The Dependent Arithmetic Machine

Alexandra Aiello

December 19, 2025

Contents

1 Abstract

I present a the Dependent Arithmetic Machine: an implementation of intensional Marin-Löf type theory based on the SK combinator calculus with a direct correspondence to zero-knowledge algebraic intermediate representations (*ZK-AIR*). I present an arithmetization encoding type-checking in the calculus based on an interpretation of the syntactic category corresponding to the typed calculus as graph constraints in a ZK-AIR. I also outline a **Lean 4** implementation of the ZK-AIR and demonstrate some interesting properties of it.

2 Dependent SK Combinators

2.1 Our Type Equations

- $c(\mathbb{1}) = 1$
- $c(I) = 1$
- $c(K) = 2$
- $c(S) = 3$
- $c(MI) = 4$
- $c(MK) = 5$

- $c(MS) = 6$
- $c(Ty_n) = n + 6$
- $MN = c(m) \cdot c(N)$

Edges:

- $(1, n + 6)$
- $(1, 4)$

Thoughts:

- Need some notion of an atomic expression. Rows are atomics, edges are applications?
- Rows = arguments. We need to check all of them. We're not interested in β-reduction
- First row = spine
- Each atomic element has an "obvious" type - we can infer this from the rules
- Each argument only has one edge. It's an argument for exactly one node, so we need vertices per row

We can make constraints to make sure each app has exactly the number of arguments we want. I like our path approach.

If every row has an "obvious" type, how do we combine types together? It's either: just concatenate them reduce them

Each element has an obvious type. Apps are what we're concerned with.

Mapping + constraints. I is just a mapping over the type. Nondepednent K is just a mapping over the type. We're just exchanging.

Dependent K has extra constraints.

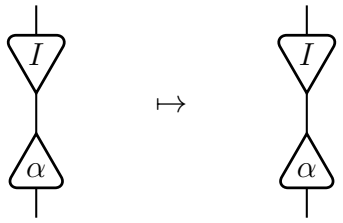
"To follow this path, this path must already exist". It's a conditional. There's also beta reduction, we can't foget about that.

What if we mix types and terms? Assume x is an atomic term. Then, it has a well-known type: $\langle x, t \rangle$.

Interaction nets idea:

When we have the requisite number of arguments for a combinator, a constraint applies.

Thought: we don't need to model the types of our base combinators in the ZK-AIR itself. The AIR is the equivalent of the infer function. The infer function says nothing about our actual combinator types and values.



These can be rows themselves. We can create ad-hoc combinators, or inference rules, as long as they are founded on previous inference rules.

Our core inference rules are encoded in the AIR constraints, but our actual formulae are lines in the table.

This way we can kind of add more as we see fit.

Equational design is kind of BS, since we can't type-check terms that don't reduce to β-normal forms. We can use our old meta-combinator hierarchy though. Each one is well-typed even in "uncurried" form.

The issue is with typing S. I'm curious to see if we can adapt the de bruijn approach from the CAM paper to LCCC's instead of just CCC's.

With calculus of constructions, we will need to implement contexts and type unification, which is really annoying.

Unflattened meta combintaor approach is really annoying, since dependent S has a lot of inferene rules.

Munchausen method by Alternkirch [Alt+23, p. 15]:

- Forward-declarations to postpone value assignments when a type depends on a term
- Example: church-encoded dependent product $\Sigma AB \equiv \lambda(b : Bool), ifbthenAelseB$?
  - ? is postponed by declaring $sigma - helper : A$ and accompanying the encoding of the sigma with the rewerite rule $val : A\ sigma - helper(true) = val$
  - We can probably do something similar for combinators, as they showed [Alt+23, p. 15]. But we will need to make the theory more "freestanding", since their formalism leans on Agda's support for extensionality (UIP?)

References

[Alt+23] Thorsten Altenkirch et al. "The Münchhausen Method in Type Theory". In: *28th International Conference on Types for Proofs and Programs (TYPES 2022)*. Ed. by Delia Kesner and Pierre-Marie Pédrot. Vol. 269. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss

Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 10:1–10:20. ISBN: 978-3-95977-285-3. DOI: 10.4230/LIPIcs.TYPES.2022.10. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.TYPES.2022.10.

