1 Aim and Goals:

My subgoal this month was to learn more about Abstract Interpretation

1.1 Abstracting Abstract Machines

I continued implementing this paper[1] and was able to go on upto a simple k-CFA analysis where each state is instrumented by the last k positions where it was called.

1.2 Allocation characterizes Polyvariance

I also started studying this [2] paper to better implement the different analyses that are relevant for functional programs, such as object and closure creation analyses. This paper provides a structured way of implementing the machine where only the allocator (and instrumentation portion of the state) have to be modified to accomodate finer analyses as well as variable degrees of abstract interpretation based on the current knowledge of the system and the desired goal of the analysis.

1.3 Simple visualization and API for the implementation

One thing that I am working towards is a simple visualization of the analyses which can be both used to debug my implementation as well as allow for the development of an API that other tools can plug into to perhaps annotate/ speed up parts of the analysis. Currently it is just a React component that has a graph visualization as well as a debugger style panel that will display the state of the abstract machine + relevant addresses of the store when selected.

1.4 A verified AAM Machine

Professor suggested I read a paper on how to construct a verifiable prrof of Galois Connections[3].

The proof is in Agda so my first step was to get Agda setup on my machine. My first task is to learn enough Agda to be able to port the proof over to either Rocq or Lean.

Rocq would be useful since then I can use the power of CompCert[4] to ensure that post analyses, I can emit a subset of C code that can then be optimized by CompCert. However if I wished to directly emit x64/ ARM assembly, I might look for an alternative.

Lean is a contender for the reason that I might be able to use the proof of the Etch compiler [5], use the abstractions in that paper to define a Datalog engine like FVLog[6] to then create a flow from a specification of an AAM based analyses to a Datalog Program, which is then compiled with the high-level verified compilation passes of Etch and the runtime datastructures of FVLog to emit fast RA CUDA kernels and run them quickly.

1.5 Testing Dataset

If I need to have any hope of being able to successfully implement any of the above, I need a solid dataset of example programs to run the implementations on, collect metrics about, and prove that any hopeful achievement of this endeavour is non-trivial and can potentially be useful in the real world.

To this end, Professor suggested I first start out with a small set of hand-written non trivial programs that have interesting execution pattterns that will test all the features of the analyses and whose outputs can be manually verified.

I think the best way to start would be to implement a few common algorithms in the subset of Lisp that is focused on in the Compiler Construction class.

It would be a dream to be able to match the speed and efficiency of Chez Scheme [7] using AAM based optimization passes, but it would be much beyond my capabilities at this point. Nevertheless as a northstar, I am willing to take inspiration on some aspects of this and improve where possible the Class Compiler.

A concrete goal would be to add a small autovectorization pass on the class compiler, where loops over numerical algorithms are identified and the relevant AVX instructions are emitted to perform the loops in a more parallel fashion.

Studying methods of improving GC in the runtime of the Machine would also be a nontrivial goal to work towards. Towards this end, I have procured this book *The Garbage Collection Handbook*[8] from which I intend to take inspiration.

2 Concrete Steps

This Month I intend to ensure the following are completed:

- Complete the class compiler upto Functions
- Complete atleast 20 different algorithms to test the AAM analyses on

- Begin work on the visualizer and tune it continually until it becomes a joy to use.
- Perform the *Constant Folding* optimization.
- Perform the *Register Allocation* optimization.
- Perform the *Tail Call* optimization.
- Perform the *Common Subexpression Elimination* optimization.
- Perform the *Hash Consing Memoization* dynamic optimization.

References

[1] David Van Horn and Matthew Might. Abstracting abstract machines, 2010.

[2] Thomas Gilray, Michael D. Adams, and Matthew Might. Allocation characterizes polyvariance: a unified methodology for polyvariant control-flow analysis. *SIGPLAN Not.*, 51(9):407–420, September 2016.

[3] DAVID DARAIS and DAVID VAN HORN. Constructive galois connections. *Journal of Functional Programming*, 29, 2019.

[4] Sandrine Blazy and Xavier Leroy. Formal verification of a memory model for C-like imperative languages. In *International Conference on Formal Engineering Methods (ICFEM 2005)*, volume 3785 of *Lecture Notes in Computer Science*, pages 280–299. Springer, 2005.

[5] Scott Kovach, Praneeth Kolichala, Tiancheng Gu, and Fredrik Kjolstad. Indexed streams: A formal intermediate representation for fused contraction programs. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023.

[6] Yihao Sun, Sidharth Kumar, Thomas Gilray, and Kristopher Micinski. Column-oriented datalog on the gpu, 2025.

[7] R. Kent Dybvig. The development of chez scheme. *SIGPLAN Not.*, 41(9):1–12, September 2006.

[8] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edition, 2011.