Compiling with Arrays

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/LIPIcs.ECOOP.2024.33

Related Version *Full Version*: `https://arxiv.org/abs/2405.18242`

Supplementary Material *Software (ECOOP 2024 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.10.2.18`
Software: `https://github.com/stg-tud/ainf-compiling-with-arrays` [32]
archived at `swh:1:dir:8e0e755d11e4e3e91fb05bf8df1a5c8bec0f553a`

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).

1 Introduction

Linear algebra computations are of rising importance due to their foundational role in neural networks and other machine learning systems. The fundamental unit of computation in linear algebra is the multidimensional array (or just *array* from now on). Linear algebra programs are full of computations that construct arrays from other arrays, such as element-wise sum, matrix multiplication, convolution, (transformer) attention, and more.

In functional programming we usually work with lists not arrays. Lists are inductively defined data types, and processed using recursion. Even element access on lists is implemented by recursion, so it has to traverse the list until the element is found, giving it running time[1] linear in the size of the list. Arrays, on the other hand, should have constant-time access and be processed using bulk operations. As such arrays do not fit into the usual pattern of inductive data types. A number of functional programming languages that aim to facilitate programming of array computations have been proposed [12, 37, 31]. Aiming for high expressivity with few language constructs, they leverage the idea that arrays can be represented as functions [12, 37, 31], or that arrays are eager functions [28].

Functions are lazy in the sense that a function definition does not perform any computation and the function body is only executed when a function is applied. Arrays are eager in the sense that all contents of an array are evaluated during construction, and array access does not perform further computation. Prior work had this intuition on the duality between arrays and functions, and here we ground that correspondence on proof-theoretic concepts, which explain why arrays are eager and functions are lazy and yield further convenient consequences.

A key insight of our work is that arrays can be interpreted as a positively polarized version of the function type. The connection between the positive and negative formulation of data types and the computational interpretation of elimination forms as pattern matching has been developed in the context of polarized type theory and focused logic [39, 1, 9, 10, 8, 19, 40]. Thus, similarly to how pattern matching on a binary tuple introduces two variables, the computational content of pattern matching on an array of size N is the introduction of N-many variables. Further, we explore pattern matching on Booleans by introducing *conditional variables*. This insight provides the foundation for the design of the *indexed administrative normal form* ($\mathrm{A}_i\mathrm{NF}$), an intermediate representation for array computations. We also present a simple surface array language, called POLARA, and show how these changes together (the positive formulation for the array type and a negative presentation of Booleans) enables us to combine *typed partial evaluation* (a.k.a *normalization by evaluation*) and *common subexpression eliminiation* into an elegant optimization algorithm overcoming some challenges usually associated to partial evaluation.

In particular, partial evaluation of let-binding is not *safe* in the sense of always resulting in a better, or at least equal, performance than the original. This is because a variable may appear multiple times, hence substituting it multiple times would duplicate code. Ideally, common subexpression elimination (CSE) would remove redundancies introduced by partial evaluation. But the presence of scopes, as e.g., introduced by functions, loops, and branches – all constructs that prevail in array computations – can complicate CSE. While compilers can rectify these issues by using additional rules often summarized under the general term of *code motion*, this comes at the cost of having to decide in what order and how often to apply these additional rules, i.e., it implies creating an optimization schedule, which complicates the algorithm.

[1] We distinguish run-time (as in run-time library) from running time as the time it takes to run something.

```
let z =
  let y1 = x + 1
  2 * y1


let y2 = x + 1
...
```

(a) Nested lets.

```

let y1 = x + 1
let z = 2 * y1


let y2 = x + 1
...
```

(b) Flat let-bindings.

```
let f = fun i:nat.
  let y1 = x + 1
  2 * y1


let y2 = x + 1
...
```

(c) Functions.

```
let z = if c then
  let y1 = x + 1
  2 * y1
else
  2 * x
let y2 = x + 1
...
```

(d) Branches.

Figure 1 Sample programs.

Instead of these complications, with A$_i$NF, we propose a novel intermediate representation for array programs based on logical foundations that avoids the complexity of optimization schedules. Like ANF, which is normal with regard to commuting conversion of let-bindings implying maximal flatness, A$_i$NF is normal with regard to commuting conversions of for-loops, thus enabling what we call maximal loop fission. Maximal loop fission is a fundamental property to enable further optimizations such as dead code elimination or common subexpression elimination. We provide a translation of Polara into A$_i$NF, which performs maximal loop fission and loop invariant code motion.

Contributions. In summary, this paper makes the following contributions:

- We present A$_i$NF, an intermediate representation that makes use of the unconventional idea of treating arrays as positive types from polarized type theory. A$_i$NF is normal with regard to commuting conversion for both let-bindings and for-loops, leading to flat and maximally fissioned terms.
- We present Polara, a simple surface array language, along with a translation of Polara to A$_i$NF, for which we prove termination, type preservation, and maximal fission.
- We present an optimization algorithm for A$_i$NF based on normalization by evaluation and common subexpression elimination, for which we prove termination and type preservation.

2 Problem Statement

Typed partial evaluation is a powerful optimization technique [17], which can reduce excessive terms by applying computation laws. For example, it can reduce a projection on a pair $(a, b).1 \equiv a$ by applying β-reduction. Or, it can eliminate superfluous branches like in $\textsf{if } x \textsf{ then } (\textsf{if } x \textsf{ then } a \textsf{ else } b) \textsf{ else } c \equiv \textsf{if } x \textsf{ then } a \textsf{ else } c$ by applying uniqueness laws (e.g., η-expansion).

But, while shining on its logical foundation, partial evaluation is not a *safe* optimization. A *safe* optimization has to either reduce the running time of a program or at least preserve it. Partial evaluation of let-bindings is not *safe* because a variable may appear multiple times, hence substituting it multiple times would duplicate code. Ideally, common subexpression elimination (CSE) would remove all redundancies introduced by partial evaluation. But scopes introduced by nested let-bindings, functions, and branches complicate CSE. For illustration, below we consider a few examples of redundancies that can occur in programs and how CSE handles them.

In Figure 1a, a program with nested let-bindings is shown. Here, the variable `z` is bound to `2 * y1`, where `y1` is bound to the successor of `x`; and then the variable `y2` is bound to the successor of `x` as well. It is easy to see that `y1` is redundant with `y2`, yet `y1` is not in scope at the definition of `y2`, so we cannot simply replace one by the other. The problem can be avoided by bringing the program into a form, where no let-binding is nested inside another let-binding, such that all previously bound variables are in scope for the whole remaining expression. Consider the program shown in Figure 1b, which is equivalent to the previous program, but this time no expression has a subexpression. Now, the former definition is in scope at the latter definition, and thus `y2` can be replaced by `y1`, thereby eliminating a duplicate subexpression.

As mentioned, functions and branches introduce scope as well, and therefore complicate CSE. Yet, the solution of flattening the code is not as straightforward to apply. To illustrate the problem with functions, consider the program shown in Figure 1c, which defines a function `f`. Inside the function the successor of `x` is bound to `y1`, and outside the function it is bound redundantly to `y2`. To share the expressions, we could consider moving the definition of `y1` out of the functions. But moving an expression out of a function is not *safe*, as long as we do not know whether the function will be called at all.

To illustrate the problem with branches, consider the program shown in Figure 1d. Here, the result of a conditional expression is bound to the variable `z`, in one branch the successor of `x` is bound to `y1`, and after the conditional expression the successor of `x` is bound to the variable `y2`. Similar to the function case, to share the expressions, we could consider moving the definition of `y1` out of the branch, but that is again not a *safe* optimization, as long as we do not know that this branch is taken.

To rectify the issues outlined above compilers use additional rules often summarized under the general term of *code motion*. But this comes at the cost of introducing the problem of having to decide in what order and how often to apply these additional rules (i.e., creating an optimization schedule).

Our work avoids the complexity of optimization schedules. We argue that – instead of complicating the CSE algorithm with optimization schedules – a better approach is to design an intermediate representation for array programs, which like ANF bans nested expressions. This simplifies the optimization of array programs, which now can safely rely on algorithms based on logical foundations such as partial evaluation and CSE. The novel intermediate representation, called A$_i$NF, is informally presented in the following along with a simple surface arrays language and the optimized translation of the latter to the former.

3 A$_i$NF, Polara, and Simplified Optimizations

We describe the two key insights on which our approach is based (Sections 3.1 and 3.2), introduce Polara and A$_i$NF by example (Section 3.3), and explain how A$_i$NF simplifies optimizations (Section 3.4).

3.1 The Duality of Functions and Arrays

The list type is an inductive datatype defined by its constructors `nil` for the empty list and `cons` for constructing a list from another list with an additional element. Accordingly, algorithms over lists work by recursion, expressed with functions and branches. As element access on lists is implemented by recursion, it has to traverse the list until the element is found, giving it running time linear in the size of the list. Arrays, on the other hand, enjoy constant-time access and feature bulk operations. The consequence is that arrays do not to fit into the usual pattern of inductive data types. Nevertheless, because custom semantics would require further proofs to ensure soundness, arrays are occasionally modelled as lists, with the hint that the actual running time can differ (in Lean for example[2]).

[2] `https://lean-lang.org/lean4/doc/array.html`

In functional array languages, we exploit the equivalence of an array of type X and length n with a function from a natural number below n to a value of type X. Forward, this equivalence allow us to access (`get`) elements of an array by its index. Backward, we create (`tabulate`) an array from a function describing each individual element based on their index. The forward direction is indeed already very much ingrained in everyday programming, as array access `a[i]` and function application `a(i)` look very much alike in many languages, and even share identical syntax in some.

$$\text{Array}_n\ X \ \leftrightarrow\ (\text{Fin}_n \to X)$$

$$\begin{aligned} \text{get} &: \text{Array}_n\ X \to (\text{Fin}_n \to X) \\ \text{tabulate} &: (\text{Fin}_n \to X) \to \text{Array}_n\ X \end{aligned}$$

But something important changes in the conversion from a function to an array, and vice versa. Functions are *lazy*, in the sense that the evaluation of a function is delayed until it is applied, while arrays are *eager*, in the sense that all elements of an array have already been evaluated and on array access only need to be looked up. Also, in a language with (side) effects, the two types can be distinguished in that a function application can trigger effects, while an array access cannot trigger effects. Dually, constructing a function cannot trigger effects, while constructing an array can trigger effects.

We can put the relationship between functions and arrays on a logical foundation by considering the difference between positive and negative types [39]. A positive type is defined by a set of constructors (introduction forms), and we get a single corresponding destructor (elimination form) for it with one continuation for the content of each possible constructor (pattern matching). A negative type is defined by a set of destructors, and we get a single corresponding constructor for it that has to provide one value for each destructor to extract (copattern matching). Positive types are usually associated with eager (call-by-value) evaluation, and negative types with lazy (call-by-name) evaluation. Many types can be defined either as a positive or as a negative type. For illustration, we consider the positive and the negative formulations of the product type below.

Products as Positive and as Negative Types. The product type as a positive type $\times$ has a single constructor $(a,\ b)$ (Intro). A corresponding destructor (Elim) can be systematically derived as pattern matching on the constructor. Reduction (Beta) occurs when a destructor is applied to a constructor, and they eliminate each other.

$$\textsc{Intro}\ \frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash (a,\ b) : A \times B} \qquad \textsc{Elim}\ \frac{\Gamma \vdash p : A \times B \qquad \Gamma,\ a : A,\ b : B \vdash c : C}{\Gamma \vdash \mathsf{let}\ (a,\ b) = p;\ c : C}$$

$$\textsc{Beta}\ \frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B \qquad \Gamma,\ x : A,\ y : B \vdash c : C}{\Gamma \vdash (\mathsf{let}\ (x,\ y) = (a,\ b);\ c) \equiv c[x := a,\ y := b]}$$

Alternatively, products can also be defined as negatives types $\otimes$. In this case, we give primacy to a set of destructors, namely the projections $p.\mathsf{fst}$ and $p.\mathsf{snd}$ (Elim1, Elim) to access the individual elements of a tuple p, and derive systematically the corresponding constructor (Intro) providing one value for each destructor to extract. Beta reduction occurs (Beta1, Beta1) when a destructor is applied to a constructor by extracting the corresponding value.

$$\frac{\Gamma \vdash p : A \otimes B}{\Gamma \vdash p.\mathsf{fst} : A}\ \textsc{Elim1} \qquad \frac{\Gamma \vdash p : A \otimes B}{\Gamma \vdash p.\mathsf{snd} : B}\ \textsc{Elim2} \qquad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash (\mathsf{fst} = a;\ \mathsf{snd} = b) : A \otimes B}\ \textsc{Intro}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash (\mathsf{fst} = a, \mathsf{snd} = b).\mathsf{fst} = a}\ \textsc{Beta1} \qquad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash (\mathsf{fst} = a, \mathsf{snd} = b).\mathsf{snd} = b}\ \textsc{Beta2}$$

Functions as Negative and Positive Types. Usually, the function type is considered a negative type. It has a single destructor – function application $f\,a$ (Elim) – and the corresponding constructor is systematically derived by copattern matching on the possible destructors (Intro). When a destructor is applied to the constructor, we extract the value provided as the body of the function, and substitute the variable with the argument (Beta).

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B}\ \textsc{Elim} \qquad \frac{\Gamma, a : A \vdash b : B}{\Gamma \vdash \mathsf{fun}\ a.\ b : A \to B}\ \textsc{Intro} \qquad \frac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash a : A}{\Gamma \vdash (\mathsf{fun}\ x.\ b)\ a \equiv b[x := a]}\ \textsc{Beta}$$

The function type can also be represented as a positive type. In this case, the function is primarily defined through its constructor, and the destructor is systematically derived from pattern matching on the constructor. But the interpretation of positive function types comes with some challenges for the metatheory. The introduction form of a function turns a term-in-the-context-of-a-variable $a : A \vdash b : B$ into a function $(\mathsf{fun}\ a.\ b) : A \to B$. Thus, the corresponding elimination form of a function $(\mathsf{fun}\ a.\ b) : A \to B$ should introduce a variable of type term-in-the-context-of-a-variable $a : A \vdash b : B$ into the context. But to properly model that, we need a judgment where we have a context in the context, in other words a "higher-order judgment" [26, 25]. A judgment is higher-order when an entailment $\vdash$ occurs inside the context of another entailment An implementation of higher-order judgments needs to ensure that a variable which has such a judgment as a type is only used in larger contexts, where all required variables are available. For example, $b : (a : A \vdash B) \vdash b : B$ is invalid, given that b must occur in a context where an $a : A$ is available; while $b : (a : A \vdash B) \vdash (\mathsf{fun}\ a{:}A.\ b) : A \to B$ is valid, because a variable $a : A$ has been introduced such that the use of b afterwards is safe.

$$\frac{\Gamma, a : A \vdash b : B}{\Gamma \vdash (\mathsf{fun}\ a.\ b) : A \to B}\ \textsc{Intro} \qquad \frac{\Gamma \vdash f : A \to B \qquad \Gamma, x : (a : A \vdash B) \vdash c : C}{\Gamma \vdash \mathsf{let}\ (\mathsf{fun}\ a.\ x) = f;\ c : C}\ \textsc{Elim}$$

$$\frac{\Gamma, a : A \vdash b : B \qquad \Gamma, x : (a : A \vdash B) \vdash c : C}{\Gamma \vdash \mathsf{let}\ (\mathsf{fun}\ a.\ x) = (\mathsf{fun}\ a.\ b);\ c \equiv c[x := b]}\ \textsc{Beta}$$

Interpreting positive function types as arrays. We avoid the challenges of interpreting functions as positive types by proposing to interpret positive function types as arrays, re-interpreting the rules of the positive function type as the rules of the array type. We require the argument type to be the type of natural numbers below some number `n`, which corresponds to the index type of an array. Traditionally, functional programming works with lists and not arrays, therefore the constant-time access of arrays is not accurately represented by the model; in functional array languages [15, 37], arrays tend to live in the shadow of the function type, as their introduction and elimination forms depend on (higher-order) functions. Interpreting the array as a positive function type makes them independent and puts them on an equal footing to the other types with regard to their logical foundation.

We distinguish positive functions, i.e., arrays, from normal functions by using $\Rightarrow$ for the type, writing ($\mathsf{for}\ a.\ b$) as the introduction form for arrays, while the elimination form is given, as always, by pattern matching on all possible introduction forms:

$$\textsc{Intro}\quad \frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash (\mathsf{for}\ x.\ b) : A \Rightarrow B}\ A = \mathrm{Fin}_n \qquad \textsc{Elim}\quad \frac{\Gamma \vdash f : A \Rightarrow B \qquad \Gamma, x : (a : A \vdash B) \vdash c : C}{\Gamma \vdash \mathsf{let}\ (\mathsf{for}\ a.\ x) = f;\ c : C}\ A = \mathrm{Fin}_n$$

$$\textsc{Beta}\quad \frac{\Gamma, a : A \vdash b : B \qquad \Gamma, x : (a : A \vdash B) \vdash c : C}{\Gamma \vdash \mathsf{let}\ (\mathsf{for}\ a.\ x) = (\mathsf{for}\ a.\ b);\ c \equiv c[x := b]}$$

Intuitively, analogously to how pattern matching on a product introduces two variables (one for each projection of the product), pattern matching on an array introduces a family of variables, one for each element. For illustration, consider $b[a := 2]$ as b_2, and ($\mathsf{let}\ (\mathsf{for}\ a.\ b) = f;\ c$) as ($\mathsf{let}\ (b_0, b_1, ..., b_{n-1}) = f;\ c$).

Arrays Enable CSE. Flattening let-bindings usually helps CSE. More precisely the rule that is used to create the ANF representation is the let-let commuting conversion:

```
(let y = (let x = e1; e2); e3) ≡
(let x = e1; let y = e2; e3)
```

Note that the following let-fun commuting conversion is not *safe* because on the left-hand side `e1` is evaluated at most once, even if it was used multiple times in `e2`; but on the right-hand side it will be evaluated once for each usage in `e2`.

```
(let y = (fun i. let x = e1; e2); e3) ≡
(let (fun i. x) = (fun i. e1); let y = (fun i. e2); e3)
```

On the other hand, the let-for commuting conversion that we use in A$_i$NF below is *safe* and states that the following two lines are equivalent. As array construction is evaluated eagerly, the expression `e1` is evaluated just once for each iteration, on both sides of the equation.

```
(let y = (for i. let x = e1; e2); e3) ≡
(let (for i. x) = (for i. e1); let y = (for i. e2); e3)
```

Intuitively, this rule allows us to split a complex loop into multiple simpler loops, hence it is closely connected to loop *fission*. If we use this rule to split every loop as much as possible, then we end up with a normal form in which every loop only contains a single operation. First performing loop fission as much as possible helps with implementing other optimizations, for example allows CSE to remove redundancies that it could not otherwise eliminate.

The use of this rule means that frequently both the left side and the right side of a variable definition are surrounded by the same form (on the left as a pattern form, on the right as a term form), so we will introduce some syntactic sugar and write `let for i. (x = e1); e2` to mean `let (for i. x) = (for i. e1); e2` in the following.

3.2 Lifting Branching into the Context

A different problem arises with values of the Boolean type, Booleans have two constructors, true and false (Intro1, Intro2). They have one destructor, the conditional expression (Elim), where one continuation is provided for each constructor, the consequent and the alternative. A conditional reduces to the consequent when the condition is true (Beta1), and to the alternative when the condition is false (Beta2).

$$\textsc{Intro1}\quad \frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool}} \qquad \textsc{Intro2}\quad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool}} \qquad \textsc{Elim}\quad \frac{\Gamma \vdash p : \mathsf{bool} \qquad \Gamma \vdash e : C \qquad \Gamma \vdash f : C}{\Gamma \vdash \mathsf{if}\ p\ \mathsf{then}\ e\ \mathsf{else}\ f}$$

$$\textsc{Beta1}\quad \frac{\Gamma \vdash e : C \qquad \Gamma \vdash f : C}{\Gamma \vdash \mathsf{if\ true\ then}\ e\ \mathsf{else}\ f \equiv e} \qquad \textsc{Beta2}\quad \frac{\Gamma \vdash e : C \qquad \Gamma \vdash f : C}{\Gamma \vdash \mathsf{if\ false\ then}\ e\ \mathsf{else}\ f \equiv f}$$

The let-if commuting conversion below is *safe* with regard to running time. But applying the commuting conversion duplicates the expression `e3`. If `e3` is a big expression, then even if duplicating it in different branches may not impact the running time, having nested branches will lead to a blow-up of the code size exponential in the number of branches (which is also bad for the compiling time).

```
(let z = (if e0 then e1 else e2); e3) ≡
(if e0 then (let z=e1; e3) else (let z=e2; e3))
```

Essentially, the above rule bubbles up conditionals to the top of the expression. Instead, we propose to trickle down the conditionals using conditionally defined variables and state a new `let-if` commuting conversion that does not duplicate the branches. To avoid duplicating branches, we introduce a syntactically single-branch `if`. A single-branch `if` produces a conditional value, i.e., a value that can only be accessed if the condition is true, and the single-branch `if!` produces a value that can only be accessed if the condition is false.

Analogously, we have let-bindings for conditional variables `let (if e. x) :=` ... (or `let (if! e. x) :=` ...), which define a variable `x` that can only be accessed if the condition `e` is true (or false, respectively). Two simple syntactical conditions can be used to check whether a conditional variable is accessible: First, a conditional variable is accessible on the right-hand side of the definition of another conditional variable that has the same condition. In other words, a conditional variable can be used to define the value of another conditional variable with the same condition. Second, a conditional variable is accessible in one of the branches of a standard two-branched `if` condition. In other words, two mutually exclusive conditional variables can be combined with an `if` to define a non-conditional variable.

Using the single-branch if, we can now express a `let-if` commuting conversion, that does duplicate `e3`. Here the double-branched if is seperated into two single-branch ifs and `e3` remains to be executed once afterwards:

```
(let z = (if e0 then e1 else e2);
 e3)
≡
(let (if  e0. z1) = (if e0. e1);
 let (if! e0. z2) = (if! e0. e2);
 let          z   = (if e0 then z1 else z2);
 e3)
```

Table 1 Common linear algebra operations in POLARA; NumPy for reference.

Name	NumPy	POLARA
Vector addition	`v + w`	`for i. v[i] + w[i]`
Matrix addition	`A + B`	`for i j. A[i,j] + B[i,j]`
Element-wise product (vector)	`v * w`	`for i. v[i] * w[i]`
Element-wise product (matrix)	`A * B`	`for i j. A[i,j] * B[i,j]`
Outer product	`np.multiply.outer(A, B)`	`for i j k l. A[i,j] * B[k,l]`
Trace	`A.trace()`	`sum i. A[i, i]`
Transpose	`A.transpose()`	`for i j. A[j, i]`
Matrix multiplication	`A @ B`	`for i k. sum j. A[i,j] * A[j,k]`
Matrix-vector multiplication	`A @ v`	`for i. sum j. A[i,j] * v[j]`

Correct use of conditional values will thus frequently lead to the use of the same condition on the variable bound by a `let` and in a single-branched if in the bound expression. We will thus make use of syntactic sugar writing `let if e0. (x1 = e1); e2` to mean `let (if e0. x1) = (if e0. e1); e2`.

3.3 Polara and A$_i$NF by Example

In this section, we informally introduce both POLARA and A$_i$NF. We do so by giving examples of array operations and programs in POLARA and showing the result of compiling them to A$_i$NF. Just for reference, we will also provide versions of the POLARA examples written in the widely-adopted array programming library NumPy [14]. Please note that the focus of this paper lies in the exploration of A$_i$NF, rather than POLARA. The latter serves merely as a vehicle to elucidate how A$_i$NF effectively facilitates optimizations during the translation process from a surface array language. Hence, compared to NumPy, we have purposely kept it closer to low-level imperative code.

In POLARA, an expression `e` is either a constant `c`, or an arithmetic operator `e + e`, function application `e e`, array access `a[i]`, array construction `for i:n. e`, or summation `sum i:n. e`, as well as pairs `(e, e)` and projection `e.1`, `e.2`. The array construction `for i:n. e` constructs an array of length `n` by repeatedly evaluating `e`, with `i` bound to the values from `0` to `n-1`. For example, `for i:3. 10*i` evaluates to `[0, 10, 20]`. We will write `for i. e` if the size of the array can be inferred from the context. Summation is syntactic sugar for constructing an array and then summing it, so `sum i.e` ≡ `sum (for i. e)`.

In Table 1, we list several common linear algebra operations and compare how they can be expressed using the linear algebra library NumPy and POLARA. We assume as given that the vectors `v, w` and matrices `A, B` are of appropriate sizes.

Dense Layer. As a slightly more involved example, we show how a dense neural network layer can be implemented in NumPy, POLARA, and A$_i$NF, respectively. The NumPy example makes use of the built-in matrix multiplication operator `@`. While such an operation can be implemented as function in POLARA, we show an example that only relies on the few POLARA primitives, using the `for` looping construct and indexing. Likewise, while the NumPy definition uses the built-in `maximum` function and addition, the POLARA version uses an explicit loop that performs element-wise multiplication and additions across the vectors.

Compared to the untyped NumPy program, we also declare the types of the arguments. A type `n⇒flt` describes an array of floating point numbers with size `n`.

Obviously, the corresponding A$_i$NF program (Figure 2a) is rather lengthy, as every intermediate result gets assigned to a variable, just like in ANF.

```
def dense(b, W, x):
  return np.maximum(0, W @ x + b)
                                        NumPy
```

```
dense(b:n⇒flt, W:n⇒m⇒flt, x:m⇒flt): n⇒flt :=
  for i. max(0, (sum j. W[i][j] * x[j]) + b[i])
                                        Polara
```

Convolution. We now describe how to express convolution in Polara. Convolution involves moving a vector, called the kernel, across another vector while repeatedly calculating the dot product. For this example, we need to subtract two indices to indicate that we shift one array while keeping the other as it is. In the A$_i$NF example (Figure 2b), we create a two-dimensional array `x10` containing all the possibilities for shifting the array `y`. For example if `y = [1,2,3]`, then `x10` is a matrix of size 3×3 so that `tmp1 = [[1,2,3], [2,3,1], [3,1,2]]`. We then form the dot product of each entry with `x`.

```
def conv(x, y):
  return np.convolve(x, y, 'same')
                                        NumPy
```

```
conv(x: n⇒flt, y: (n+m-1)⇒flt): m⇒flt :=
  for i. sum j. x[j] * y[j+i]
                                        Polara
```

Black-Scholes. Black-Scholes is a simplified mathematical model for the dynamics of derivative investments in financial markets. The Black-Scholes formula provides an estimate for the price of the call option (buying) and the put option (selling) of a European-style option given the original price S, the strike price K, the expiration time T, the force-of-risk r and the standard deviation σ. The interesting part, from an array programming language's perspective, is that with a naive implementation of the calls and puts as separate functions, common subexpression elimination is not able to identify the redundant computation across these functions over two separate loops.

In particular, note the redundant definition of `d1` and `d2` in the `calls` and the `puts` function. This code gets inlined into the `blackScholes` function, but the two function calls land in separate loops. Nevertheless, using loop fission the output can be reduced to just 22 lines of A$_i$NF (see Figure 2c); without fission and CSE the generated code would have 54 lines.

```
calls(S: flt, K: flt, T: flt: r: flt: sigma: flt): flt :=
  let d1 := (log (S / K) + (r + sigma * sigma / 2) * T) / (sigma * sqrt T)
  let d2 := d1 - sigma * sqrt T
  S * normCdf d1 - K * exp (0 - var r * var T) * normCdf d2

puts(S: flt, K: flt, T: flt: r: flt: sigma: flt): flt :=
  let d1 := (log (S / K) + (r + sigma * sigma / 2) * T) / (sigma * sqrt T)
  let d2 := d1 - sigma * sqrt T
  K * exp (0 - r * T) * normCdf (0 - d2) - S * normCdf (0 - d1)

blackScholes(arr: (n ⇒ flt)): n ⇒ (flt × flt) :=
  let S := 1; let K := 1; let r := 1; let sigma := 1
  let Calls: (n ⇒ flt) := for i. calls(S, K, arr[i], r, sigma)
  let Puts:  (n ⇒ flt) := for i. puts(S,  K, arr[i], r, sigma)
  for i. (Calls[i], Puts[i])
                                                              Polara
```

3.4 Simplifying Optimizations with A$_i$NF

This section provides an overview showing how some classical optimizations can be applied to A$_i$NF and illustrates why our novel normal form simplifies their implementation.

To improve readability, we will sometimes present A$_i$NF code in a way that deviates from the actual representation by putting multiple operations in one line, when this does not affect the optimization.

```
dense(b: n ⇒ flt, W: n ⇒ m ⇒ flt,
      x: m ⇒ flt): n ⇒ flt :=
let for i:n,      (x0 : flt      := 0          )
let for i:n, j:m, (x1 : m ⇒ flt := W[i]       )
let for i:n, j:m, (x2 : flt      := x1[j]      )
let for i:n, j:m, (x3 : flt      := x[j]       )
let for i:n, j:m, (x4 : flt      := x2 * x3    )
let for i:n,      (x5 : m ⇒ flt := for j:m, x4)
let for i:n,      (x6 : flt      := sum x5     )
let for i:n,      (x7 : flt      := b[i]       )
let for i:n,      (x8 : flt      := x6 + x7    )
let for i:n,      (x9 : flt      := max x0 x8  )
let               (x10: n ⇒ flt := for i:n, x9)
x10
                                            AᵢNF
```

(a) A_iNF for a dense layer.

```
conv(x: n ⇒ flt, y: p ⇒ flt): m ⇒ flt :=
let for i:m, j:n, (x0 : flt      := x[j]       )
let for i:m, j:n, (x1 : fin p    := j + i      )
let for i:m, j:n, (x2 : flt      := y[x1]      )
let for i:m, j:n, (x3 : flt      := x0 * x2    )
let for i:m,      (x4 : n ⇒ flt := for j:n, x3)
let for i:m,      (x5 : flt      := sum x4     )
let               (x6 : m ⇒ flt := for i:m. x5)
x6

  where p = n+m-1
                                            AᵢNF
```

(b) A_iNF for convolution.

```
blackScholes(arr: n ⇒ flt):  n ⇒ flt × flt :=
let for i1:n, (x0  : flt := 1.500000          )
let for i1:n, (x1  : flt := i0[i1]            )
let for i1:n, (x2  : flt := x0 * x1           )
let for i1:n, (x4  : flt := sqrt x1           )
let for i1:n, (x5  : flt := x2 / x4           )
let for i1:n, (x6  : flt := normCdf x5        )
let for i1:n, (x7  : flt := 0.000000          )
let for i1:n, (x9  : flt := x7  - x1          )
let for i1:n, (x10 : flt := exp x9            )
let for i1:n, (x19 : flt := x5  - x4          )
let for i1:n, (x20 : flt := normCdf x19       )
let for i1:n, (x21 : flt := x10 * x20         )
let for i1:n, (x22 : flt := x6  - x21         )
let for i1:n, (x37 : flt := x7  - x19         )
let for i1:n, (x38 : flt := normCdf x37       )
let for i1:n, (x39 : flt := x10 * x38         )
let for i1:n, (x47 : flt := x7  - x5          )
let for i1:n, (x48 : flt := normCdf x47       )
let for i1:n, (x49 : flt := x39 - x48         )
let for i1:n, (x50 : (flt × flt) := (x22, x49))
let (x51 : (n ⇒ flt × flt) := for i1:1, x50  )
x51
                                            AᵢNF
```

(c) A_iNF for a Black-Scholes.

Figure 2 Generated A_iNF.

Loop Fission. Loop fission is an optimization pass that prepares code to improve the effectiveness of dead code elimination. Standard dead code elimination can only delete whole loops. Loop fission splits a loop into parts, so that individual parts that are not used can be removed. In A_iNF, the body of each loop is a single operation, which means that any A_iNF program is necessarily as fissioned as possible. A simple partial evaluation pass on A_iNF can then perform dead-code elimination.

Below, the left side shows an array computation in Polara. On the right, that same computation has been transformed to A_iNF, which implies loop fission. The code follows the principle from ANF that expressions should be atomic, i.e., only have one operation. In terms of array programming, this leads to the first loop being split into three loops. Partial evaluation could then reduce the full program to just `for i. f(xs[i])`.

```
let x = for i.
  let ys = f(xs[i])
  let zs = f(xs[i])
  (ys, zs)
for i.
  fst x[i]
                                   POLARA
```

```
let for i. (ys = f(xs[i]))
let for i. (zs = f(xs[i]))
let for i. (x  = (ys, zs))
let for i. (y  = fst x)
let z = for i. y
z
                                   AᵢNF
```

A further advantage of loop fission is that it improves loop *fusion*: Splitting a program into as many loops as possible, gives more freedom to the algorithm for combining loops again.

$$n \in \mathbb{N} \qquad f \in \mathbb{F} \qquad x \in \mathrm{Var} \qquad i \in \mathrm{Idx}$$

$$\begin{array}{llcl}
\text{Types} & t & ::= & \mathsf{fin}\ n \mid \mathsf{flt} \mid t \mathbin{\hat{\times}} t \mid t \mathbin{\hat{\to}} t \mid n \mathbin{\hat{\Rightarrow}} t \\
\text{Constants} & c & ::= & n \mid f \mid \hat{+} \mid \hat{\cdot} \mid \hat{-} \mid \hat{/} \mid \mathsf{app} \mid \mathsf{get} \mid \mathsf{pair} \mid \mathsf{fst} \mid \mathsf{snd} \mid \mathsf{sum} \\
\textsc{Polara} & e & ::= & c\ \overline{e} \mid x \mid \mathsf{fun}\ x{:}t.\ e \mid \mathsf{for}\ x{:}n.\ e \mid \mathsf{ite}\ e\ e\ e \mid \mathsf{let}\ e;\ e \\
 & & & \\
\text{A}_i\text{NF} & a & ::= & \mathsf{let}\ C[x = p];\ a \mid x \\
\text{Primitives} & p & ::= & c\ \overline{x} \mid i \mid \mathsf{fun}\ i{:}t.\ x \mid \mathsf{for}\ i{:}n.\ x \mid \mathsf{ite}\ x\ x\ x \\
\text{Scope Contexts} & C[\cdot] & ::= & \cdot \mid C[\mathsf{fun}\ i{:}t.\ \cdot] \mid C[\mathsf{for}\ i{:}n.\ \cdot] \mid C[\mathsf{if}\ x{\neq}0.\ \cdot] \mid C[\mathsf{if}\ x{=}0.\ \cdot]
\end{array}$$

Figure 3 Polara and A$_i$NF.

Common subexpression elimination. We can now see how A$_i$NF helps with CSE. On the left, we recapitulate the example from above; on the right, we can see the same program in A$_i$NF.

```
let f = for i.
  let y = x+1
  let y' = 2*y
  y'
let z = x+1
...
                              POLARA
```

```
let for i. (y  = x + 1)
let for i. (y' = 2 * y)
let for i. (f  = y')

let for i. (z  = x + 1)
...
                              AiNF
```

The loop computing `f` has been broken down into two loops. As a result, `z` is clearly redundant, as it performs the same computation in the same scope as `y`. Therefore, compared to array languages using higher-order functions, A$_i$NF allows us to use the simple, standard approach to CSE, and nonetheless remove redundancies between expressions inside and outside of loops.

Loop invariant code motion. Loop invariant code motion (LICM), which moves constants out of a loop, is another optimization that benefits from A$_i$NF. In A$_i$NF, this would correspond to dropping an unused index; hence, the implementation of LICM is very simple. On the left, we generate an array `ys`, in which every element is the constant `1`. We then compute an array `zs` that makes use of `ys`. Notice that the index `i` that is bound in the creation of `ys` is not used. We can therefore eliminate that loop, adjusting uses of `ys` accordingly from `ys[i:=j]` to `ys`, as seen on the right.

```
let for i. (ys = 1)
let zs = for i. f(xs[i], ys)
...
                              AiNF
```

```
let ys = 1
let zs = for i. f(xs[i], ys)
...
                              AiNF
```

4 Mechanization

We mechanized Polara, partial evaluation of Polara, A$_i$NF, the translation from Polara to A$_i$NF, and common subexpression elimination over A$_i$NF, using the dependently typed programming language Lean 4 [7].

4.1 Polara and Partial Evaluation

Polara. The Polara grammar uses the set of natural numbers, floating point numbers, variables and indices (Figure 3), but the distinction between variables and indices is only relevant for A$_i$NF. Types are floating point numbers, products, functions, and arrays, as well as bounded natural numbers, i.e. fin n is the type of numbers smaller than n. Constants are

$$\frac{n < m}{\vdash n : \mathsf{fin}\ m} \qquad \vdash f : \mathsf{flt} \qquad \vdash \mathsf{app} : (t_1 \hat{\to} t_2) \to t_1 \to t_2 \qquad \vdash \mathsf{get} : (n \hat{\Rightarrow} t_1) \to \mathsf{fin}\ n \to t_1$$

$$\vdash \mathsf{pair} : t_1 \to t_2 \to (t_1 \hat{\times} t_2) \qquad \vdash \mathsf{fst} : (t_1 \hat{\times} t_2) \to t_1 \qquad \vdash \mathsf{snd} : (t_1 \hat{\times} t_2) \to t_2$$

$$\vdash \hat{+} : \mathsf{fin}\ n \to \mathsf{fin}\ m \to \mathsf{fin}\ (n + m - 1) \qquad \vdash \hat{+} : \mathsf{flt} \to \mathsf{flt} \to \mathsf{flt} \qquad \vdash \mathsf{sum} : (n \hat{\Rightarrow} \mathsf{flt}) \to \mathsf{flt}$$

$$\overset{\textsc{Var}}{\frac{x{:}t \in \Gamma}{\Gamma \vdash x : t}} \qquad \overset{\textsc{Const}}{\frac{\vdash c : \overline{t_i} \to t' \quad \overline{\Gamma \vdash e_i : t_i}}{\Gamma \vdash c\ \overline{e_i} : t'}}$$

$$\overset{\textsc{Fun}}{\frac{\Gamma, x{:}t_1 \vdash e : t_2}{\Gamma \vdash \mathsf{fun}\ x{:}t_1.\ e : t_1 \to t_2}} \qquad \overset{\textsc{For}}{\frac{\Gamma, i{:}\mathsf{fin}\ n \vdash e_2 : t}{\Gamma \vdash \mathsf{for}\ i{:}n.\ e_2 : n \hat{\Rightarrow} t}}$$

$$\overset{\textsc{Let}}{\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma, x{:}t_1 \vdash e_2 : t_2}{\Gamma \vdash \mathsf{let}\ x = e_1;\ e_2 : t_2}} \qquad \overset{\textsc{Ite}}{\frac{\Gamma \vdash e_1 : \mathsf{fin}\ 2 \quad \Gamma \vdash e_2 : t \quad \Gamma \vdash e_3 : t}{\Gamma \vdash \mathsf{ite}\ e_1\ e_2\ e_3 : t}}$$

Figure 4 POLARA's type system.

natural number and floating point literals, arithmetic symbols, function application (app), array access (get), pair construction (pair), first and second projection (fst, snd), and array summation (sum). POLARA terms are variable access, n-ary constant application, function abstraction, array construction, branching (ite), and let-binding. We decided to put first-order syntax forms such as function application, array access, pairing, and the product projections into the constants, because they are all handled uniformly by the following algorithms, while the higher-order syntax forms, i.e., the ones that bind variables, such as function abstraction, array construction, branching, and let-binding are kept in the terms because they are all treated differently.

Intrinsic Types. The typing rules for POLARA are given in Figure 4. First, we give the types for constants. Note that we use the $\to$ symbol for the typing judgement of constants that take arguments. This is not to be confused with the type constructor $\hat{\to}$. The typing rules for variables (VAR), function abstractions (FUN), and `let`-bindings (LET) are standard. The rule CONST allows one to apply a constant to a number (possibly zero) of arguments. For example, as app has type $(t_1 \hat{\to} t_2) \to t_1 \to t_2$, the expression app e_1 e_2 has type t_2 when $e_1 : t_1 \hat{\to} t_2$ and $e_2 : t_1$. The FOR rule shows that constructing an array with for requires an expression of type nat for the size and another expression, which can use the (numerical) index i and whose type gives the element type of the array. The ITE rule states that the condition has to be of type nat and the two branches have to be of the same type (the condition is considered true if nonzero).

PHOAS. Our formal development uses parametric higher-order abstract syntax (PHOAS) [29, 5], allowing us to leverage the binders of the host language as binders for the guest language. Terms are parametrized by an abstract denotation of types Γ, and variables contain a value of that type. By using PHOAS, we can avoid certain technicalities relating to variable binding such as capture-avoiding substitution, thereby streamlining the implementation.

Static Size. As mentioned above, our array types have the form `n ⇒ a`, where `n` is the size of the array. The fact that the size of an array is always part of its static type, implies that the sizes of all arrays are known at compile time. This guarantees that indexing can be statically checked for out-of-bounds array accesses, ensuring the absence of run time errors without requiring run time checks.

Because Polara is not polymorphic, expressions operating on arrays are fixed to specific array sizes. For example, there is no single expression in Polara that can map a function over an array of *arbitrary* size. This restriction is alleviated because our language is embedded, allowing us to reuse polymorphism from the host language. More concretely, we can define a function in the host language that for each number `n` returns a Polara term implementing `map` on an array of size `n` (here, λ belongs to the host language and `fun` belongs to Polara):

$$\begin{array}{lll} \text{map} & : & (n : \mathbb{N}) \to (\Gamma \vdash (t_1 \dashrightarrow t_2) \dashrightarrow (n \Rightarrow t_1) \dashrightarrow (n \Rightarrow t_2)) \\ \text{map} & := & \lambda n.\ \mathsf{fun\ f\ a.\ for\ i.\ f\ a[i]} \end{array}$$

Termination. The use of static array sizes ensures that array indexing is total. In fact, every language construct in Polara is deterministic and terminating, making the language total; hence it is not Turing-complete. Most notably, we eschew general recursion in favor of the more well-behaved looping construct `for`. The lack of non-termination allows us to give a simple denotational semantics and guarantees termination of normalization, as described next.

Normalization by Evaluation (NbE). Normalization is defined in Figure 5 by a denotation for types ($[\![t]\!]_\Gamma$: Type), a corresponding denotation for terms and constants (such that when e has type t, then ($[\![e]\!]_\Gamma : [\![t]\!]_\Gamma$)), as well as functions `quote` (η), `splice` (η'), and `norm`. Note the additional argument Γ – this is a peculiarity of the PHOAS representation, where Γ determines the denotation of variables. This argument can take different values, depending on which information we want to extract from a term. For example, when pretty-printing a term we want to produce a string, so we associate every variable also with a string ($\Gamma\ t :=$ String). For NbE, every term should be translated to the denotation of their type, so we associate every variable to the denotation $[\![\cdot]\!]_\Gamma$ of its type using Γ. The function `norm` takes a value of type ($\forall\Gamma.\ \Gamma \vdash t$) and returns one of the same type. The quantification means that we can only use variables that were created by the language's binding constructs, so the type represents closed terms.

The denotation of a bounded natural number is a bounded natural number term, the denotation of a floating point number is a floating point number term, the denotation of a product is a product of the denotations, the denotation of a function is a function of the denotations, the denotation of an array is a function from a bounded natural number term to a denotation of the array's content. Later, for code generation, we will again distinguish functions and arrays. But for the purpose of normalization by partial evaluation (NbE), we model arrays as functions so as to reduce the need for rules for both of them.

The quote η and splice η' functions perform eta-expansion of terms by recursion over the types. Quote turns denotations into terms, and splice turns terms back into denotations. The denotation of a Polara term is a corresponding host-language value of that term (i.e., a Lean value in our mechanization). NbE then evaluates terms in the environment of splicing, followed by quoting the denotation back into a term.

Constants denote functions that check for whether their argument is known, and the partial evaluation of their argument; otherwise, they quote/splice the term into a denotation of the type.

$$\begin{aligned}
\llbracket\cdot\rrbracket_\Gamma &: \mathrm{Ty} \to \mathrm{Type} \\
\llbracket \mathsf{fin}\ n \rrbracket_\Gamma &= \Gamma \vdash \mathsf{fin}\ n \\
\llbracket \mathsf{flt} \rrbracket_\Gamma &= \Gamma \vdash \mathsf{flt} \\
\llbracket t_1 \hat{\times} t_2 \rrbracket_\Gamma &= \llbracket t_1 \rrbracket_\Gamma \times \llbracket t_2 \rrbracket_\Gamma \\
\llbracket t_1 \hat{\to} t_2 \rrbracket_\Gamma &= \llbracket t_1 \rrbracket_\Gamma \to \llbracket t_2 \rrbracket_\Gamma \\
\llbracket n \hat{\Rightarrow} t \rrbracket_\Gamma &= (\Gamma \vdash \mathsf{fin}\ n) \to \llbracket t \rrbracket_\Gamma
\end{aligned}$$

(a) Denotation of types.

$$\llbracket \mathsf{ite}\ e_1\ e_2\ e_3 \rrbracket = \begin{cases} \llbracket e_2 \rrbracket & \text{if } \llbracket e_1 \rrbracket = 1 \\ \llbracket e_3 \rrbracket & \text{if } \llbracket e_1 \rrbracket = 0 \\ \eta'(\mathsf{ite}\ \llbracket e_1 \rrbracket\ (\eta\llbracket e_2 \rrbracket)\ (\eta\llbracket e_3 \rrbracket)) & \text{otherwise} \end{cases}$$

(b) Denotation of ite.

$$\begin{aligned}
\llbracket\cdot\rrbracket &: (\llbracket\cdot\rrbracket_\Gamma \vdash t) \to \llbracket t \rrbracket_\Gamma \\
\llbracket x \rrbracket &= x \\
\llbracket \mathsf{fun}\ i.\ e \rrbracket &= \lambda i.\ \llbracket e\ i \rrbracket \\
\llbracket \mathsf{for}\ i.\ e \rrbracket &= \lambda i.\ \overline{\llbracket e\ i \rrbracket} \\
\llbracket c\ \bar{e} \rrbracket &= \llbracket c \rrbracket\ \overline{\llbracket e \rrbracket} \\
\llbracket \mathsf{let}\ e_1;\ e_2 \rrbracket &= \llbracket e_2 \rrbracket\ \llbracket e_1 \rrbracket \\
\\
\llbracket \mathsf{app} \rrbracket\ e_1\ e_2 &= e_1\ e_2 \\
\llbracket \mathsf{get} \rrbracket\ e_1\ e_2 &= e_1\ e_2 \\
\llbracket \mathsf{pair} \rrbracket\ e_1\ e_2 &= (e_1,\ e_2) \\
\llbracket \mathsf{fst} \rrbracket\ e &= e.1 \\
\llbracket \mathsf{snd} \rrbracket\ e &= e.2 \\
\llbracket \mathsf{sum} \rrbracket\ e &= \eta'\ (\mathsf{sum}\ (\eta\ e)) \\
\llbracket \hat{+} \rrbracket\ n_1\ n_2 &= n_1 + n_2 \\
\llbracket \hat{+} \rrbracket\ e_1\ e_2 &= e_1 \hat{+} e_2
\end{aligned}$$

(c) Denotation of terms and constants.

$$\begin{aligned}
\eta &: \forall t.\ \llbracket t \rrbracket_\Gamma \to (\Gamma \vdash t) \\
\eta\ (t_1 \hat{\to} t_2)\ e &= \mathsf{fun}\ i : t_1.\ \eta\ t_2\ (e\ (\eta'\ t_1\ i)) \\
\eta\ (n_1 \hat{\Rightarrow} t_2)\ e &= \mathsf{for}\ i : n_1.\ \eta\ t_2\ (e\ (\eta'\ t_1\ i)) \\
\eta\ (t_1 \hat{\times} t_2)\ e &= \mathsf{tup}\ (\eta\ t_1\ e.1)\ (\eta\ t_2\ e.2) \\
\eta\ (\mathsf{fin}\ n)\ e &= e \\
\eta\ \mathsf{flt}\ e &= e \\
\\
\eta' &: \forall t.\ (\Gamma \vdash t) \to \llbracket t \rrbracket_\Gamma \\
\eta'\ (t_1 \hat{\to} t_2)\ e &= \lambda i.\ \mathsf{app}\ (\eta'\ t_2\ e)\ (\eta\ t_1\ i) \\
\eta'\ (n_1 \hat{\Rightarrow} t_2)\ e &= \lambda i.\ \mathsf{get}\ (\eta'\ t_2\ e)\ (\eta\ t_1\ i) \\
\eta'\ (t_1 \hat{\times} t_2)\ e &= (\eta'\ t_1\ (\mathsf{fst}\ e),\ \eta'\ t_2\ (\mathsf{snd}\ e)) \\
\eta'\ (\mathsf{fin}\ n)\ e &= e \\
\eta'\ \mathsf{flt}\ e &= e \\
\\
\mathrm{norm} &: (\forall\Gamma.\ \Gamma \vdash t) \to (\forall\Gamma.\ \Gamma \vdash t) \\
\mathrm{norm}\ e &= \eta\ \llbracket e \rrbracket
\end{aligned}$$

(d) Quote η, splice η', and normalization norm.

Figure 5 Typed partial evaluation.

4.2 A$_i$NF and Common Subexpression Elimination

FOAS. An essential component for implementing common subexpression elimination is the ability to compare to terms for equality. As we cannot decide equality over functions, we have to convert from parametric higher-order abstract syntax (PHOAS) to first-order syntax (FOAS) to get decidable equality for identifiers and terms containing variables.

A$_i$NF. In A$_i$NF, we distinguish between variables x and indices i (Figure 3). Variables are introduced by let-binding, while indices are introduced by functions and loops. An A$_i$NF term is a sequence of pattern-matching let-bindings of primitives, ending in a final variable (Figure 3, A$_i$NF). An essential property of A$_i$NF is thus, that it is both maximally fissioned (each for loop just has a single operation as a body) and maximally flat (an A$_i$NF term is a single list of terms without subterms, executed one after another). Pattern matching contexts C have one hole for the variable, and one form for each higher-order argument to any term former, namely array construction, function abstraction, if-consequence, and if-alternative. Primitives are constant application, indices, variable access, function abstraction, and array construction.

$$\begin{array}{ll}
C\llbracket\ \cdot\ \rrbracket\ \cdot & : (\vdash t_1) \to (\mathrm{Var}\ t_1 \to\ \mathrm{A}_i\mathrm{NF}\ t_2) \to\ \mathrm{A}_i\mathrm{NF}\ t_2 \\
C\llbracket\ x\ \rrbracket\ k & = \llparenthesis\ x\ \rrparenthesis\ k \\
C\llbracket\ i\ \rrbracket\ k & = \llparenthesis\ i\ \rrparenthesis\ k \\
C\llbracket\ c\ e_1\ e_2\ \rrbracket\ k & = C\llbracket e_1\rrbracket\ \lambda\, x_1.\ \ C\llbracket e_2\rrbracket\ \lambda\, x_2.\ \ \llparenthesis\ c\ x_1\ x_2\ \rrparenthesis\ k \\
C\llbracket \mathsf{fun}\ i{:}t.\ e\rrbracket\ k & = C[\mathsf{fun}\ i{:}t.\ \cdot]\llbracket e\rrbracket\ \lambda\, x.\ \ \llparenthesis\ \mathsf{fun}\ i{:}t.\ x\ \rrparenthesis\ k \\
C\llbracket \mathsf{for}\ i{:}e_1.\ e_2\rrbracket\ k & = C\llbracket e_1\rrbracket\ \lambda\, x_1.\ \ C[\mathsf{for}\ i{:}x_1.\ \cdot]\llbracket e_2\rrbracket\ \lambda\, x_2.\ \ \llparenthesis\ \mathsf{for}\ i{:}x_1.\ x_2\ \rrparenthesis\ k \\
C\llbracket \mathsf{ite}\ e_1\ e_2\ e_3\rrbracket\ k & = C\llbracket e_1\rrbracket\ \lambda\, x_1.\ \ C[\mathsf{if}\ x_1{=}0.\ \cdot]\llbracket e_2\rrbracket\ \lambda\, x_2.\ \ C[\mathsf{if}\ x_2{\neq}0.\ \cdot]\llbracket e_3\rrbracket\ \lambda\, x_3.\ \ \llparenthesis \mathsf{ite}\ x_1\ x_2\ x_3\rrparenthesis\ k \\
C\llbracket \mathsf{let}\ e_1;\ e_2\rrbracket\ k & = C\llbracket e_1\rrbracket\ \lambda\, x_1.\ \ C\llbracket e_2\ x_1\rrbracket\ k
\end{array}$$

(a) Fission.

$$\begin{array}{lcl}
\llparenthesis\ \cdot\ \rrparenthesis & : & \mathrm{Prim}\ t_1 \to (\mathrm{Var}\ t_1 \to \mathrm{A}_i\mathrm{NF}\ t_2) \to \mathrm{A}_i\mathrm{NF}\ t_2 \\
\llparenthesis\ x\ \rrparenthesis\ k & = & k\ x \\
\llparenthesis\ p\ \rrparenthesis\ k & = & \mathsf{let}\ x = p;\ \ k\ x \qquad \text{where } x \text{ unique}
\end{array}$$

(b) Smart binding.

Figure 6 Fission with smart binding.

Conversion to A$_i$NF (Figure 6) exploits the fact that continuation-passing-style automatically flattens code. The function $C\llbracket\ e\ \rrbracket\ k$ takes as inputs a POLARA term e, a pattern matching context C, and a continuation k, and returns an A$_i$NF term. In the mechanization the function uses a reader monad as well to generate unique variable names. The function is initialized with the empty pattern matching context, and the identity continuation; the variable counter is initialized with zero.

Another important helper function is smart binding $\llparenthesis\ p\ \rrparenthesis\ k$, which takes a primitive p and a continuation k. Smart binding ensures that every primitive term passed to it is bound to a variable name, and that variable name is passed to the continuation. If the primitive term is a variable already, this variable name is passed to the continuation; otherwise the term is bound to a unique variable name, incrementing the counter.

Translation to A$_i$NF by $C\llbracket\ e\ \rrbracket\ k$ recurses structurally over the term e. In the case of a variable or an index, the term is forwarded to smart binding. In the case of a constant application (exemplary shown for binary constant application), first the first subterm is translated, then in the continuation the second subterm is translated, and in the continuation the term is reconstructed as a primitive with variable referencing the name of the translated subterms, which is passed to smart binding to generate a new name for this term, passing the continuation along. In the case of function abstraction, array construction, and conditional expressions, the subterms are translated as well, but in adapted contexts, and the final term is passed as well to smart binding to generate a name for it, and the continuation is passed along. Concretely, in the case of function abstraction, the function body is translated in a context which includes the function argument. In the case of array construction, the array body is translated in a context which includes the iteration variable. In the case of conditional expressions, the consequence is translated in a context which includes the condition, and the alternative is translated in a context which includes the negation of the condition. Finally, in the case of let-binding, first the right-hand side of the binding is translated, and then the body of the binding.

CSE. In addition to deciding equality for terms, a further complication with common subexpression elimination is that we also need to decide equality in the presence of already established equalities. For example consider the term `x=v, y=v, z=(x+y), q=(y+x), t=z+1,`

$$\begin{array}{lcl}
\text{Ren} & = & [(t_1 : \text{Ty}) \times \text{Var } t_1 \times \text{Var } t_1] \\
\text{Nam} & = & [(t_1 : \text{Ty}) \times \text{Prim } t_1 \times \text{Var } t_1] \\
\text{CSE} & : & \text{Ren} \to \text{Nam} \to \text{A}_i\text{NF } t_2 \to \text{A}_i\text{NF } t_2 \\
\text{CSE } r\ \sigma\ x & = & (\sigma,\ \text{ren}_r\ x) \\
\text{CSE } r\ \sigma\ (\mathsf{let}\ C[x = p];\ a) & = & \begin{cases} \text{CSE } r\ (\mathsf{let}\ C'[x = p']; \sigma)\ a \\ \qquad\qquad \text{if lookup } \sigma\ C'\ x\ p' = \text{none} \\ \\ \text{CSE } ([x := x'] :: r)\ \sigma'\ a \\ \qquad\qquad \text{if lookup } \sigma\ C'\ x\ \ p' = \text{some } (x', \sigma') \end{cases} \\
& & \text{where } p' = \text{ren}_r\ p \\
& & \text{where } C' = \text{ren}_r\ C
\end{array}$$

Figure 7 Common subexpression elimination.

`r=q+1, ...`. Correct CSE should eliminate it to `x=v, z=(x+x); t=z+1; rename [y→x; q→z; r→t], ...`. Notice how the later eliminations are dependent on the earlier ones. If we simply rename the remaining term every time we detect a variable to be redundant, then this algorithm would perform exponentially worse, because every renaming is a traversal over the whole remaining term, and CSE itself is already a traversal over the whole term. To keep everything with a single traversal, we adapt CSE to carry a renaming with it, which is applied just before a term is checked for redundancy.

CSE (Figure 7) takes a renaming, a naming, and an A_iNF term, and returns a new A_iNF term of the same type. A renaming is a list of pairs of variables of the same type, representing that the first variable is to be replaced by the second. A naming is a list of pairs of a primitive and a variable of the same type, representing that the primitive term has been previously bound to that variable. CSE works by structural recursion over the term. When the input term is just a variable, it simply applies the renaming. When the input term is a let-binding, then the renaming is applied to the term as well. The renamed term is looked up in the list of previously defined terms. If the term has not been bound to a variable name already (none), then the term is now let-bound to a variable, inside a renamed pattern matching context C. CSE proceeds with the remaining terms a, remembering that the term p' has been bound to σ, so that future redundant occurrences of p' can be eliminated. If the term has already been bound to a variable name (some x'), then no let-binding is produced, but only the renaming is extended to replace future references to x to the already existing x' instead. CSE proceeds with the remaining terms a, remembering that the term p' has been bound to σ, so that future redundant occurrences of p' can be eliminated.

4.3 Mechanization in Lean

In this section, we present excerpts from the Lean mechanization and relate them to the paper formalization. The type of terms `Tm` corresponds to $(\Gamma \vdash t)$ and 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`. We define the following types corresponding to the above definitions of syntax (Figure 3) in Lean.

```
inductive Var : Ty → Type -- Variables Var
inductive Par : Ty → Type -- Indices   Idx

inductive Ty                          -- Types t
inductive Const0 : Ty → Type          -- Constants c (nullary)
inductive Const1 : Ty → Ty → Type     -- Constants c (unary)
inductive Const2 : Ty → Ty → Ty → Type -- Constants c (ternary)
inductive Tm (Γ: Ty → Type): Ty → Type -- Terms e

inductive Prim : Ty → Type -- Primitives p
inductive Env  : Type      -- Scoped Contexts C
inductive AINF : Ty → Type -- AINF a
                                                                  Lean
```

In particular, we define the following functions in Lean. The function `Ty.de` corresponding to denotation of types $[\![\cdot]\!]_\Gamma$, `quote` to η and `splice` to η', `Const0.de`, `Const1.de`, `Const2.de` and `Tm.de` were shown as term, constant, and its denotations $[\![\cdot]\!]$. Finally, `norm` is defined using `term` denotations and `quote`.

```
def Ty.de (Γ : Ty → Type): Ty → Type

def quote {Γ} : {α : Ty} → Ty.de Γ α → Tm Γ α
def splice {Γ} : {α : Ty} → Tm Γ α → Ty.de Γ α

def Const0.de : Const0 α → Ty.de Γ α
def Const1.de : Const1 β α → Ty.de Γ β → Ty.de Γ α
def Const2.de : Const2 γ β α → Ty.de Γ γ → Ty.de Γ β → Ty.de Γ α
def Tm.de : Tm (Ty.de Γ) α → Ty.de Γ α

def Tm.norm : (∀ Γ, Tm Γ α) → Tm Γ α
  | e ⇒ quote (Tm.de (e _))
                                                                  Lean
```

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. The `rename` functions define how a renaming is 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`. 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 particular, we define the following functions in Lean, corresponding to the functions above:

```
def Prim.beq : Prim α → Prim α → Bool
def AINF.beq : AINF α → AINF α → Bool
def AINF.smart_bnd : Env → Prim α → (VPar α → Counter (AINF β)) → Counter (AINF β)
def Tm.toAINF (e : Tm VPar α) : AINF α
def Var.rename : Ren → Var α → Var α
def VPar.rename (r: Ren): VPar α → VPar α
def Env.rename (r: Ren): Env → Env
def Prim.rename (r: Ren): Prim α → Prim α
def AINF.rename (r: Ren): AINF α → AINF α
def AINF.rename (r: Ren): AINF α → AINF α
def Env.or (Γ: Env) (Δ: Env): Tern → Option Env := fun t ⇒ match Γ, Δ with
def RAINF.upgrade : RAINF → Var b → Env → Option RAINF
def AINF.cse' : Ren → RAINF → AINF α → (RAINF × VPar α)
def merge: RAINF → VPar α → AINF α

def AINF.cse : Ren → RAINF → AINF α → AINF α
  | r, σ, a ⇒ let (b, c) := a.cse' r σ; merge b.reverse c
                                                              Lean
```

4.4 Proofs

In this section, we show that normalization and translation to A$_i$NF are type-preserving, i.e. given a well-typed term, they always produce a valid term of the same type. We also show that translation to A$_i$NF produces maximally fissioned terms.

We use an intrinsically typed approach where the type system of the object language is included in the encoding of the data type for the language's syntax. Therefore, the host languages type system ensures only well-typed terms can be constructed.

Following an intrinsically typed approach means that the soundness properties hold simply because our (appropriately typed) definitions type check. We do not have to state and prove explicit, separate theorems, because the types of the functions already carry the necessary information.

▶ **Theorem 1** (Well-typedness of Optimization).
Our optimization procedure is terminating and type preserving.

Proof. Termination is ensured by Lean's built-in termination check. The fact that normalization terminates relies on POLARA being a total language. In particular, the absence of unbounded recursion and the combination of static array sizes with intrinsic typing avoids infinite loops and out-of-bounds accesses, ensuring that our normalization function always successfully terminates. Type preservation is ensured by intrinsically-typed mechanization; consider the types of normalization and CSE in Lean:

```
def Tm.norm : (∀ Γ, Tm Γ α) → Tm Γ α
  | e ⇒ quote (Tm.de (e _))
def AINF.cse : Ren → RAINF → AINF α → AINF α
  | r, σ, a ⇒ let (b, c) := a.cse' r σ; merge b.reverse c
                                                              Lean
```

Intrinsic typing defines the typing of the object language (here, POLARA) using the typing of the host language (here, Lean), so the host language's type checker prevents the creation of ill-typed object language programs. This means that an element of `(∀ Γ, Tm Γ α)` is a well-typed POLARA program and an element of `AINF α` is a well-typed A$_i$NF term. Further, given a well-typed term, each function returns a well-typed term, which is what we mean by soundness with regard to the type system. ◀

▶ **Theorem 2** (Well-typedness of Translation).
Our translation procedure is terminating and type preserving.

Proof. Again, termination is guaranteed by Lean's termination checker. The argument for type preservation is similar to the one above: As both POLARA and A$_i$NF are defined using intrinsic typing, we can only construct well-typed programs. Consider the type of `toAINF` (we omit the definition):

```
def Tm.toAINF (e : Tm VPar α) : AINF α
                                                              Lean
```

If one tried to define `toAINF` in a way that produces an ill-typed program, the definition would be rejected by the type checker. ◀

Finally, A$_i$NF is inductively defined to be maximally fissioned, i.e., as a list of primitives without subterms, therefore the act of translating POLARA terms into A$_i$NF in a total programming language performs loop fission by definition.

▶ **Theorem 3** (Maximal Fission).
Our translation into A$_i$NF produces terms with maximal fission.

Proof. Consider the definition of A$_i$NF terms:

```lean
inductive AINF : Ty → Type
  | ret : VPar α → AINF α
  | bnd : Env → Var α → Prim α → AINF β → AINF β
```
Lean

Here, a value of type `VPar α` can be a variable or a parameter. A value of type `Prim α` is a primitive (not nested) operation. The constructor `bnd` represents a variable assignment while `ret` returns a variable or parameter and represents the end of the program. From this inductive definition, it is apparent that all A$_i$NF terms have a flat structure where nested expressions are impossible. Recall that, in A$_i$NF, each assignment is considered its own separate loop. Because the body of each assignment only contains a single primitive, each loop has a body only consisting of one operation and hence an A$_i$NF term is guaranteed to be maximally fissioned. Because the translation function `toAINF` has output type `AINF α`, it can *only* produce such maximally fissioned terms. Further, Lean's termination checker ensures that `toAINF` is total, and so always returns an A$_i$NF term in finite time. ◀

5 Related Work

5.1 Intermediate Languages

Early work by Steele [38] implemented a continuation-passing-style (CPS) IR in a functional compiler, stressing the suitability of CPS for compilation, as it closely mimics how control flow is expressed with jumps in hardware instructions, and makes evaluation order explicit in the syntax. Appel [2] observed that beta-reductions in the lambda calculus are unsound in the presence of side effects as they could duplicate the effect. Yet, CPS, which makes evaluation order explicit, enables to perform certain optimizations, such as dead code elimination (DCE), and common subexpression elimination (CSE), by exploiting that in CPS every subterm is referenced by a unique name.

Sabry and Felleisen [34] identified that additional power of compiling in CPS [30] corresponds to the additional rules of the monadic computational language [24]. Of particular importance is the so-called associativity law of the monad, i.e., the let-let commuting conversion, enabling the flattening of code. Then, Flanagan [11] coined the name "A-normal form (ANF)" for the now popular IR, which in contrast to CPS, expresses sequential execution by simple let-binding rather than continuations. The difference between ANF and the monadic language is that ANF forbids nested let-bindings, i.e., code must be normal with regard to the associativity rule of the monad.

However, Kennedy [18] showed that moving from CPS to ANF did not take into account branching. More precisely, while the let-let commuting conversion enables the flattening of code, the let-if commuting conversion *duplicates* code into each branch, in the worst case leading to blow-up of code size exponential in the number of branches. Given that recursion always includes a branch for base case(s) and step case(s), the same problem appears with recursion. Kennedy therefore argued for a return to CPS.

The issue was resolved by Maurer et al. [22], who provided an implementation of ANF for use in the Glasgow Haskell Compiler (GHC), and further simplified by Cong et al. [6] who provided implementations for MiniScala and Lightweight-Modular-Staging (LMS). Cong showed that it is possible to combine the simplicity of let-bindings for sequential execution and the power of continuations for further control flow, by adding control operators to ANF, which enable to capture the current continuation.

In our work, we highlight the importance of commuting conversions, and extend the idea of having an intermediate language that intrinsically encodes maximal let-let conversion to an intermediate language that also intrinsically encodes maximal let-for and let-if commuting conversion, without exponential blow-up of code size. Further, as this work lies in the context of array programming, recursion is not often necessary, and can thus be avoided.

The logical connection between polarity and common subexpression elimination has also been explored by Miller and Wu [23].

5.2 Array Programming

Shaikhha et al. [37] present a differentiable programming language which is an extension of the lambda calculus. For array computations, they use an approach based on higher-order functions. It directly represents the duality of functions and arrays through built-in functions `build` for creating an array from a function and `get` for turning an array into a function. The fact that `get` is a left inverse of `build` leads to the equivalence `get (build n e) i` $\equiv$ `e i`, which can be used for optimization. Another strand of research makes use of the standard technique of *rewriting strategies* to optimize functional array programs [13, 4], suggesting the viability of standard techniques from term rewriting for optimizing array programs. Liu et al. [20] present a framework that can express a variety of optimizations through formally verified term rewriting, achieving competitive performance; however, CSE is not addressed. Their representation is first-order and features an array generation construct similar to the one in Polara. Optimization in Polara is not based on rewriting, but instead uses partial evaluation.

Feldspar [3] is a DSL for array computations in Haskell. It features a `parallel` construct similar to our `for` constructor, as well as `while` loops. Feldspar is compiled to C and performs standard optimizations like fusion, as well as copy propagation and loop unrolling. Feldspar's backend uses a dataflow graph and an imperative intermediate representation, whereas our intermediate representation is functional and specifically designed to support optimizations on array programs.

SaC [35] is a functional first-order array language. Array computations are expressed using *with-loops*, which consist of at least one generator and one operator. Each generator consists of an index range and an expression giving the value of the output array at a given index in the range. The operator can provide default values for indices not included in any generator, a base array that should be modified by the generators, or it can describe an aggregation. More recently, SaC has added support for *tensor comprehensions* [36], which drop the operator part and add pattern matching on indices as well as bound and shape inference, making the notation more lightweight. Similar to our approach, their tensor comprehensions do not support summation, which is added in the form of a built-in function. SaC's optimizations are not based on a logical foundation, but consist of a pipeline of optimization algorithms. In Polara we derive our syntax form for pattern matching on arrays from polarization type theory, enabling additional commuting conversions and thus grounding our optimization algorithm on a logical foundation.

6 Conclusion

This paper introduced A_iNF, a novel intermediate representation for array computations, and Polara, a surface array language. The proposed optimization algorithm for A_iNF, based on typed partial evaluation and common subexpression elimination, simplifies program optimization by interpreting arrays as positively polarized types. This approach avoids complexities associated with optimization schedules for conventional ANF. We formalized A_iNF and Polara. We proved sound the translation from Polara to A_iNF and optimization.

For future work, we are working on extending the language with automatic differentiation and probabilistic primitives, and proving these extensions correct as well. We are interested in applying our optimization to redundancies generated by automatic differentiation. Further, given that A_iNF, based on ANF, is related to monadic notation, it would be interesting to investigate whether Applicative notation [21, 33], Arrow notation [16], and Comonad Notation [27] provide similar insights for normalization by evaluation approach to optimization.

References

1 Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *J. Log. Comput.*, 2(3):297–347, 1992. `doi:10.1093/LOGCOM/2.3.297`.

2 Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

3 Emil Axelsson, Koen Claessen, Gergely Dévai, Zoltán Horváth, Karin Keijzer, Bo Lyckegård, Anders Persson, Mary Sheeran, Josef Svenningsson, and András Vajda. Feldspar: A domain specific language for digital signal processing algorithms. In *8th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010), Grenoble, France, 26-28 July 2010*, pages 169–178. IEEE Computer Society, 2010. `doi:10.1109/MEMCOD.2010.5558637`.

4 Timon Böhler, David Richter, and Mira Mezini. Using rewrite strategies for efficient functional automatic differentiation. In Aaron Tomb, editor, *Proceedings of the 25th ACM International Workshop on Formal Techniques for Java-like Programs, FTfJP 2023, Seattle, WA, USA, 18 July 2023*, pages 51–57. ACM, 2023. `doi:10.1145/3605156.3606456`.

5 Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In James Hook and Peter Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 143–156. ACM, 2008. `doi:10.1145/1411204.1411226`.

6 Youyou Cong, Leo Osvald, Grégory M. Essertel, and Tiark Rompf. Compiling with continuations, or without? whatever. *Proceedings of the ACM on Programming Languages*, 3(ICFP):79:1–79:28, 2019. `doi:10.1145/3341643`.

7 Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021. `doi:10.1007/978-3-030-79876-5_37`.

8 Paul Downen and Zena M. Ariola. The duality of construction. In Zhong Shao, editor, *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8410 of *Lecture Notes in Computer Science*, pages 249–269. Springer, 2014. `doi:10.1007/978-3-642-54833-8_14`.

9 Paul Downen and Zena M. Ariola. Compiling with classical connectives. *Log. Methods Comput. Sci.*, 16(3), 2020. URL: `https://lmcs.episciences.org/6740`.

10 Paul Downen and Zena M. Ariola. Duality in action (invited talk). In Naoki Kobayashi, editor, *6th International Conference on Formal Structures for Computation and Deduction, FSCD 2021, July 17-24, 2021, Buenos Aires, Argentina (Virtual Conference)*, volume 195 of *LIPIcs*, pages 1:1–1:32. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPICS.FSCD.2021.1`.

11 Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In Robert Cartwright, editor, *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, pages 237–247. ACM, 1993. doi:10.1145/155090.155113.

12 Jeremy Gibbons. APLicative programming with Naperian functors. In Hongseok Yang, editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10201 of *Lecture Notes in Computer Science*, pages 556–583. Springer, 2017. doi:10.1007/978-3-662-54434-1_21.

13 Bastian Hagedorn, Johannes Lenfers, Thomas Koehler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. Achieving high performance the functional way: Expressing high-performance optimizations as rewrite strategies. *Commun. ACM*, 66(3):89–97, 2023. doi:10.1145/3580371.

14 Charles R. Harris, K. Jarrod Millman, Stéfan van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nat.*, 585:357–362, 2020. doi:10.1038/S41586-020-2649-2.

15 Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. Futhark: purely functional GPU-programming with nested parallelism and in-place array updates. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 556–571. ACM, 2017. doi:10.1145/3062341.3062354.

16 John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1-3):67–111, 2000. doi:10.1016/S0167-6423(99)00023-4.

17 Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation.* Prentice Hall international series in computer science. Prentice Hall, 1993.

18 Andrew Kennedy. Compiling with continuations, continued. In Ralf Hinze and Norman Ramsey, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, pages 177–190. ACM, 2007. doi:10.1145/1291151.1291179.

19 Neelakantan R. Krishnaswami. Focusing on pattern matching. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 366–378. ACM, 2009. doi:10.1145/1480881.1480927.

20 Amanda Liu, Gilbert Louis Bernstein, Adam Chlipala, and Jonathan Ragan-Kelley. Verified tensor-program optimization via high-level scheduling rewrites. *Proc. ACM Program. Lang.*, 6(POPL):1–28, 2022. doi:10.1145/3498717.

21 Simon Marlow, Simon Peyton Jones, Edward Kmett, and Andrey Mokhov. Desugaring Haskell's do-notation into applicative operations. In Geoffrey Mainland, editor, *Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016*, pages 92–104. ACM, 2016. doi:10.1145/2976002.2976007.

22 Luke Maurer, Paul Downen, Zena M. Ariola, and Simon L. Peyton Jones. Compiling without continuations. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 482–494. ACM, 2017. doi:10.1145/3062341.3062380.

23 Dale Miller and Jui-Hsuan Wu. A positive perspective on term representation (invited talk). In Bartek Klin and Elaine Pimentel, editors, *31st EACSL Annual Conference on Computer Science Logic, CSL 2023, February 13-16, 2023, Warsaw, Poland*, volume 252 of *LIPIcs*, pages 3:1–3:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.CSL.2023.3.

24 Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*, pages 14–23. IEEE Computer Society, 1989. `doi:10.1109/LICS.1989.39155`.

25 Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Trans. Comput. Log.*, 9(3):23:1–23:49, 2008. `doi:10.1145/1352582.1352591`.

26 nLab authors. function type. `https://ncatlab.org/nlab/show/function+type`, January 2024. Revision 33.

27 Dominic A. Orchard and Alan Mycroft. A notation for comonads. In Ralf Hinze, editor, *Implementation and Application of Functional Languages - 24th International Symposium, IFL 2012, Oxford, UK, August 30 - September 1, 2012, Revised Selected Papers*, volume 8241 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2012. `doi:10.1007/978-3-642-41582-1_1`.

28 Adam Paszke, Daniel D. Johnson, David Duvenaud, Dimitrios Vytiniotis, Alexey Radul, Matthew J. Johnson, Jonathan Ragan-Kelley, and Dougal Maclaurin. Getting to the point: index sets and parallelism-preserving autodiff for pointful array programming. *Proc. ACM Program. Lang.*, 5(ICFP):1–29, 2021. `doi:10.1145/3473593`.

29 Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In Richard L. Wexelblat, editor, *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, pages 199–208. ACM, 1988. `doi:10.1145/53990.54010`.

30 Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975. `doi:10.1016/0304-3975(75)90017-1`.

31 Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman P. Amarasinghe, and Frédo Durand. Halide: decoupling algorithms from schedules for high-performance image processing. *Commun. ACM*, 61(1):106–115, 2018. `doi:10.1145/3150211`.

32 David Richter, Timon Böhler, Pascal Weisenburger, and Mira Mezini. stg-tud/ainf-compiling-with-arrays. Software, swhId: `swh:1:dir:8e0e755d11e4e3e91fb05bf8df1a5c8bec0f553a` (visited on 2024-09-02). URL: `https://github.com/stg-tud/ainf-compiling-with-arrays`.

33 David Richter, Timon Böhler, Pascal Weisenburger, and Mira Mezini. A direct-style effect notation for sequential and parallel programs. In Karim Ali and Guido Salvaneschi, editors, *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States*, volume 263 of *LIPIcs*, pages 25:1–25:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. `doi:10.4230/LIPICS.ECOOP.2023.25`.

34 Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *LISP Symb. Comput.*, 6(3-4):289–360, 1993.

35 Sven-Bodo Scholz. Single Assignment C: efficient support for high-level array operations in a functional setting. *J. Funct. Program.*, 13(6):1005–1059, 2003. `doi:10.1017/S0956796802004458`.

36 Sven-Bodo Scholz and Artjoms Sinkarovs. Tensor comprehensions in SaC. In Jurriën Stutterheim and Wei-Ngan Chin, editors, *IFL '19: Implementation and Application of Functional Languages, Singapore, September 25-27, 2019*, pages 15:1–15:13. ACM, 2019. `doi:10.1145/3412932.3412947`.

37 Amir Shaikhha, Andrew W. Fitzgibbon, Dimitrios Vytiniotis, and Simon Peyton Jones. Efficient differentiable programming in a functional array-processing language. *Proc. ACM Program. Lang.*, 3(ICFP):97:1–97:30, 2019. `doi:10.1145/3341701`.

38 Guy L. Steele. Rabbit: A compiler for Scheme. Technical report, Massachusetts Institute of Technology, USA, 1978.

39 Noam Zeilberger. On the unity of duality. *Ann. Pure Appl. Log.*, 153(1-3):66–96, 2008. `doi:10.1016/J.APAL.2008.01.001`.

40 Noam Zeilberger. *The logical basis of evaluation order and pattern-matching.* PhD thesis, Carnegie Mellon University, USA, 2009. AAI3358066.