Compiling with Arrays (Artifact)

David Richter
Technische Universität Darmstadt, Germany

Timon Böhler
Technische Universität Darmstadt, Germany

Pascal Weisenburger
University of St. Gallen, Switzerland

Mira Mezini
Technische Universität Darmstadt, Germany
The Hessian Center for Artificial Intelligence (hessian.AI), Darmstadt, Germany

Abstract

Linear algebra computations are foundational for neural networks and machine learning, often handled through arrays. While many functional programming languages feature lists and recursion, arrays in linear algebra demand constant-time access and bulk operations. To bridge this gap, some languages represent arrays as (eager) functions instead of lists. In this paper, we connect this idea to a formal logical foundation by interpreting functions as the usual negative types from polarized type theory, and arrays as the corresponding dual positive version of the function type. Positive types are defined to have a single elimination form whose computational interpretation is pattern matching. Just like (positive) product types bind two variables during pattern matching, (positive) array types bind variables with *multiplicity* during pattern matching. We follow a similar approach for Booleans by introducing conditionally-defined variables.

The positive formulation for the array type enables us to combine typed partial evaluation and common subexpression elimination into an elegant algorithm whose result enjoys a property we call maximal fission, which we argue can be beneficial for further optimizations. For this purpose, we present the novel intermediate representation *indexed administrative normal form (A_iNF)*, which relies on the formal logical foundation of the positive formulation for the array type to facilitate maximal loop fission and subsequent optimizations. A_iNF is normal with regard to commuting conversion for both let-bindings and for-loops, leading to flat and maximally fissioned terms. We mechanize the translation and normalization from a simple surface language to A_iNF, establishing that the process terminates, preserves types, and produces maximally fissioned terms.

2012 ACM Subject Classification Software and its engineering → Domain specific languages

Keywords and phrases array languages, functional programming, domain-specific languages, normalization by evaluation, common subexpression elimination, polarity, positive function type, intrinsic types

Digital Object Identifier 10.4230/DARTS.10.2.18

Funding *Timon Böhler*: LOEWE/4a//519/05/00.002(0013)/95
Pascal Weisenburger: Swiss National Science Foundation (SNSF, No. 200429)
Mira Mezini: LOEWE/4a//519/05/00.002(0013)/95; HMWK cluster project *The Third Wave of Artificial Intelligence* (3AI).

Related Article David Richter, Timon Böhler, Pascal Weisenburger, and Mira Mezini, "Compiling with Arrays", in 38th European Conference on Object-Oriented Programming (ECOOP 2024), LIPIcs, Vol. 313, pp. 33:1–33:24, 2024.
`https://doi.org/10.4230/LIPIcs.ECOOP.2024.33`

Related Conference 38th European Conference on Object-Oriented Programming (ECOOP 2024), September 16–20, 2024, Vienna, Austria

Evaluation Policy The artifact has been evaluated as described in the ECOOP 2024 Call for Artifacts and the ACM Artifact Review and Badging Policy.

1 Scope

This artifact contains the Polara language and AINF intermediate representation implemented as a DSL in Lean4.

1.1 Relation of Artifact and Paper

Section 4 "Mechanization" is based on the contents of the src/ folder. In particular:

- Figure 2+3: `src/Polara/Syntax.lean`
- Figure 4a: `Ty.de` in `src/Polara/NbE.lean`
- Figure 4b+c: `Const0.de, Const1.de, Const2.de, Tm.de` in `src/Polara/NbE.lean`
- Figure 4d: `quote, splice, Tm.norm` in `src/Polara/NbE.lean`
- Figure 5a: `Tm.toAINF` in `src/Polara/CSE.lean`
- Figure 5b: `AINF.smart_bnd` in `src/Polara/CSE.lean`
- Figure 6: `AINF.cse` in `src/Polara/CSE.lean`
- Theorem 1: `Tm.norm` in `src/Polara/NbE.lean`
- Theorem 2: `Tm.toAINF` in `src/Polara/CSE.lean`

(Remark: Our development uses intrinsic proofs. An extrinsic proof is a separate function and a proof of a property over the function. An intrinsic proof unifies the function and the proof, meaning that proof checking is performed by type checking. Therefore, you won't find `theorems` in the source.)

Further, the two examples from Section 3.3 (a dense neural network layer and convolution) are implemented in `src/Polara/Examples.lean` as `dense` and `conv`.

1.2 Differences to the Paper

Comparing the functions in the paper to those in the artifact, one can see that, in the latter, functions relating to parametric higher-order abstract syntax (PHOAS) terms as well as the type constructor for terms itself take an additional argument Γ. This represents the denotation of the variables and is what distinguishes PHOAS from normal HOAS. We omit this technical detail in the paper. The type of terms `Tm` features constructors for variables and constants (`var`, `cst0` etc.). In the paper, we do not write these constructors explicitly, so we would write `x` rather than `var x`. In the artifact, the language has a `nat` type of natural numbers. This is omitted in the paper for brevity, which means that if-then-else takes a `nat` as the condition in the artifact, but a `fin 2` in the paper.

The `smart_bnd` function takes an additional number argument, wrapped inside a reader monad, which is used for creating fresh variables. In the paper, we leave this out and just stipulate that the variable is fresh. The same applies to `toAINF`.

When discussing CSE in the paper, we describe renamings. In the `CSE.lean` file, the `rename` functions define how a renaming is actually applied. CSE also requires us to check equality of expressions, which is done with the `beq` functions. The CSE function in the paper also calls `lookup`, which is not defined there. It corresponds to the built-in `ListMap.lookup`.

Not mentioned in the paper are the pretty-printing routines `pretty` and `toString` in `Syntax.lean` and the functions for generating Lean code from A_iNF in `Codegen.lean`.

Our code also contains a function `Env.or`, which merges two environments. This is used to allow CSE to remove redundancies which appear in different, but compatible, environments. In the paper, this is omitted.

2 Content

The artifact package includes:

- Type: Artifact Description
  Format: Markdown and PDF
  Location: `README.md` and `README.pdf`
- Type: Docker Image with Dependencies
  Format: Docker Image
  Location: `docker-image.tar`
- Type: Command Line Interface
  Format: Shell Script
  Location: `lake.sh`
- Type: Source code
  Format: Lean source code
  Location: `src/*`

3 Getting the artifact

The artifact endorsed by the Artifact Evaluation Committee is available free of charge on the Dagstuhl Research Online Publication Server (DROPS). In addition, the artifact is also available at: `https://github.com/stg-tud/ainf-compiling-with-arrays`.

4 Tested platforms

We expect the following tools to be preinstalled:

- a `.tar.gz` extraction tool,
- docker or podman,
- a shell (tested with bash),
- a text editor.

We believe no special hardware is required and any modern laptop or computer is sufficient. We tested the artifact:

- using an i5 (4x 3GHz) CPU 8GB RAM, Linux
- using an i7 (12x 4GHz) CPU 32GB RAM, Linux
- no GPU is required
- 3GB space for docker container and extracted content should suffice

Compilation and running the artifact should take less than 5 min.

5 License

The artifact is available under Apache 2.0 License.

6 MD5 sum of the artifact

69424be8c55101b3294422fda28c8eba

7 Size of the artifact

324.23 MB

A Quick-Start Guide

1. Load the Docker image. Depending on whether you are using Docker or Podman run:
   - `docker load -i docker-image.tar` or
   - `podman load -i docker-image.tar`
2. The Docker image contains an instance of the Lean4 compiler and the lake build system. We provide a thin wrapper for invoking the docker container as if it was a local lake instance, using the `lake.sh` file.
   Now, you can compile, check proofs, and run the tests:
   - `sh lake.sh exe polara` (*check proofs*)
   - `sh lake.sh exe test` (*run tests*)
   This will run the Lean compiler on the provided sources. Verify that no errors appear. The expected results are described in further detail below.

(Remark: Depending on how you installed and configured docker, you might need to execute `sudo docker load -i` and the docker command in the `sudo lake.sh` instead. Podman does not require `sudo`.)

B Expected Behavior (Functional Badge)

To verify the functional badge, perform the following steps:

1) Run `sh lake.sh exe polara`, the output should look like this:

```
info: [0/12] Building Polara.Syntax
info: [1/12] Compiling Polara.Syntax
info: [1/12] Building Polara.Codegen
info: [1/12] Building Polara.NbE
info: [1/12] Building Polara.CSE
info: [3/12] Compiling Polara.Codegen
info: [5/12] Compiling Polara.NbE
info: [7/12] Compiling Polara.CSE
info: [7/12] Building Polara
info: [8/12] Compiling Polara
info: [8/12] Building Main
info: stdout:
'Tm.toAINF' does not depend on any axioms
'AINF.cse' depends on axioms: [Classical.choice, Quot.sound, propext]
'Tm.norm' depends on axioms: [Classical.choice, Quot.sound, propext]
info: [10/12] Compiling Main
info: [12/12] Linking polara
Success!
```

What does this mean? At the end of the `src/Main.lean` file we print out all assumptions and axioms used in the development using:

```
#print axioms Tm.toAINF
#print axioms AINF.cse
#print axioms Tm.norm
```

You can verify that the only axioms we use are `Classical.choice`, `Quot.sound` and `propext`. These axioms are built into Lean and almost unavoidable. You can verify that these are exactly the axioms used, e.g., we did not "cheat" by using additional axioms.
Also, you could verify that all functions defined in the source files do not use `noncomputable def`, or `sorry`. Functions which are not marked as `noncomputable` are computable, meaning the compiler generates machine code for them. `sorry` is an escape hatch to avoid having to define a function or proof. We believe that usage of `noncomputable` or `sorry` would show up as warnings during compilation.

2) Run `sh lake.sh exe test` to run the tests, the output should list the tests, all of which succeed, e.g:

```
Running tests
* OK ...
* OK ...
...
* === 16 / 16 tests passed ===
```

C Re-Use Scenario (Reusable Badge)

We give an example of how the artifact can be reused on additional programs. We describe a simple change to demonstrate how the case studies can be modified and how the new code can be compiled:

Besides the commands above, you can also

- compile and check proofs: `sh lake.sh exe polara`
- compile and run tests: `sh lake.sh exe tests`

To demonstrate that our code is not set in stone, e.g., can be modified and reused, you may try following one or more of the following usage scenarios.

1) Open the file `src/Polara/Test.lean`. This file contains tests that print `* OK ...` on success and `* ERROR ...` on failure.
You can duplicate one of the tests, for example, the file contains a test `codegen` that creates an expression using the function `egypt`, transforms it to A_iNF, then generates Lean code from A_iNF, and finally evaluates the Lean code.
In the test, try setting `base` and `expo` to other values. This will generate different code producing different outputs, but the result should still be equal to the reference output from `egyptLean` (in `Examples.lean`) and the test should still pass.
2) Open the file `src/Polara/Examples.lean`. This file contains the functions that are tested by the file above.
In Polara, an expression is either
   - a constant natural `.cst0 (.litn n)`
   - a constant floating point `.cst0 (.litf f)`
   - a constant index `.cst0 (liti i)`
   - an operator, for example `cst2 addn a b` or `cst2 app f x`
   - an array construction `bld fun i : Gamma (idx n) => e`. (The array construction constructs an array of length `n` by repeatedly evaluating `e`, with `i` bound to the values from `0` to `n-1`. For example, `bld fun i => cst2 muln 10 (i2n i)` evaluates to `#[0, 10, 20]`.)
   - for more info see the `inductive Tm` in the file `src/Polara/Syntax.lean`

The `egyptLean` function is defined as a Lean function calculating `base ^ (2 ^ n)`. The egypt function is defined as a Polara function that calculates the same result, but does so using multiplication instead of powers. This is a good example of how partial evaluation and common subexpression elimination together can optimize a function well:

```
def egyptLean (n: Nat) (x: Nat) :=
  n ^ (2 ^ x)

def egypt (n: Nat) {Gamma : Ty → Type} : Tm Gamma (Ty.nat ~> Ty.nat) :=
  let rec foo' (x : Gamma Ty.nat) : Nat → Tm Gamma Ty.nat
  | 0   => var x
  | n+1 => cst2 app (abs fun y => cst2 muln (var y) (var y)) (foo' x n)
  abs fun x => foo' x n
```

To simulate writing your own Polara function, you could perform the following steps. We define a new variant of `egyptLean` that performs multiplication instead of powering, and an equivalent variant of the `egypt` function that performs repeated addition. For this replace the first of the power functions in `egypt` by multiplication, and the `muln` operator in `egypt` by `addn` (in `Examples.lean`):

```
def egyptLean2 (n: Nat) (x: Nat) :=
  n * (2 ^ x)

def egypt2 (n: Nat) {Gamma : Ty → Type} : Tm Gamma (Ty.nat ~> Ty.nat) :=
  let rec foo' (x : Gamma Ty.nat) : Nat → Tm Gamma Ty.nat
  | 0   => var x
  | n+1 => cst2 app (abs fun y => cst2 addn (var y) (var y)) (foo' x n)
  abs fun x => foo' x n
```

To test this, we can also duplicate the test function, rewritten such that it invokes `egypt2` and `egyptLean2` instead, but still compares the two functions for equality (in `Test.lean`, within the `Test` namespace):

```
def codegen2 : IO Unit := group "codegen" <| do
  let base := 3
  let expo := 5

  let e1 := (Tm.norm (fun _ => Tm.cst2 Const2.app
    (egypt2 base) (.cst0 (.litn expo)))).toAINF.cse [] [] |>.codegen id
  let e2 := (Tm.norm (fun _ => (egypt2 base))).toAINF.cse [] [] |>.codegen
    fun x => x ++ s!" {expo}"
  let actual1 ← evalStr e1
  let actual2 ← evalStr e2
  let expected := egyptLean2 expo base
  assertEq "egypt2: generated code evaluates correctly"
    actual1.trim s!"{expected}"
  assertEq "egypt2: generated code evaluates correctly"
    actual2.trim s!"ok: {expected}"
```

Do not forget to add the test to the main function in `Test.lean`:

```
def main := do
  IO.println ""
  IO.println "Running tests"
  Test.codegen
  Test.codegen2 -- new!
  Test.toAINF
  Test.CSE
  let total := (← success.get).size + (← failure.get).size
  IO.println s!"=== {(← success.get).size} / {total} tests passed"
```

You can now check that an additional test runs successfully, using `sh lake.sh exe test`.

3) The implementation of conversion to A_iNF, i.e., fission, is clean and simple in a few lines. We envision a usage scenario of this artifact is its purpose as a tutorial reference for people who would like to reproduce it.