Correctness is Demanding, Performance is Frustrating

ANONYMOUS AUTHOR(S)

In this paper we demonstrate a technique for developing high performance applications with strong correctness guarantees. We use a theorem prover to derive a high-level specification of the application that includes correctness invariants of our choice. After that, within the same theorem prover, we implement an extraction of the specified application into a high-performance language of our choice. Concretely, we are using Agda to specify a framework for automatic differentiation (reverse mode) that is focused on index-safe tensors. This framework comes with an optimiser for tensor expressions and the ability to translate these expressions into SaC and C. We specify a canonical convolutional neural network within the proposed framework, compute the derivatives needed for the training phase and then demonstrate that the generated code matches the performance of hand-written code when running on a multi-core machine.

Additional Key Words and Phrases: Dependent Types, Agda, Array Programming, Automatic Differentiation, SaC

ACM Reference Format:
Anonymous Author(s). 2025. Correctness is Demanding, Performance is Frustrating. 1, 1 (February 2025), 27 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

The year is 2024, and we still have to make a formidable choice between correctness and performance for all the programming projects that we start. Low level programming languages such as C or Fortran make it possible to leverage intricate hardware features for the price of poor analysability and correctness guarantees. Dependently-typed systems such as Lean or Agda make it possible to describe arbitrary invariants within the given program, yet they can rarely generate high-performance code.

The idealist inside of us is exasperated, because there should be a perfect solution that caters for both cases. However, the practitioner within us proposes a different perspective. Instead of using a single language, we may use two languages in cooperation. Specifically, we envision an alliance between a proof assistant and some high-performance language of our choice.

To explore this idea, we investigate the concrete problem of automatic differentiation (AD) which is often found in machine learning applications. This is a convenient case study as it comes with the following challenges. From the correctness perspective, it is crucially important to track the shapes and ranks of the tensors, guaranteeing the absence of out-of-bound indexing. This is a very common source of errors that can be incredibly difficult to find. Secondly, we have to compute derivatives of the given tensor expressions, preserve safe indexing guarantees while we do so, and we have to be able to translate the computed expressions into some high-performance language. As machine learning applications are known to be numerically intensive problems, our performance challenge lies in running the program as fast as we can on the chosen hardware architecture.

We follow [26] which demonstrates that it is possible to implement one of the canonical convolutional neural network (CNN) in the array language SaC [11, 22], obtaining good sequential

ACM XXXX-XXXX/2025/2-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

and parallel performance that is competitive with TensorFlow [2] and PyTorch [18]. Focussing on correctness, we propose a theory of rank-polymorphic arrays [24] in Agda [3]. Within this framework, we encode the CNN from [26] and lift it into an embedded DSL. We implement AD (reverse mode) and domain-specific optimisations for expressions in that DSL. Finally, we implement an extraction into SaC (functional array language) and C (low-level imperative language).

As a result, we demonstrate an approach where the entire specification, optimiser, AD and code generation are available to us within a proof assistant of our choice. We can prove facts about all the stages of the pipeline and easily adjust them to our liking. We argue that such a liberating approach is feasible in practice, at least for the times of dialectic of correctness and performance.

The contributions of this paper are as follows:

(1) a rank-polymorphic array theory in Agda;
(2) an implementation of the CNN from [26] in Agda;
(3) an embedded DSL in Agda which supports AD (reverse mode);
(4) an extraction mechanism for generating SaC or C code from the DSL; and
(5) an experimental evaluation of the generated codes.

This paper is written in literate Agda, which guarantees that all the code snippets have been type-checked.

2 Background

Automatic differentiation has been around for many decades [12, 17], so it is well-understood at a conceptual level. However, a number of questions related to bringing AD into the context of programming languages remain open. Recent successes in machine learning have spurred further interest in AD which has led to several new developments. For the context of this paper, we focus on recent work that contributes to the perspective of balancing correctness guarantees and performance. Our selection here is by no means exhaustive, for a broader scope we refer the reader to [5].

There has been a number of programming-language-oriented approaches that explain how to add AD to a programming language of choice. Examples of these include Futhark [20], Haskell [14], and Jax [15]. Furthermore, a number of machine learning frameworks that incorporate AD have been proposed in recent years: TensorFlow [2], PyTorch [18], MXNet [7] and many more. While in particular the dedicated frameworks tend to find widespread acceptance by practitioners, both, correctness and performance leave two open questions: (i) how is it possible to ensure that the AD algorithm is implemented correctly? (ii) if the language or the framework do not perform as expected, what are the options to solve this? Unfortunately, for many cases the answer to both questions is unsatisfying. Most of the languages/frameworks do not come with formal correctness guarantees, so one has to trust the implementers of these tools. One can run tests as well to gain trust in the implementation but that is far from a formal guarantee. If one relies on the AD provided by a chosen language/framework, and the generated code does not perform well, one has to modify the language/framework, as these solutions are tightly integrated with the tools. The problem here is that most of of these tools have very large and sophisticated implementations typically comprising of hundreds of thousands of lines of code. Furthermore, these systems often rely on sophisticated combinations of sub-systems that need to be fine-tuned to the executing hardware.

Another line of work studies high-level principles of AD using category theory [8, 10, 25]. While this indeed comes with great correctness guarantees due to some naturality principles, it is not always clear how to implement this in a way that leads to efficiently executable specifications. Also, the entire treatment of index-safe tensors is unclear.

In [9] the author proposes to view AD problem using the language of cartesian categories. It has been shown that this approach can be used in practice by implementing the proposed technique in Haskell. Type classes are a vehicle to restrict expressions that are differentiable. There is a Haskell plugin that translates expressions that are instances of the mentioned type classes into categorical primitives, AD is performed on these and the code is reflected back to Haskell. This is a nice approach that makes it easy to verify the correctness of the algorithm. However, the treatment of tensors and general extractability remains a little unclear. While it is briefly mentioned that representable functors are supported, it is unclear whether this is sufficient to represent rank-polymorphic arrays with strict bound checks. Also, correctness guarantees are inevitably restricted by the Haskell type system, so we are likely to find invariants that are inexpressible in that setup.

3 Array Theory

The central data structures of our case study are multi-dimensional arrays. This section is dedicated to defining a minimalist array theory in Agda which is well-suited for a specification of CNNs.

We assume that the reader is sufficiently familiar with Agda's syntax. For gentle introductions we refer to one of the tutorials that are freely available online[1].

The conciseness of the specification in [26] relies on rank-polymorphism, which is the ability to operate on arrays of arbitrary rank. We capture this feature in in our array theory. The central consideration when working with dependent types is how to represent data. Some encodings are better suited for reasoning, others are more efficient at runtime. Due to our two-language setup, our choice of representation is driven by proof considerations only. This enables us to represent arrays as functions from indices to values.

An absence of out-of-bound errors requires that all array indices fall within the shapes of the arrays that they are selecting from. The shape of an array describes the extent of each of its axes. We represent shapes as binary trees of natural numbers using the data type S. Leaves of the shape tree are constructed with ι which takes one argument. The _⊗_ constructor makes a tree of two sub-trees. Note that underscores in _⊗_ specify the position where arguments go, therefore ⊗ is an infix binary operation.

Array positions (indices) are given by the dependent type P which is indexed by shapes. A position within an array of shape s has exactly the same tree structure as s, but the leaves are natural numbers that are bounded by the corresponding shape elements.

Arrays are given by the data type Ar which is indexed by a shape and an element type. The formal definitions of S, P and Ar are as follows:

```
data S : Set where
  ι   : ℕ → S
  _⊗_ : S → S → S

data P : S → Set where
  ι   : Fin n → P (ι n)
  _⊗_ : P s → P p → P (s ⊗ p)

Ar : S → Set → Set
Ar s X = P s → X
```

As arrays are functions, selections are function applications and array construction is a function definition (*e.g.* via λ-abstraction).

Array Combinators. It is helpful to invest a little time in defining array combinators. First, we can observe that Ar of a fixed shape is an applicative functor [16], so we can trivially derive: K x to produce a constant array; map f a to apply f to all the elements of a; and zipWith f a b to

[1] See https://agda.readthedocs.io/en/v2.6.3/getting-started/tutorial-list.html.

point-wise apply the binary operation f to a and b.

```
K : X → Ar s X                    map : (X → Y) → Ar s X → Ar s Y
K x i = x                         map f a i = f (a i)
```

```
zipWith : (X → Y → Z) → Ar s X → Ar s Y → Ar s Z
zipWith f a b i = f (a i) (b i)
```

Arrays are homogeneously nested, *i.e.* the shapes of all the sub-arrays have to be the same. Therefore, we can switch between the array of a product shape and the nested array (array of arrays). This operation is very similar to currying except it happens at the level of shapes. The combinators that achieve this are named nest and unnest and their definitions are:

```
nest : Ar (s ⊗ p) X → Ar s (Ar p X)      unnest : Ar s (Ar p X) → Ar (s ⊗ p) X
nest a i j = a (i ⊗ j)                   unnest a (i ⊗ j) = a i j
```

Reduction. We implement reduction of the binary operations over arrays in two steps. Firstly, we define 1-d reductions that we call sum_1 which is very similar to right fold on lists. The arrays of higher ranks iterate sum_1 bottom-up. The definition of the primitives are as follows:

```
ιsuc : P (ι n) → P (ι (suc n))
ιsuc (ι i) = ι (suc i)
```

```
sum₁ : (X → X → X) → X → Ar (ι n) X → X
sum₁ {n = zero}  f ϵ a = ϵ
sum₁ {n = suc n} f ϵ a = f (a (ι zero)) (sum₁ f ϵ (a ∘ ιsuc))
```

```
sum : (X → X → X) → X → Ar s X → X
sum {s = ι n}     f ϵ a = sum₁ f ϵ a
sum {s = s ⊗ p} f ϵ a = sum f ϵ $ map (sum f ϵ) (nest a)
```

Note that our reduction forces the types of the arguments of the binary operation to be the same, which is different from the usual foldr definition. While we do not need this functionality for our example, it is worth noting that the standard behaviour can be recovered[2] through reduction of function composition.

Reshaping. One common operation on arrays is element-preserving change of shape. We call such an operation reshape. It is clear that array elements can be preserved only in cases when the number of elements in the original array and the reshaped one is the same. We define an inductive relation Reshape that relates only those shapes that preserve the number of array elements.

```
data Reshape : S → S → Set where
  eq     : Reshape s s
  _•_    : Reshape p q → Reshape s p → Reshape s q
  _,_    : Reshape s p → Reshape q r → Reshape (s ⊗ q) (p ⊗ r)
  split  : Reshape (ι (m * n)) (ι m ⊗ ι n)
  flat   : Reshape (ι m ⊗ ι n) (ι (m * n))
  swap   : Reshape (s ⊗ p) (p ⊗ s)
  assocl : Reshape (s ⊗ (p ⊗ q)) ((s ⊗ p) ⊗ q)
  assocr : Reshape ((s ⊗ p) ⊗ q) (s ⊗ (p ⊗ q))
```

[2]We recover regular fold behaviour by running sum over function composition:

```
sum' : (X → Y → Y) → Y → Ar s X → Y
sum' f ϵ a = sum _∘'_ id (map f a) ϵ
```

Any expression r of the type (Reshape s p) comes with a straight-forward action on indices that we denote $\_\langle\_\rangle$ (its definition is omitted). Such a (contravariant) action translates the index within the shape p into the index within the shape s. Given this translation, we can easily define reshape as shown below. Reshape is constructed such that if s and p are related, then p and s are related too. This fact is given by the function rev (its definition is omitted) and it immediately implies that all the actions on indices as well as array reshapes are invertible.

Note that two shapes can be related by Reshape in more than one way, which results in different array reshapes. For example, consider Reshape (ι 5 $\otimes$ ι 4) (ι 5 $\otimes$ ι 4) given by swap or through (split $\cdot$ flat). While the former transposes the array elements, the latter does not.

```
_⟨_⟩ : P p → Reshape s p → P s

reshape : Reshape s p → Ar s X → Ar p X
reshape r a = λ ix → a (ix ⟨ r ⟩)

rev : Reshape s p → Reshape p s
```

From the perspective of category theory, if S is an object then Reshape is a Hom set, where eq is identity and $\_\cdot\_$ is a composition with the expected properties. In the language of containers [4], Ar is a container and Reshape is an inductive subset of cartesian container morphisms.

4 CNN Building Blocks

With the array theory from the previous section we can define the actual primitives that are required for our case study.

4.1 One-dimensional convolution

We start with plus and minus operations for 1-d indices which will be used in the definition of convolution:

```
_⊕_ : Fin m → Fin (1 + n) → Fin (m + n)
zero  ⊕ j = inject-left j
suc i ⊕ j = suc (i ⊕ j)

_⊖_ : (i : Fin (m + n)) (j : Fin m)
    → Dec (∃ λ k → j ⊕ k ≡ i)
```

While the definitions look very innocent, their types carry non-trivial information. Consider $\_\oplus\_$ which is addition of bounded i and j. However, the type says:

$$\frac{i < m \qquad j < 1 + n}{i + j < m + n}$$

This looks a little surprising, but this indeed holds for natural numbers. A reader may convince herself by considering the maximum value that i and j can possibly take. The $\_\oplus\_$ have partial inverses making it possible to define left and right subtraction. We consider left subtraction $\_\ominus\_$. Its type says that there exists a decision procedure for finding k of type Fin $(1 + n)$ together with the proof that k is an inverse. In some sense Dec is similar to Maybe type, except it forces one to prove why the value does not exist as opposed to just returning nothing. This happens to be very useful, as it is really easy to introduce off-by-one errors otherwise.

With the above definitions we can define convolution for 1-dimensional cases. A side note for mathematically inclined readers. We use the term *convolution* in the way it is used in machine learning. Technically, we compute a cross-correlation, because the array of weights is not flipped. However, in practice this is not a problem, as we assume that weights are stored flipped in memory.

We define a handy shortcut Vec and Ix which are Ar and P for 1-dimensional cases.

```
Vec : ℕ → Set → Set
Vec m X = Ar (ι m) X

Ix : ℕ → Set
Ix m = P (ι m)
```

We introduce the slide_1 primitive that selects a $(1 + n)$-element vector from the $(m + n)$-element vector starting at the offset i. Then, following [26], we compute m-element array of slides and then sum it up.

```
slide₁ : Ix m → Vec (m + n) X → Vec (1 + n) X
slide₁ (ι i) v (ι j) = v (ι (i ⊕ j))

conv₁ : Vec (m + n) ℕ → Vec m ℕ → Vec (1 + n) ℕ
conv₁ a w = sum (zipWith _+_) (K 0) (λ i → map (w i *_) (slide₁ i a))
```

4.2 Generalisation

We want to define convolution for arrays of higher ranks. The first thing to do is to express $m + n$ and $1 + n$ where m and n become arbitrary shape trees. In case of addition, we need a witness that both arguments have the same tree structure. If they do, we can simply add their nodes point-wise. We define the three-way relation $\_+\_\approx\_$ that combines the witness and the action. That is, the type $p + q \approx r$ says that p and q have the same tree structure and that q is a point-wise addition of p and q. We introduce a similar relation $\text{suc}\_\approx\_$ for $1 + n$ case, and we introduce the relation $\_*\_\approx\_$ that witnesses point-wise multiplication that will be needed for blocking.

```
data _+_≈_ : S → S → S → Set where
  ι   : ι m + ι n ≈ ι (m + n)
  _⊗_ : s + q ≈ u → p + r ≈ w
      → s ⊗ p + q ⊗ r ≈ u ⊗ w

data suc_≈_ : S → S → Set where
  ι   : suc (ι n) ≈ ι (suc n)
  _⊗_ : suc s ≈ u → suc p ≈ w
      → suc (s ⊗ p) ≈ u ⊗ w

data _*_≈_ : S → S → S → Set where
  ι   : ι m * ι n ≈ ι (m * n)
  _⊗_ : s * q ≈ u → p * r ≈ w → (s ⊗ p) * (q ⊗ r) ≈ u ⊗ w
```

With these relations in place, how do we define generalised convolution? One possible way is to use the sum approach where we recurse over the shape tree and perform one operation at a time. However, there is a good point made in [26] that we can shift the shape recursion into index computation. Therefore we define $\_\oplus_p\_$ and $\_\ominus_p\_$ which generalise $\_\oplus\_$ and $\_\ominus\_$ for higher ranks. Once again, Dec type forces $\ominus_p$ to justify the cases when the inverse does not exist.

```
_⊕ₚ_ : P s → P u → suc p ≈ u → s + p ≈ r → P r
(ι i       ⊕ₚ ι j)       ι       ι       = ι (i ⊕ j)
((a ⊗ a₁) ⊕ₚ (b ⊗ b₁)) (s ⊗ s₁) (p ⊗ p₁) = (a ⊕ₚ b) s p ⊗ (a₁ ⊕ₚ b₁) s₁ p₁

_⊖ₚ_ : (i : P r) (j : P s) (su : suc p ≈ u) (sp : s + p ≈ r) → Dec (∃ λ k → (j ⊕ₚ k) su sp ≡ i)
```

We do not show the implementation of the $\ominus_p$, but it very much follows the structure of $\oplus_p$: we apply $\ominus$ on the leaves and we recurse on the product shape with a little bit plumbing to construct the proof of (non-)existence of the inverse.

Our generalised slide looks very much the same as its 1-dimensional counterpart. All the difference lies in the index computation. We also introduce a section of the slide that we call backslide which embed a $(1 + p)$-dimensional array into a $(s + p)$-dimensional one at offset i using some the

provided default element *def*.

```
slide : P s → s + p ≈ r → Ar r X → suc p ≈ u → Ar u X
slide i pl a su j = a ((i ⊕ₚ j) su pl)

backslide : P s → Ar u X → suc p ≈ u → (def : X) → s + p ≈ r → Ar r X
backslide i a su def pl j with ((j ⊖ₚ i) su pl)
... | yes (k , _) = a k
... | _           = def
```

4.3 Remaining primitives

In the rest of this section we implement the remaining CNN-specific primitives. We are going to use the builtin Float type that we call ℝ so that we can run our specification with concrete values. However, all we require from R is a set of standard arithmetic operations. Therefore, ℝ can be abstracted out as a parameter.

Generalised convolution is given by conv, and it is almost identical to its 1-dimensional counterpart (except it used slide instead of slide_1). The mconv runs u convs adds biases to each of them from the array b.

```
conv : s + p ≈ r → Ar r ℝ → Ar s ℝ → suc p ≈ u → Ar u ℝ
conv sp a w su = sum (zipWith _+_) (K 0.0) λ i → map (w i *_) (slide i sp a su)

mconv : s + p ≈ r → (inp : Ar r ℝ) (w : Ar (u ⊗ s) ℝ) (b : Ar u ℝ)
      → suc p ≈ q → Ar (u ⊗ q) ℝ
mconv sp inp w b su = unnest λ i → map (b i +_) (conv sp inp (nest w i) su)
```

The logistic function computes $1/(1 + e^{-x})$ for every element in the array.

```
logistic : Ar s ℝ → Ar s ℝ
logistic = map λ x → 1.0 ÷ (1.0 + e^ (- x))
```

Average Pooling. One of the steps of the machine learning algorithm is average pooling which splits an array into sub-blocks and computes the average for every such block. While this sounds almost trivial, implementing this generally is quite tricky. In the proposed framework it is not straight-forward to block an array of shape $(\iota\ 12 \otimes \iota\ 12)$ into an array of shape $((\iota\ 6 \otimes \iota\ 6) \otimes (\iota\ 2 \otimes \iota\ 2))$. The difficulty is in preserving local neighbourhood within the blocks (for example it would be wrong to flatten the array and then reshape it into the desired shape as the local neighbourhood will be lost). At the same time, it would be inconvenient to block arrays beforehand as this does not go well with slides. Therefore we introduce the mechanism to block and unblock arrays of shape $(s * p)$ and $(s \otimes p)$.

The key to array blocking is a reshaping operation that turns arrays of shape $((s \otimes p) \otimes (q \otimes r))$ into arrays of shape $((s \otimes q) \otimes (p \otimes r))$, by swapping p and q. We express a Reshape relation, and as it follows from the type this reshape is a self inverse, as can be also seen from the theorem below:

```
rblock : Reshape ((s ⊗ p) ⊗ (q ⊗ r)) ((s ⊗ q) ⊗ (p ⊗ r))
rblock = assocl • eq , (assocr • swap , eq • assocl) • assocr

rblock-selfinv : ∀ i → i ⟨ rev (rblock {s} {p} {q} {r}) ⟩ ≡ i ⟨ rblock ⟩
rblock-selfinv ((i ⊗ j) ⊗ (k ⊗ l)) = refl
```

With this primitive we define block and unblock as follow:

```
block : s * p ≈ q → Ar q X → Ar (s ⊗ p) X
block ι           = reshape split
block (l ⊗ r)     = reshape rblock ∘ unnest ∘ block l ∘ map (block r) ∘ nest

unblock : s * p ≈ q → Ar (s ⊗ p) X → Ar q X
unblock ι         = reshape flat
unblock (l ⊗ r) = unnest ∘ unblock l ∘ map (unblock r) ∘ nest ∘ reshape rblock
```

We specialise average pooling to the 2-dimensional case, that is needed in our running example.

```
avgp₂ : (m n : ℕ) → Ar (ι (m ℕ.* 2) ⊗ (ι (n ℕ.* 2))) ℝ → Ar (ι m ⊗ ι n) ℝ
avgp₂ m n a = map ((_÷ fromℕ 4) ∘ sum _+_ 0.0) (nest $ block (ι ⊗ ι) a)
```

We are now ready to provide the implementation of the forward part of the CNN as follows. The *inp* argument is the image of a hand-written digit, all the other arguments are weights, and the function returns the 10-element vector with probabilities which digit that is.

```
forward : (inp  : Ar (ι 28 ⊗ ι 28) ℝ) → (k₁ : Ar (ι 6 ⊗ (ι 5 ⊗ ι 5)) ℝ)
        → (b₁ : Ar (ι 6) ℝ)            → (k₂ : Ar (ι 12 ⊗ (ι 6 ⊗ (ι 5 ⊗ ι 5))) ℝ)
        → (b₂ : Ar (ι 12) ℝ)           → (fc : Ar (ι 10 ⊗ (ι 12 ⊗ (ι 1 ⊗ (ι 4 ⊗ ι 4)))) ℝ)
        → (b  : Ar (ι 10) ℝ)           → Ar (ι 10 ⊗ (ι 1 ⊗ (ι 1 ⊗ (ι 1 ⊗ ι 1)))) ℝ
forward inp k₁ b₁ k₂ b₂ fc b = let
  c₁ : Ar (ι 6 ⊗ (ι 24 ⊗ ι 24)) ℝ
  c₁ = logistic $ mconv (ι ⊗ ι) inp k₁ b₁ (ι ⊗ ι)

  s₁ : Ar (ι 6 ⊗ (ι 12 ⊗ ι 12)) ℝ
  s₁ = unnest $ map (avgp₂ 12 12) (nest c₁)

  c₂ : Ar (ι 12 ⊗ (ι 1 ⊗ (ι 8 ⊗ ι 8))) ℝ
  c₂ = logistic $ mconv (ι ⊗ (ι ⊗ ι)) s₁ k₂ b₂ (ι ⊗ (ι ⊗ ι))

  s₂ : Ar (ι 12 ⊗ (ι 1 ⊗ (ι 4 ⊗ ι 4))) ℝ
  s₂ = unnest $ map (unnest ∘ map (avgp₂ 4 4) ∘ nest) (nest c₂)

  r = logistic $ mconv (ι ⊗ (ι ⊗ (ι ⊗ ι))) s₂ fc b (ι ⊗ (ι ⊗ (ι ⊗ ι)))
  in r
```

5 Embedded DSL

Any implementation of automatic differentiation has to decide which operations are supported. Surely, it does not make sense to compute derivatives of a function that opens a file. This choice, no matter how it is implemented, can be seen as a definition of an embedded language. Once we accept to identify an embedded language, the idea of embedding it in a way that facilitates extraction actually appears rather naturally and thus advances the approach that we propose in this paper.

Coming back to our example, we have to choose the primitives that the embedded language should support. They need to be sufficient to express AD as well as to define CNNs. The main trade-off here is the choice of the level of abstraction of these primitives: low-level primitives are easier to differentiate, but they make the overall expressions more complex which also adds to the challenge of optimising code. Making this choice is difficult and, most likely, requires quite some adjustment when striving for performance. Here we see a key benefit of the approach we

propose in this paper: the use of a single framework for the embedding, the optimisation, and the extraction makes the implementation comparatively small, allowing for quick adjustments in the level of abstraction, code optimisation and its extraction. We start with a pragmatic approach; we include the primitives that are either shared by the model and the back-end or that can be easily implemented in the back-end language.

It turns out that it is possible to choose the primitives in a way that the derivatives can be expressed in the very same embedded language. While this at first glance may just seem to be just a nice coincidence, it turns out that this has several tangible benefits: high-order derivatives can be computed by the same transformation and we can share all optimisations between the code itself and its derivatives.

As we operate within a dependently-typed proof-assistant, we can easily make our embedded language well-scoped and intrinsically typed (shaped in our case). Our context Ctx is a snoc-list of shapes where each shape has a tag indicating whether it is an index or an array. We use de Bruijn variables given by the relation _∈_ in the usual way. We also define variables v_1, v_2, *etc.* by iteratively applying v_s to v_0 (definition not shown).

```
data IS : Set where
  ix : S → IS
  ar : S → IS

data Ctx : Set where
  ϵ   : Ctx
  _▹_ : Ctx → IS → Ctx

data _∈_ : IS → Ctx → Set where
  v₀ : is ∈ (Γ ▹ is)
  vₛ : is ∈ Γ → is ∈ (Γ ▹ ip)
```

Note that while our contexts are non-dependent (*i.e.* the shapes do not depend on the terms), we use non-trivial shape dependencies within the constructors. The embedded language does not have a notion of shape as a value, therefore all the shape dependencies are handled by Agda, keeping our language simply typed (shaped). This separation is very helpful when it comes to writing embedded programs.We start with two helper definitions: a singleton shape that we call unit and the type for binary operations that we support (for now only addition and multiplication).

```
unit : S
unit = ι 1

data Bop : Set where
  plus mul : Bop
```

The embedded language E includes: variables var; constants 0 and 1 given by zero and one correspondingly; three flavours of array constructor/eliminator pairs given by imap_s/sel_s, imap/sel and imapb/selb; summation sum; conditional zero-but where the predicate is fixed to equality of two indices and the else branch is zero; slide and backslide exactly as described before; and numerical operations. The latter includes logistic, plus and multiplication, division by a constant scaledown, and unary minus. The definition of the embedded language E follows. We also introduce the syntax for infix plus and multiplication denoted ⊞ and ⊠ correspondingly.

```
data E : Ctx → IS → Set where
  var    : is ∈ Γ → E Γ is
  zero   : E Γ (ar s)
  one    : E Γ (ar s)

  imapₛ  : E (Γ ▹ ix s) (ar unit) → E Γ (ar s)
  selₛ   : E Γ (ar s) → E Γ (ix s) → E Γ (ar unit)

  imap   : E (Γ ▹ ix s) (ar p) → E Γ (ar (s ⊗ p))
  sel    : E Γ (ar (s ⊗ p)) → E Γ (ix s) → E Γ (ar p)

  imapb  : s * p ≈ q → E (Γ ▹ ix s) (ar p) → E Γ (ar q)
  selb   : s * p ≈ q → E Γ (ar q) → E Γ (ix s) → E Γ (ar p)
```

```
  sum       : E (Γ ▹ ix s) (ar p) → E Γ (ar p)
  zero-but  : E Γ (ix s) → E Γ (ix s) → E Γ (ar p) → E Γ (ar p)

  slide     : E Γ (ix s) → s + p ≈ r → E Γ (ar r) → suc p ≈ u → E Γ (ar u)
  backslide : E Γ (ix s) → E Γ (ar u) → suc p ≈ u → s + p ≈ r → E Γ (ar r)

  logistic  : E Γ (ar s) → E Γ (ar s)
  bin       : Bop → E Γ (ar s) → E Γ (ar s) → E Γ (ar s)
  scaledown : ℕ → E Γ (ar s) → E Γ (ar s)
  minus     : E Γ (ar s) → E Γ (ar s)

pattern _⊠_ a b = bin mul a b
pattern _⊞_ a b = bin plus a b
```

5.1 Evaluation

We define the interpretation ⟦_⟧ for (E Γ *is*) into the value (Val *is*) in the environment (Env Γ). The values are either arrays or positions of the corresponding shape. Environments for the given context Γ are tuples of values of the corresponding shapes. The lookup function translates variables within the context into variables within the environment.

```
Val : IS → Set                Env : Ctx → Set                lookup : is ∈ Γ → Env Γ → Val is
Val (ar s) = Ar s ℝ           Env ϵ       = ⊤                lookup v₀     (ρ , x) = x
Val (ix s) = P s              Env (Γ ▹ is) = Env Γ × Val is  lookup (vₛ v) (ρ , _) = lookup v ρ
```

In the definition of ⟦_⟧ we wrap the environment argument into double braces. This is an Agda-specific syntax for instance arguments[3] which behave similarly to hidden arguments, but they have a more powerful resolution algorithm. As a result we can omit mentioning the environment in recursive calls when it is passed unchanged.

```
⟦_⟧ : E Γ is → ⦃⦃ Env Γ ⦄⦄ → Val is
⟦ var x          ⟧ ⦃⦃ ρ ⦄⦄ = lookup x ρ
⟦ zero           ⟧ ⦃⦃ ρ ⦄⦄ = K 0.0
⟦ one            ⟧ ⦃⦃ ρ ⦄⦄ = K 1.0
⟦ imapₛ e        ⟧ ⦃⦃ ρ ⦄⦄ = λ i → ⟦ e ⟧ ⦃⦃ ρ , i ⦄⦄ (ι (# 0))
⟦ selₛ e e₁      ⟧ ⦃⦃ ρ ⦄⦄ = K $ ⟦ e ⟧ ⟦ e₁ ⟧
⟦ imap e         ⟧ ⦃⦃ ρ ⦄⦄ = unnest λ i → ⟦ e ⟧ ⦃⦃ ρ , i ⦄⦄
⟦ sel e e₁       ⟧ ⦃⦃ ρ ⦄⦄ = nest ⟦ e ⟧ ⟦ e₁ ⟧
⟦ imapb m e      ⟧ ⦃⦃ ρ ⦄⦄ = CNN.unblock m $ unnest λ i → ⟦ e ⟧ ⦃⦃ ρ , i ⦄⦄
⟦ selb m e e₁    ⟧ ⦃⦃ ρ ⦄⦄ = nest (CNN.block m ⟦ e ⟧) ⟦ e₁ ⟧
⟦ zero-but i j e ⟧ ⦃⦃ ρ ⦄⦄ = if ⌊ ⟦ i ⟧ ≟ₚ ⟦ j ⟧ ⌋ then ⟦ e ⟧ else K 0.0
⟦ sum e          ⟧ ⦃⦃ ρ ⦄⦄ = Array.sum (zipWith _+_) (K 0.0) λ i → ⟦ e ⟧ ⦃⦃ ρ , i ⦄⦄
⟦ e ⊞ e₁         ⟧ ⦃⦃ ρ ⦄⦄ = Array.zipWith _+_ ⟦ e ⟧ ⟦ e₁ ⟧
⟦ e ⊠ e₁         ⟧ ⦃⦃ ρ ⦄⦄ = Array.zipWith _*_ ⟦ e ⟧ ⟦ e₁ ⟧
⟦ slide i pl e su ⟧ ⦃⦃ ρ ⦄⦄ = Array.slide ⟦ i ⟧ pl ⟦ e ⟧ su
⟦ backslide i e su pl ⟧ ⦃⦃ ρ ⦄⦄ = Array.backslide ⟦ i ⟧ ⟦ e ⟧ su 0.0 pl
⟦ scaledown n e  ⟧ ⦃⦃ ρ ⦄⦄ = Array.map (_÷ fromℕ n) ⟦ e ⟧
```

[3]For more details on instance arguments see: https://agda.readthedocs.io/en/v2.6.3/language/instance-arguments.html

```
⟦ minus e     ⟧ {{ ρ }} = Array.map (-_) ⟦ e ⟧
⟦ logistic e  ⟧ {{ ρ }} = CNN.logistic ⟦ e ⟧
```

With the above definition we can better explain the choices of language constructors. The most important question to clarify is why do we have three array constructors/eliminators. As the only conceptual datatype of our language is an array (of some shape), we do not have any direct way to talk about array elements. Therefore, we model the type of array elements (scalars) as arrays of a singleton shape. As can be seen, scalar selection sel_s returns a singleton array (application of K) where all the element(s) are equal to the element we are selecting. The corresponding array constructor imap_s makes sure that if we compute s elements of the shape unit, we produce an array of shape s (and not $s \otimes \mathrm{unit}$). This soves the problem of constructing arays from scalars, but how do we construct an array of a product shape? Given that we have an expression in the context ($\Gamma \triangleright \mathrm{ix}\ s \triangleright \mathrm{ix}\ p$), we need to produce an array of $s \otimes p$. There are several ways how to solve this (*e.g.* introducing nest/unnest or projections and pairing on indices), but it is clear that we need something more than just an imap_s. This is the reason to introduce imap/sel pair which operates on arrays of product shapes. As average pooling operates on blocked arrays, we need a construction to express this in E. One could introduce explicit block/unblock, but we merge blocking/unlocking action with imap/sel obtaining imapb/selb. Our sum constructor gets an argument in the extended context which is summation index, so conceptually we generate the values at every summation index before summing these values together. As a result, we only need one instance of sum which makes our expressions a little tidier.

5.2 Weakening and Substitution

As our language has explicit de Bruin variables (as opposed to HOAS [19] approaches), we need the means to do weakening and substitution when we optimise expressions in E. Our language is intrinsically typed(shaped) which makes the definition of both operations challenging. However, this problem has been well-understood, and we adopt the solution from [13]. We only show the basic mechanisms of the definition, for full details refer to [13].

The key structure that gives rise to weakening and substitution is a function that computes the context Γ *without* the variable v (denoted $\Gamma \ / \ v$). Then we define the weakening for variables (wkv) and expressions (wk) that take a variable or expression in the context without the variable v and return this variable or expression in the context where v is present.

```
_/_ : (Γ : Ctx) → is ∈ Γ → Ctx
(Γ ▷ x) / v0    = Γ
(Γ ▷ x) / vs v = (Γ / v) ▷ x
```

```
wkv : (v : is ∈ Γ) → ip ∈ (Γ / v) → ip ∈ Γ
wk  : (v : is ∈ Γ) → E (Γ / v) ip → E Γ ip
```

We give ourselves a nicer syntax for common cases when expressions are lifted into the context with extra one or two variables:

```
↑_ : E Γ is → E (Γ ▷ ip) is
↑_ = wk v0
```

```
↑↑_ : E Γ is → E (Γ ▷ ip ▷ iq) is
↑↑_ = ↑_ ∘ ↑_
```

A prerequisite for substitution is decidable equality of variables which will be also useful during optimisations. The code below is a copy-paste from [13], but we reiterate its wonderfully mind-twisting mechanics here. The relation for variable equality is given by the type Eq which has two constructors. In case variables are equal (eq constructor) they literally have to match. In case variables x and y are different (neq constructor), we would like to know where to find y in the context without x. After that, eq? shows that variable equality is decidable. The substitution sub

explains how to substitute the variable v in the expression e with the expression e_1.

```
data Eq : is ∈ Γ → ip ∈ Γ → Set where
  eq  : {x : is ∈ Γ} → Eq x x
  neq : (x : is ∈ Γ) → (y : ip ∈ (Γ / x))
      → Eq x (wkv x y)

sub : (v : is ∈ Γ) (e : E Γ ip) (e₁ : E (Γ / v) is)
    → E (Γ / v) ip
```

```
eq? : (x : is ∈ Γ) → (y : ip ∈ Γ) → Eq x y
eq? v₀     v₀     = eq
eq? v₀     (vₛ y) = neq v₀ y
eq? (vₛ x) v₀     = neq (vₛ x) v₀
eq? (vₛ x) (vₛ y) with eq? x y
... | eq       = eq
... | neq .x y = neq (vₛ x) (vₛ y)
```

As our context do not encode explicit dependencies between the variables, we can define the operation that swaps two consequent variables at any given position in the context. Similarly to $(\Gamma \,/\, v)$, we define the function SwapAt that computes the context where x and its successor are swapped. Then we define the operation ctx-swap that translates the expression e into the context where x is swapped with its successor.

```
SwapAt : (Γ : Ctx) → is ∈ Γ → Ctx
SwapAt (Γ ▹ is)       v₀       = Γ ▹ is
SwapAt (Γ ▹ ip ▹ is) v₁       = Γ ▹ is ▹ ip
SwapAt (Γ ▹ ip ▹ is) (vₛ (vₛ x)) = SwapAt (Γ ▹ ip) (vₛ x) ▹ is

ctx-swap : (x : is ∈ Γ) (e : E Γ ip) → E (SwapAt Γ x) ip
```

Building Blocks. Now we implement the remaining building blocks in E that are needed to define our CNN. We start with a several convenience functions that wrap imaps and sum such that when we write (Imap $\lambda\ i \to \cdots$), Agda's variable i is mapped to the E's variable v_0.

```
Imapₛ f = imapₛ (f (var v₀))      Imap f = imap (f (var v₀))      Sum f = sum (f (var v₀))
```

The remaining operations are conv, mconv and avgp_2 which can be defined as functions on E as follows.

```
conv : E Γ (ar r) → s + p ≈ r → E Γ (ar s) → suc p ≈ u → E Γ (ar u)
conv f sp g su = Sum λ i → slide i sp (↑ f) su ⊠ Imapₛ λ _ → selₛ (↑↑ g) (↑ i)

mconv : s + p ≈ r → (inp : E Γ (ar r)) (we : E Γ (ar (u ⊗ s))) (b : E Γ (ar u))
      → suc p ≈ w → E Γ (ar (u ⊗ w))
mconv sp inp we b su = Imap λ i → conv (↑ inp) sp (sel (↑ we) i) su ⊞ Imapₛ λ _ → selₛ (↑↑ b) (↑ i)

avgp₂ : ∀ m n → (a : E Γ (ar (ι (m ℕ.* 2) ⊗ ι (n ℕ.* 2)))) → E Γ (ar (ι m ⊗ ι n))
avgp₂ m n a = Imapₛ λ i → scaledown 4 $ Sum λ j → selₛ (selb (ι ⊗ ι) (↑↑ a) (↑ i)) j
```

Note that these definitions are not very different from those found in Section 3. Some operations such as nest and unnest got inlined into E's operators, and all we really have to take care of is weakening of the expressions whenever we go under binders.

6 Automatic Differentiation

We implement automatic differentiation in reverse mode for expressions in E. We focus on reverse mode because it is of most interest in machine learning, and it is more challenging to implement.

We start with a brief introduction of the AD, for much more in-depth explanations refer to [5, 6]. Consider differentiating a function composition consisting of three functions:

$$y = (f \circ g \circ h)\ x$$

rewrite it using temporary variables:

$$\begin{aligned} w_0 &= x \\ w_1 &= h\ w_0 \\ w_2 &= g\ w_1 \\ w_3 &= f\ w_2 = y \end{aligned}$$

The chain rule gives us $\frac{\partial y}{\partial x} = \frac{\partial y}{\partial w_2}\frac{\partial w_2}{\partial w_1}\frac{\partial w_1}{\partial x}$. The difference between the forward and reverse mode lies in the direction that we traverse the chain rule. In forward mode we traverse the chain inside-out, and the revers mode traverses the chain outside-in thus computing recursive relation: $\frac{\partial y}{\partial w_i} = \frac{\partial y}{\partial w_{i+1}}\frac{\partial w_{i+1}}{\partial w_i}$. For our example, we compute $\frac{\partial y}{\partial w_2}$, then $\frac{\partial w_2}{\partial w_1}$ and finally $\frac{\partial w_1}{\partial x}$. While there seem to be no difference for functions of one variable, there is a big difference for functions of n variables as we can compute derivatives of all the non-dependent variables simultaneously. Consider an example of the $z = f\ x\ y = sin(xy + x)$:

$$\begin{aligned} w_0 &= x \\ w_1 &= y \\ w_2 &= w_1 w_2 \\ w_3 &= w_2 + w_0 \\ w_4 &= \sin w_3 = z \end{aligned}$$

We compute the adjoints $\bar{w}_i = \frac{\partial y}{\partial w_i}$ using the following rule. If w_i has successors in the computational graph, we can apply the chain rule as follows:

$$\bar{w}_i = \sum_{j \in succ\ i} \bar{w}_j \frac{\partial w_j}{\partial w_i}$$

For our example:

$$\begin{aligned} \bar{w}_4 &= 1 = \frac{\partial z}{\partial z} \\ \bar{w}_3 &= \bar{w}_4 \cos w_3 \\ \bar{w}_2 &= \bar{w}_3 \cdot 1 \\ \bar{w}_1 &= \bar{w}_2 w_0 \\ \bar{w}_0 &= \bar{w}_3 + \bar{w}_2 w_1 \end{aligned}$$

If we inline all the $\bar{w}_i$ definitions and inspect the values of partial derivatives with respect to x and y we obtain expected results: $\frac{\partial z}{\partial x} = \cos(xy + x)(y + 1)$ and $\frac{\partial z}{\partial y} = \cos(xy + x)x$.

In the implementation of the AD for E in some context Γ, we would like to obtain all the partial derivatives with respect to the variables in context Γ. Each partial derivative is itself an expression E in context Γ. Therefore, we need to define a data type for an environment of Γ-many expressions in context Γ. We call this environment Env defined as follows:

```
Env : Ctx → Ctx → Set
Env ϵ       Δ = ⊤
Env (Γ ▷ ar s) Δ = Env Γ Δ × E Δ (ar s)
Env (Γ ▷ ix s) Δ = Env Γ Δ
```

Note that Env only keeps array expressions, as (i) derivatives for indices do not exist; and (ii) we can always make an initial environment by populating all the elements with zeros.

We define several helper operations to manipulate environments: env-zero is an environment where all the values are zeros; update modifies the expression at the v-th position by applying f to it; env-map applies the function f from E to E to all the elements of the environment; and env-zipWith applies the binary function f on two environments point-wise. The types of these helper functions follow. As environments are very similar to lists, the implementation of the above functions are straight-forward.

```
env-zero : Env Γ Δ
update : Env Γ Δ → (v : ar s ∈ Γ) → (f : E Δ (ar s) → E Δ (ar s)) → Env Γ Δ
env-map : ∀ {Γ Δ Φ} → (f : ∀ {s} → E Δ (ar s) → E Φ (ar s)) → Env Γ Δ → Env Γ Φ
env-zipWith : ∀ {Γ Δ Φ Ξ} → (f : ∀ {s} → E Δ (ar s) → E Φ (ar s) → E Ξ (ar s))
              → Env Γ Δ → Env Γ Φ → Env Γ Ξ
```

We define the function ∇ that takes an expression E and the seed which is the multiplier on the left of the chain, and we compute a function from that updates the environment.

```
∇ : E Δ is → (seed : E Δ is) → Env Δ Δ → Env Δ Δ

map-sum : (e s : E (Δ ▹ ix s) ip) → Env Δ Δ → Env Δ Δ
map-sum {Δ} e s δ = env-zipWith {Δ} _⊞_ (env-map {Δ} sum (∇ e s (env-zero {Δ}))) δ

∇ (zero)              s δ = δ
∇ (one)               s δ = δ
∇ (var {ix _} x)      s δ = δ
∇ (var {ar _} x)      s δ = update δ x (_⊞ s)

∇ (imap_s e)          s = map-sum e (sel_s     (↑ s) (var v0))
∇ (imap e)            s = map-sum e (sel       (↑ s) (var v0))
∇ (imapb m e)         s = map-sum e (selb m    (↑ s) (var v0))

∇ (sel_s e i)         s = ∇ e (imap_s     (zero-but (var v0) (↑ i) (↑ s)))
∇ (sel e i)           s = ∇ e (imap       (zero-but (var v0) (↑ i) (↑ s)))
∇ (selb m e i)        s = ∇ e (imapb m    (zero-but (var v0) (↑ i) (↑ s)))

∇ (zero-but i j e)    s = ∇ e (zero-but i j s)
∇ (sum e)             s = map-sum e (↑ s)

∇ (e ⊞ e1)            s = ∇ e s ∘ ∇ e1 s
∇ (e ⊠ e1)            s = ∇ e (s ⊠ e1) ∘ ∇ e1 (s ⊠ e)
∇ (slide i pl e su)   s = ∇ e (backslide i s su pl)
∇ (backslide i e su pl) s = ∇ e (slide i pl s su)

∇ (scaledown x e)     s = ∇ e (scaledown x s)
∇ (minus e)           s = ∇ e (minus s)
∇ (logistic e)        s = ∇ e (s ⊠ logistic e ⊠ (one ⊞ minus (logistic e)))
```

Let us now walk through the cases. Derivative of constants (zero and one) is zero, so nothing needs to be updated in the environment. Index variables are not stored in the environment, so no updates are needed either. If we differentiate the variable x with some seed s, we update the x-th position in the environment by adding s to it. Differentiation of imaps proceeds as follows: we recursively

apply ∇ to e (in the context $\Gamma \triangleright (\text{ix } p)$) with the element of the original seed s selected at the top variable. This gives us the environment in the extended context, then we map sum to every element of the environment to accumulate the derivatives at every index. When differentiating selections we recurse on the array we are selecting from with the seed that is zero everywhere except the index we were selecting at. Differentiating conditional is straight-forward, as i and j must be in the context, we can simply differentiate e with the condition on seed. If indices were equal, we will compute the update, otherwise we will differentiate with seed zero which has no effect. As we are operating in a total language, there is no need to worry about pulling the expression out of conditional. The argument of sum lives is in the extended context, so we apply the same rules as for the imap family, except we propagate the original seed to all the summands. Addition and multiplication rules are straight-forward application of differentiation rules. The slide/backslide pair forms a satisfying ∇-symmetry. Finally, scaledown, minus and logistic follow the rules of differentiation.

6.1 Optimisation

Our algorithm often delivers expressions that are not computationally efficient. While we can hope for the backend to take care of this, it is relatively easy to implement a number of rewriting rules prior to extraction. We constructed E such that no computation is happening in the shape or context positions. As a result, dependent pattern-matching is always applicable on E expressions, and our optimisations can be formulated very concisely. We omit constant-folding like rewrites such as addition with zero and multiplication by one and focus on less trivial cases that have to do with selections a+nd sum. Consider the snippet of the optimiser for sel_s and sum.

```
opt : E Γ is → E Γ is
opt (sel_s e e₁) with opt e | opt e₁
... | zero          | i = zero
... | one           | i = one
... | imap_s e      | i = sub v₀ e i
... | bin op a b    | i = bin op (sel_s a i) (sel_s b i)
... | sum e         | i = sum (sel_s e (↑ i))
... | zero-but i j a | k = zero-but i j (sel_s a k)
... | a             | i = sel_s a i

opt (sum e) with opt e
... | zero          = zero
... | imap a        = imap     (sum (ctx-swap v₁ a))
... | imap_s a      = imap_s   (sum (ctx-swap v₁ a))
... | imapb m a     = imapb m (sum (ctx-swap v₁ a))
... | zero-but (var i) (var j) a with eq? v₀ i | eq? v₀ j
... | eq    | eq    = sum a
... | neq _ i' | eq = sub v₀ a (var i')
... | eq    | neq _ j' = sub v₀ a (var j')
... | neq _ i' | neq _ j' = zero-but (var i') (var j') (sum a)
opt (sum e) | a = sum a
– ...
```

Selection into zero and one is zero and one, as our constants are shape-polymorphic. Selection into an imap_s is evaluation of the imap_s body at the given index (this is an array version of the β-rule).

Selection from the binary operation is a binary operation of selections. Selection into sum is the sum of selections. Selection into conditional is the same as conditional over selection. Summing zero is zero. Summing s-many p-shaped arrays is the same as computing the sum of i-th index of every array for all p indices. If we have a sum of the conditional with the predicate is the equality of indices i and j and we know that i and j are variables, we can compare the index variable of the sum with i and j. If they match, then conditional will be triggered at every iteration so it can be removed. If only one of them match, and we are comparing variables of the same shape, there will be exactly one case (for non-empty shapes) where this conditional will be triggered. Therefore, all the iterations except the one at the non-matching variable will turn to zero, and we can simply return the expressions substituted at this variable. If the shape of the index variables is empty, we are in the absurd case, as we cannot possibly create an element of an empty type. Finally, if none of the variables match, the iteration within the sum do not affect the result of the predicate — it will be either true or false for all the iterations. Therefore, we can lift the conditional outside of the sum.

Let us observe optimisation effects when computing derivatives of the scalar dot-product defined as follows.

```
dotp : E Γ (ar s) → E Γ (ar s) → E Γ (ar unit)
dotp a b = Sum λ i → sel_s (↑ a) i ⊠ sel_s (↑ b) i
```

We define the context C where two top variables are of 5-element vector shape and the last variable (v_2) is of scalar shape. We bind these variables to Agda variables for convenience.

```
C = ϵ ▹ ar (ι 1)          ▹ ar (ι 5)     ▹ ar (ι 5);
      seed = var v₂ ; a = var v₁ ; b = var v₀
```

We compute the derivatives of dotp a b with seed seed and we inspect the a-th position in the returned environment that we call ∂a. Then we repeatedly apply opt (three times) to ∂a and save it in ∂a'. We force Agda to verify that the content of the variables is as follows:

```
non-opt  : ∂a  ≡ (Sum λ i → zero ⊞ Imap_s λ j → zero-but j (↑ i) (↑↑ seed ⊠ sel_s (↑↑ b) (↑ i))) ⊞ zero
with-opt : ∂a' ≡ Imap_s λ i → (↑ seed ⊠ sel_s (↑ b) i)
```

As it can be seen, ∂a sums-up the arrays, where only one element is non-zero at every iteration. Such a computation is highly inefficient when executed directly, as it needs to compute all the inner arrays before summing them up. However, the optimised version correctly rewrites ∂a into imap that multiplies the *seed* by b, which is the expected answer. This reduces complexity of the expression form squared to linear.

6.2 Extraction

Extraction from E into SaC translates constructors of E into corresponding SaC expressions or function calls. The translation starts with a definition of an environment (SEnv Γ) that assigns SaC variable names to all positions in Γ. The assumption here is that when we compile expressions in context Γ, variable names of the corresponding shapes are available in SaC.

Next, we have to take care of shapes. Array shapes in E are binary trees, but array shapes in SaC are 1-dimensional arrays (flattened binary trees). When some expression in E is of product shape, we usually have to supply left or right subshapes of the product to SaC. These are always available through implicit arguments of E constructors. Assuming that by the time we come to extraction, all the E shapes are constants, we can always generate shape expressions in SaC. This is implemented in show-shape. Relaxing the assumption about constant shapes is possible but requires extension of E so that we can always bind the shapes used in E to some expressions in SaC.

We also need a source of fresh variables so that we can generate indices for imap expressions. We define a stateful function iv that generates a fresh index variable.

Extraction is given by to-sac that translates the expression e in the environment ρ. The function is stateful so that we can generate fresh variables when needed.

The definitions of SEnv, iv, show-shape, and to-sac follow.

```
SEnv : Ctx → Set
SEnv ϵ       = ⊤
SEnv (Γ ▹ is) = SEnv Γ × String
```

```
iv : S → State ℕ String
iv s = do v ← get
          modify suc
          return $ printf "x%u" v
```

```
show-shape : S → String
show-shape s = printf "[%s]"
                 $ intersperse ", "
                 $ go s
  where
    go : S → List String
    go (ι x)     = show-nat x :: []
    go (s ⊗ p) = go s ++ go p
```

```
to-sac : (e : E Γ is) → (ρ : SEnv Γ) → State ℕ String
to-sac (imap {s = s} e) ρ = do
  i ← iv s
  b ← to-sac e (ρ , i)
  return $ printf "{ %s -> %s | %s < %s }"
                  i b i (show-shape s)
to-sac (sel e e₁) ρ =
  printf "(%s)[%s]" <$> to-sac e ρ ⊛ to-sac e₁ ρ
_ ...
```

6.2.1 SaC Primitives. As can be seen from the two cases of to-sac, the extraction process is not complicated. In essence, we define a small snippet of SaC code for each E constructor. Consider the imap/sel family from the code snippet. The imap constructor maps directly to SaC's tensor comprehensions [23] expressed as: `{ iv -> e | iv < s }`. This expression constructs arrays by evaluating `e` for every array non-negative index vector `iv` whose components are element-wise smaller than the shape `s`. The shape of the resulting array is concatenation of `s` and whatever the shape of `e` is. Selections sel correspond to the built-in array selection using C-like syntax `e[iv]` where `e` is the array we are selecting from and `iv` is the index vector. Shape constraints are exactly as in E: if `e` is of shape `s ++ p`, and `iv` is bounded by `s` then `e[iv]` is of shape `p`.

Scalar versions of imap/sel require a little wrapping. For imap_s we generate a tensor comprehension that selects inner expressions (they are 1-element vectors) at zero-th position. For sel_s we make selection into an array and we wrap the result in a 1-d vector:

```
#define IMAPS(iv, e, shp) \
  {iv -> (e)[[0]] | iv < shp}
```

```
inline float[1]
sels(float[d:shp] x, int[d] iv)
{
  return [x[iv]];
}
```

When translating (imap_s { s } e) we pick a fresh index variable `iv`, then we translate e (in the environment extended with `iv`) into `e'` and we generate `IMAPS(iv, e', shp)`, where `shp` is a translation of `s`. On the side of SaC we expand this macro as shown above. We could have expanded this macro on the Agda side, but this abstraction makes it possible to make adjustments in the generated code without running Agda. We map sel_s into the `sels` function. Consider the type of `sels` which uses the recently added feature of SaC that makes it possible to encode shape constraints in types [1]. While these constraints are potentially checked at runtime, they are very useful for readability and they provide some confidence about the generated code. The meaning of the type `float[d:shp]` is that it is an array of base type `float` of rank `d` and shape `shp`. When a

variable of the same name is used within different arguments, it automatically triggers the equality constraint between the corresponding ranks/shapes.

Blocking. Implementation of selb/imapb pair relies on the notion of blocking, so we introduce the analogue to block/unblock functionality in SaC as follows:

```
inline float[n:s,n:p]
block(float[n:sp] x, int[n] p)
     | all(s*p == sp)
     , all(p   >= 0)
{
  return { iv -> tile(p, iv * p, x)
         | iv < sp / p};
}
```

```
inline float[n:sp]
unblock(float[n:s,n:p] a, int[n] p)
       | all(s*p == sp)
       , all(p   >= 0)
{
  return { iv -> a[(iv / p) ++ mod (iv, p)]
         | iv < s*p};
}
```

The type `float[n:s,n:p]` denotes an array of the shape `s ++ p` where `s` and `p` are of length `n`. This is a product shape in terms of our array theory. As `sp` is just a variable that is not related to `s` or `p`, we add two constraints (expressions behind the bar after the function definition) saying that: (i) `sp` is a point-wise product of `s` and `p`; (ii) all the elements of the `p`-shape are greater than zero. Keep in mind that these are potential runtime constraints, they may be proved or flagged as disproved during compilation but they do not provide a static guarantee. The implementation of block uses the `tile` operation from the standard library of SaC. It selects a sub-array of the given shape at the given position. In `unblock` we use a division and a modulo operation to remap the indices. When translating selb, we simply select into `block`-ed array. When translating imapb, we use the tensor comprehension as in case of imap to compute blocked array and then we call `unblock` on it.

Sliding. Slides and backslides are translated into calls to the following SaC functions:

```
inline float[d:n1]
slide(int[d] i, float[d:mn] x, int[d] n)       | all(n1        == n + 1)
                                               , all(n + 1 + i <= mn)
{
  return { iv -> x[iv + i] | iv < n + 1 };
}

inline float[d:mn]
backslide(int[d] i, float[d:n1] y, int[d] mn)  | all(i < 1 + mn - n1)
{
  return { iv -> y[iv - i] | i <= iv < n1 + i;
           iv -> 0f        |      iv < mn };
}
```

Shape constraints become a little bit involved here because we implicitly reconstruct the proof objects such as $m + n \approx mn$ and suc $n \approx n1$. Otherwise, `slide` selects a sub-array of the shape (n+1) starting at the index `i`. The `backslide` populates the sub-array with the elements of `y` and the second partition of the tensor comprehension specifies that all the other indices evaluate to zero. Translation of slide and backslide maps the arguments one-to-one, additionally providing the n-shape in case of slide and the $(m + n)$ shape in case of backslide.

Summation. When translating (sum {s} e), where e is of shape p (and the index variable within the sum is bounded by s), we map these arguments into the following SaC function:

```
inline float[n:p] sumOuter(float[m:s,n:p] a, int[m] s, int[n] p) {
  return { jv -> sum({iv -> a[iv++jv] | iv < s}) | jv < p };
}
```

We use SaC's builtin `sum` function that sums-up all the elements of the given array.

The rest of the constructions are mapped into regular arithmetic operations that are provided by SaC.

6.3 Local Variables

The framework that we built so far computes derivatives of the variables in the context. This means that for complex expressions in E (such as forward), all the let bindings will be inlined. This is often not desirable both for performance and readability. Here we present a mechanism that introduce local variables and preserves them during AD.

The key data structure that makes it possible to introduce local variables is called Chain which has two constructors. The empty chain consists of the names for all the variables in the context Γ. This represents the case where no local variables have been introduced. The _▹_ constructor takes a chain in context Δ and the array expression of shape p in the same context together with the variable name. This produces the chain in the context extended by two variables. One variable is a place-holder for the expression and the other variable is a placeholder for the derivative of that expression.

```
data Chain : Ctx → Set where
  ϵ   : Sac.SEnv Γ → Chain Γ
  _▹_ : Chain Δ → (String × E Δ (ar p)) → Chain (Δ ▹ ar p ▹ ar p)
```

The computation of the derivative in Chains follows the following simple idea. Consider the chain with two variables a and b in the initial context Γ, and two local variables x and y. Here is what happens when we compute the derivative of some expression e (that may depend on a, b, x, y) with some seed s in the empty δ_0 environment.

a	b	∂x	x	∂y	y	compute $\nabla\ e\ s\ \delta_0$
δ_a	δ_b	-	δ_x	-	δ_y	assign δ_y to ∂y
δ_a	δ_b	-	δ_x	δ_y	δ_y	compute $\nabla\ y_e\ \partial y$
δ'_a	δ'_b	-	δ'_x	δ_y	δ_y	assign δ'_x to ∂x
δ'_a	δ'_b	-	δ'_x	δ_y	δ_y	compute $\nabla\ x_e\ \partial x$
δ''_a	δ''_b	δ'_x	δ'_x	δ_y	δ_y	done

First of all, the computation of e returns the environment δ that can be found in the first line of the table. Then we repeat the following steps while traversing the chain backwards: we copy the y-th position of the δ-environment to the ∂y-th position, and we compute the expression y_e that is assigned to y (xx in this case) with the seed ∂y-th variable. Just to clarify, the seed is the variable ∂y and not its value. Then we repeat the same process for x and potentially all the other remaining local variables (not in this case) until we hit the beginning of the chain.

At the end of the process we obtain an environment where derivatives for a and b are expressed in terms of ∂x and ∂y. The remaining step is to collect the values of ∂x and ∂y which can be found at the corresponding positions in the δ-environment.

Let us consider a small example to see this in action. We start with a little convenience data structure ChainCtx that keeps the shapes and the variable names together. We also define the function ce-split that splits ChainCtx into the context and the environment with variable names in that context:

```
data ChainCtx : Set where
  ϵ : ChainCtx
  _▹_ : ChainCtx → String × S → ChainCtx
```

```
ce-split : ChainCtx → Σ Ctx Sac.SEnv
```

Consider an initial environment of two 5-element vectors a and b; local computations $x = ab$ and $y = xx$; and the generated code when computing derivative of y (var v_0) on the right.

```
test-chain : Chain _
test-chain = let
  Γ , ρ = ce-split (ϵ ▹ ("a" , ι 5) ▹ ("b" , ι 5))
  a = var v₁; b = var v₀
  C₁ = ϵ ⁅ Γ ⁆ ρ ▹ ("x" , a ⊠ b)
  x = var v₀
  C₂ = C₁      ▹ ("y" , x ⊠ x)
  in C₂
```

```
x = (a) * (b);
y = (x) * (x);
ddy = one;
ddx = ((ddy) * (x)) + ((ddy) * (x));
ddb = (ddx) * (a);
dda = (ddx) * (b);
```

Let us convince ourselves that the result is correct. Our expression is $abab = a^2b^2$, and its partial derivatives $\frac{\partial}{\partial a} = 2ab^2$, $\frac{\partial}{\partial b} = 2ba^2$. If we fold the assignments, we get:

$$\begin{aligned} \texttt{dda} &= (x + x)b = (ab + ab)b = 2ab^2 \\ \texttt{ddb} &= (x + x)a = (ab + ab)a = 2ba^2 \end{aligned}$$

Note that computations in x and `ddx` are shared in further computations which was the main goal of introducing this mechanism.

There are two inconveniences in the above implementation that we would like to mention:

(1) There is no restriction on using the placeholders for derivatives in the chain expressions, so in principle, one could write expression in terms of variables and their derivatives. However, this is not being handled and likely to generate bogus terms. If this is a useful feature, it requires more thinking on how exactly it should work. Otherwise it is easy to introduce restrictions that rule out such cases.

(2) If we define variables in the chain that do not contribute to the final expression, we may introduce extra computations. We do not compromise correctness, as all inaccessible terms will get zero value. However, direct execution of the resulting expressions may introduce redundant computations.

Both of these are future work. For now, we make an assumption that placeholders are not used in the expressions and that we do not insert bindings that do not contribute to the final result.

We present the specification of our case study in E using Chain. We start with the context cnn-ctx that contains the `target` digit that is depicted on the image, the input image `inp` and the weights of the network. The definition of the chain is a one-to-one copy of the definition found in Section 4. The only real difference is that we have to take care of maintaining bindings between Agda variables and the variables in E. Fortunately, let expressions in Agda make it possible to shadow the binding, which comes very useful in this case.

```
cnn-ctx : ChainCtx
cnn-ctx = ϵ
        ▹ ("target" , ι 10 ⊗ (ι 1 ⊗ (ι 1 ⊗ (ι 1 ⊗ ι 1))))  − 7
        ▹ ("inp"    , ι 28 ⊗ ι 28)                          − 6
        ▹ ("k₁"     , ι 6 ⊗ (ι 5 ⊗ ι 5))                    − 5
        ▹ ("b₁"     , ι 6)                                  − 4
        ▹ ("k₂"     , ι 12 ⊗ (ι 6 ⊗ (ι 5 ⊗ ι 5)))           − 3
        ▹ ("b₂"     , ι 12)                                 − 2
        ▹ ("fc"     , ι 10 ⊗ (ι 12 ⊗ (ι 1 ⊗ (ι 4 ⊗ ι 4)))) − 1
```

```
        ▹ ("b"          , ι 10)                                  − 0


cnn-chain : Chain _
cnn-chain = let
  Γ , ρ = ce-split cnn-ctx
  inp = var v6; k1 = var v5; b1 = var v4; k2 = var v3; b2 = var v2; fc = var v1; b = var v0
  C1 = ϵ {Γ} ρ ▹ ("c11" , mconv (ι ⊗ ι) inp k1 b1 (ι ⊗ ι));   k2 = ↑↑ k2; b2 = ↑↑ b2; fc = ↑↑ fc; b = ↑↑ b; c11 = var v0
  C2 = C1 ▹ ("c1" , logistic c11);                            k2 = ↑↑ k2; b2 = ↑↑ b2; fc = ↑↑ fc; b = ↑↑ b; c1 = var v0
  C3 = C2 ▹ ("s1" , Imap λ i ⟶ avgp2 12 12 (sel (↑ c1) i)); k2 = ↑↑ k2; b2 = ↑↑ b2; fc = ↑↑ fc; b = ↑↑ b; s1 = var v0
  C4 = C3 ▹ ("c21" , mconv (ι ⊗ (ι ⊗ ι)) s1 k2 b2 (ι ⊗ (ι ⊗ ι)));   fc = ↑↑ fc; b = ↑↑ b; c21 = var v0
  C5 = C4 ▹ ("c2" , logistic c21);                                   fc = ↑↑ fc; b = ↑↑ b; c2 = var v0
  C6 = C5 ▹ ("s2" , Imap λ i ⟶ Imap λ j ⟶ avgp2 4 4 (sel (sel (↑↑ c2) (↑ i)) j));   fc = ↑↑ fc; b = ↑↑ b; s2 = var v0
  C7 = C6 ▹ ("r1" , mconv (ι ⊗ (ι ⊗ (ι ⊗ ι))) s2 fc b (ι ⊗ (ι ⊗ (ι ⊗ ι))));   r1 = var v0
  C8 = C7 ▹ ("r"  , logistic r1)
  in C8
```

7 Performance

One of the goals of this work is to demonstrate that it is possible to formulate the problem in a proof assistant and then pass it on to the other system that can run the algorithm efficiently. In order to substantiate this claim, we compare the running times of the code that we generate from the specification at the end of the Section 6 and the hand-written SaC code from [26]. We are not interested in an exhaustive performance study similar to what is provided in [26]. Instead, we take the version from that paper as reference point and we aim to find out whether we are in the same ballpark.

We take the code from [26], make sure that it runs, and we replace the hand-written CNN training with the Agda-generated one. Our first observation is that both versions produce the same results, and none of the shape constraints that we defined in Section 6.2.1 fired. This means that our code generation is working. Unfortunately, the runtime comparison revealed that our version is about 10× slower than the hand-written SaC version.

We got in touch with the SaC team who provided a lot of support in identifying the causes of the disappointing difference in performance. It turns out that the main culprit has to do with the inability to optimise away selections into tensor comprehensions in a few situations. With-Loop-Folding [21], SaC's mechanism for fusing tensor comprehensions fails to fold tensor comprehensions that are nested and cannot be flattened statically. In simple terms, the expression `{iv -> e(iv)}[jv]` in some situation does not reduce to `e(jv)` which, in our generated code, is essential to match the hand-written performance. As a result, several arrays were created simply to make a single selection into them. The original code never ran into this problem as the hand-written code avoided such patterns. Our E optimiser from Section 6.1 could not help either, because the problem was occurring after the SaC primitives such as slide and block were inlined.

After numerous attempts on altering E to fit SaC requirements and the SaC team trying to implement some of the missing optimisations, on the February 23rd (6 days before the ICFP deadline) we realised that the best runtime we can obtain is still 6× slower than the hand-written code. The compiler is too sensitive to the flavour of the code that we pass to it, and when certain patterns are not recognised, there is very little one can do other than trying to fit those patterns. However, this is not always possible with the generated code. Performance *is* frustrating!

7.1 Generating C

After overcoming the natural instinct to give up, we realised that the real power of the proposed approach lies in the ability to modify any part of the code generation pipeline. This includes swapping the back-end language of choice to something else. Therefore, we decided to try generating C code instead of SaC code.

While C is a canonical low-level language, it has excellent support for multi-dimensional arrays, given that the ranks are statically known. At runtime these arrays are flattened vectors, they do not have to live on the stack, and the language takes care of multi-dimensional indexing.

However, the key difference between the C and SaC is memory management. SaC is a functional language that uses reference counting to automate operations on allocating and freeing memory. In C these decisions are manual, and as we have seen before, excessive memory allocation is detrimental for the runtime. For our use case we avoid memory management problem entirely, by assuming that all the variables in the Chain have to be preallocated, and if we need any temporary array variables when extracting array values, we fail extraction. This way we guarantee that no memory allocation is ever needed.

Meeting such a requirement means that we need to optimise away operations like slide/backslide as they require conceptual array allocation. The same goes for imaps appearing in some of the argument positions. Putting these considerations together, we extended E with the following explicit operations on indices:

```
data E : Ctx → IS → Set where
  div        : s * p ≈ q → (i : E Γ (ix q)) → E Γ (ix s)
  mod        : s * p ≈ q → (i : E Γ (ix q)) → E Γ (ix p)
  ix-plus    : (i : E Γ (ix s)) → (j : E Γ (ix u)) → suc p ≈ u → s + p ≈ r → E Γ (ix r)
  ix-minus   : (i : E Γ (ix r)) → (j : E Γ (ix s)) → s + p ≈ r → suc p ≈ u
               → (e : E (Γ ▸ ix u) (ar q)) → E Γ (ar q)
  ix-minus_r : (i : E Γ (ix r)) → (j : E Γ (ix u)) → s + p ≈ r → suc p ≈ u
               → (e : E (Γ ▸ ix s) (ar q)) → E Γ (ar q)
  – ...
```

The div and mod constructors perform point-wise division or modulo operation on the index i and the shape p. This is needed to express selections into blocked arrays as we have seen in Section 6.2.1. The ix-plus is a point-wise addition of i and j. The ix-minus and ix-minus$_r$ correspond to left and right subtraction from the Section 4. The meaning of these constructors is follows: if j can be subtracted from i (in the sense of existence of inverse to $\oplus_p$ exists) then we evaluate e at that index, otherwise we return zero.

7.1.1 Optimisations. We add the following optimisations to facilitate removal of temporary arrays in the generated code. We show the only ones that we added, all the optimisations we defined before are still valid.

```
opt : E Γ is → E Γ is
opt (sel_s e e₁) with opt e | opt e₁
... | imapb m e     | i = sel_s (sub v₀ e (div m i)) (mod m i)
... | slide i pl a su | k = sel_s a (ix-plus i k su pl)
— | ... as before ...
```

Here we optimise away scalar selections into blocked imaps. Recall that m tells us that we have an array of shape $s * p$, and e computes blocks of shape p. If we are selecting into such a blocked

array at the index i, we know that we are selecting (i/p)-th block, and from that block we are selecting $(i\%p)$ element. Existence of explicit div and mod operations on indices makes it possible to implement this rewrite rule that is again very similar to β-reduction.

```
opt (sum e) with opt e
... | zero-but (var i) (ix-plus (var j) (var k) su pl) a with eq? v₀ i | eq? v₀ j | eq? v₀ k
... | neq _ i' | neq _ j' | eq          = ix-minus   (var i') (var j') pl su a
... | neq _ i' | eq        | neq _ k' = ix-minusᵣ (var i') (var k') pl su a
... | _        | _         | _        = sum (zero-but (var i) (ix-plus (var j) (var k) su pl) a)
— | ... as before ...
```

Here we are dealing with the sum over summation index t where the inner expression is a conditional on indices of the form `i == j + k ? e : 0`. Here we apply the same comparison of index variables as before. If k happens to be the variable t, then overall sum will only add one non-zero element at $(i - j)$-th index, given that this (left) subtraction is possible in the sense of existence of the inverse to $\_\oplus_p\_$ operation defined in Section 4.2. The same happens when the summation index t is equal to j, we only need to consider $(i - k)$-th element given that this (right) subtraction is possible. One could cover other cases where t is equal to i, or when i and $j + k$ are swapped, but these are not occurring in our running example.

```
opt (scaledown x e) with opt e
... | sum a = sum (scaledown x a)
— | ... as before ...
```

Finally, here is a rule that looks very innocent in the high-level language, yet becomes of importance in the low-level one. The rule says that if we are summing the array and then dividing it by a constant, we should move division inside the summation. The reason for this rewrite rule being important is when the result of the sum is non-scalar, we need to create a temporary array, before scaling down all its elements. A language with first class arrays can obviously take care of such minor details, but in C we have to be explicit about it.

7.1.2 Code Generation. Due to space limitations, we only consider the basic mechanisms we used in the code generator, all the code is available in supplementary materials. We use heap-allocated multi-dimensional arrays that can be defined as follows:

```
float(*k1)[6][5][5] = malloc(sizeof(*k1));
```

This ensures that `k1` is represented as a continuous region of memory of size $6 * 5 * 5$ floats. When such arrays are indexed (*e.g.* `(*k1)[i][j][k]`), the indices are translated into a single offset into the continuous memory. Therefore, there is no pointer chasing which makes this approach efficient at runtime. As C uses row-major order to compute the offsets, we do obtain partial array selections on the left, *e.g.* `(*k1)[i]` is a 5×5 array that can be further indexed or passed to `sizeof` that correctly identifies the size of this subarray. Surely, this is a pointer into the `k1` array, so all the modifications to `(*k1)[i]` will modify `k1`. As a great convenience feature, C compiler tracks the ranges of the indices and produces warnings in cases when it figures out that ranges of indices and the array we are indexing do not match.

Whenever we translate some e in E into C, we have to provide a storage where e has to be written to. In case of compiling the Chain every local variable becomes such a storage for the bound expression. Therefore, our extractor always has a result variable as an argument.

For example, let us consider an expression $a \boxplus a$ of shape $(\iota\ 5 \otimes \iota\ 5)$, where a is mapped to the C variable `float (*a)[5][5]` that is written to the result variable `float (*r)[5][5]`. Here is the code that we generate:

```
for (size_t x1_1 = 0; x1_1 < 5; x1_1++) {
  for (size_t x1_2 = 0; x1_2 < 5; x1_2++) {
    (*r)[x1_1][x1_2] = ((*a)[x1_1][x1_2] + (*a)[x1_1][x1_2]);
  }}
```

We started with checking that $a \boxplus a$ is a *selectable* expression. This means that we can always generate expression at the given index. As we know that the shape of $a \boxplus a$ is $(\iota\ 5 \otimes \iota\ 5)$, we need to generate a loop nest of that shape that assigns where we assign the expression at the given index to the result at the given index.

We need to distinguish whether we are writing into the result or adding into it as in cases when dealing with sum. Consider the code that is generated for (sum (sel_s (a (var v_0))) where we are adding all the elements of the array a into result variable `float (*r)[1]`:

```
for (size_t x2_1 = 0; x2_1 < 5; x2_1++) {
  for (size_t x2_2 = 0; x2_2 < 5; x2_2++) {
    for (size_t x3_1 = 0; x3_1 < 1; x3_1++) {
      (*r)[x3_1] += (&(*a)[x2_1][x2_2])[x3_1];
    }}}
```

Two things are happening here, first we generate += assignment and we make an implicit assumption that resulting variables are initialised to zero. In the extractor, additionally to the resulting variable we track whether we need to do an assignment or assignment with addition. Secondly, while a is two-dimensional, we have three-dimensional loop nest. The latter comes from the representation of scalars as 1-element vectors. When we resolved the two-dimensional summation index `x2`, we know that we need to assign into the object of shape $(\iota\ 1)$, but the left-hand-side is a scalar (float). The trick here is that in C we can always turn scalars into 1-element vectors by simply taking the address of the scalar. This is why we have this 1-iteration for-loop over `x3_1` that will be immediately optimised away by the C compiler.

Finally, when we it comes to the operation on indices, such as addition, subtraction, division or modulo, we generate the corresponding operation on the individual loop indices.

Remaining details of the code generation take care of traversing through the structure of E with some plumbing that has to do with generating loop-nests around expressions and checking that they are selectable.

7.1.3 Running the Generated C Code. In order to run the generated C code we translate the boilerplate code from SaC to C. While doing so, we made sure that our code can be run in parallel. While the SaC compiler does this automatically, there is one obvious loop that requires parallelisation which is computation of the batch. When we train the CNN, we take a batch of images and the weights and we compute gradients for those weights per every image. After that we average all the gradients in the batch, and we update the weights, after all the batch is processed. Clearly, all the gradient computations in the batch can run in parallel. We achieve this by organising the batch loop such that all the gradients are stored in a separate memory region, and we parallelise this loop using OpenMP annotations.

We verify that the code that we generate compute the same results as the hand-written SaC code. Then we replicate the experiment from the [26] using 40 epochs, 100 images in the batch, and feeding 10000 training images. We run the experiment on the 18-core 13th Gen Intel(R) Core(TM) i5-13600K machine using sac2c version `1.3.3-MijasCosta-1161-gb543c` and the GCC compiler version `12.2.0`. The first thing that we learn is that our generated C code is sensible (factor of 3 running time) to the compilation flags that we enable. We identified the set of flags that when passed to both compilers[4], the runtime at the largest number of cores are 11s for the hand-written SaC implementation and 13.5s for the generated C code, with very little variance. This 20% difference is

[4]SaC compiler generates C code, so we can control what flags it uses when compiling it.

orthogonal to parallel execution, as it is also observed when running the code on a single core. The set of flags has to do with floating point operations: `-fno-signed-zeros` ignores the distinction between negative and positive zeroes that is given by IEEE 754 standard, allowing to reduce (-0.0*x) to 0.0; `-fno-math-errno` does not set errno after calling math functions; `-fno-trapping-math` and `-fassociative-math` make sure that we can assume associativity of floating point operations which does not hold according to the IEEE 754.

The main performance difference comes from the fact that compiled SaC code uses less intermediate arrays, significantly reducing the number of memory writes. There are numerous ways how to improve the performance of the generated C code, but for the purposes of this paper we consider that getting within 20% of the hand-written SaC code is sufficient evidence for our hypothesis that the two-languages approach seems viable for achieving proved correctness and performance. We have automatic differentiation in the safe environment that generates the C code that runs almost as fast as the hand-written SaC code.

8 Conclusions

The paper demonstrates a technique of developing high performance applications with strong correctness guarantees. The key insight lies in using a proof assistant in cooperation with a high-performance language of choice. This gives a clear separation of concerns that is very difficult to achieve within a single language. The proof assistant is used to design a specification, prove all the correctness invariants of interest and performs an extraction into a high-performance language of choice. This may take a non-trivial effort, but correctness *is* demanding!

Having a trusted specification as well as entire code-generation pipeline within a single dependently-typed framework is incredibly powerful. As we have demonstrated at the example of the neural network, we can introduce domain-specific optimisations and even swap the backend in case its performance is unsatisfying. For our example, the entire framework that includes array theory, DSL, optimisations and extraction is about 2000 lines of Agda code. We managed to introduce a new backend in about two days and match performance of the hand-written code.

A lot of pieces that we have developed in this paper can be reused in other numerical applications. We used dependent types to guarantee the absence of out-of-bound indexing, certain function being inverses as well as well-scopedness and well-typedness of our DSL. However, there are many more opportunities that we did not explore. For example, one can prove the correctness of optimisations, relating evaluation of optimised an non-optimised expressions. We can provide more guarantees when we run extraction, *e.g.* we can formalise some aspects of the backend language and relate them to our DSL. As for the DSL itself, we can try extending it with internal let constructions which should improve our C code generation as well as facilitate optimisations of the derivatives.

There are indeed plenty of opportunities, but the key point is this. Correctness and performance are competing requirements when it comes to application design. Therefore, such a cooperation between correctness-oriented and performance-oriented tools is likely to persist. With this work we demonstrate that such cooperation is feasible in practice.

References

[1] Jordy Aaldering, Sven-Bodo Scholz, and Bernard Van Gastel. 2024. Type Patterns: Pattern Matching on Shape-Carrying Array Types. In *Proceedings of the 35st Symposium on Implementation and Application of Functional Languages (IFL '23)*. Association for Computing Machinery, New York, NY, USA. (to appear).

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 265–283.

[3] Agda Development Team. 2024. *Agda 2.6.3 documentation.* https://agda.readthedocs.io/en/v2.6.3/ Accessed [2024/02/28].

[4] Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor McBride, and Peter Morris. 2015. Indexed containers. *Journal of Functional Programming* 25 (2015), e5. doi:10.1017/S095679681500009X

[5] Atılım Günes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2017. Automatic differentiation in machine learning: a survey. *J. Mach. Learn. Res.* 18, 1 (jan 2017), 5595–5637.

[6] Aloïs Brunel, Damiano Mazza, and Michele Pagani. 2019. Backpropagation in the simply typed lambda-calculus with linear negation. *Proc. ACM Program. Lang.* 4, POPL, Article 64 (dec 2019), 27 pages. doi:10.1145/3371132

[7] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. arXiv:1512.01274 [cs.DC]

[8] Geoffrey S. H. Cruttwell, Bruno Gavranović, Neil Ghani, Paul Wilson, and Fabio Zanasi. 2022. Categorical Foundations of Gradient-Based Learning. In *Programming Languages and Systems*, Ilya Sergey (Ed.). Springer International Publishing, Cham, 1–28.

[9] Conal Elliott. 2018. The simple essence of automatic differentiation. *Proc. ACM Program. Lang.* 2, ICFP, Article 70 (jul 2018), 29 pages. doi:10.1145/3236765

[10] Brendan Fong, David Spivak, and Rémy Tuyéras. 2021. Backprop as functor: a compositional perspective on supervised learning. In *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science* (Vancouver, Canada) *(LICS '19)*. IEEE Press, Article 11, 13 pages.

[11] Clemens Grelck. 2005. Shared memory multiprocessor support for functional array processing in SAC. *Journal of Functional Programming* 15, 3 (2005), 353–401. doi:10.1017/S0956796805005538

[12] James W. Hanson, Jane Shearin Caviness, and Camilla Joseph. 1962. Analytic differentiation by computer. *Commun. ACM* 5, 6 (jun 1962), 349–355. doi:10.1145/367766.368195

[13] Chantal Keller and Thorsten Altenkirch. 2010. Hereditary substitutions for simple types, formalized. In *Proceedings of the Third ACM SIGPLAN Workshop on Mathematically Structured Functional Programming* (Baltimore, Maryland, USA) *(MSFP '10)*. Association for Computing Machinery, New York, NY, USA, 3–10. doi:10.1145/1863597.1863601

[14] Faustyna Krawiec, Simon Peyton Jones, Neel Krishnaswami, Tom Ellis, Richard A. Eisenberg, and Andrew Fitzgibbon. 2022. Provably correct, asymptotically efficient, higher-order reverse-mode automatic differentiation. *Proc. ACM Program. Lang.* 6, POPL, Article 48 (jan 2022), 30 pages. doi:10.1145/3498710

[15] Min Lin. 2024. Automatic Functional Differentiation in JAX. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=gzT61ziSCu

[16] Conor McBride and Ross Paterson. 2008. Applicative programming with effects. *Journal of Functional Programming* 18, 1 (2008), 1–13. doi:10.1017/S0956796807006326

[17] J. F. Nolan. 1953. *Analytical Differentiation on a Digital Computer*. Master's thesis. Massachusetts Institute of Technology.

[18] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. *PyTorch: an imperative style, high-performance deep learning library*. Curran Associates Inc., Red Hook, NY, USA.

[19] F. Pfenning and C. Elliott. 1988. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, USA) *(PLDI '88)*. Association for Computing Machinery, New York, NY, USA, 199–208. doi:10.1145/53990.54010

[20] R. Schenck, O. RÃ¸nning, T. Henriksen, and C. E. Oancea. 2022. AD for an Array Language with Nested Parallelism. In *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC) (SC)*. IEEE Computer Society, Los Alamitos, CA, USA, 829–843. https://doi.ieeecomputersociety.org/

[21] Sven-Bodo Scholz. 1998. With-loop-folding in Sac — Condensing Consecutive Array Operations. In *Implementation of Functional Languages, 9th International Workshop (IFL'97), St. Andrews, UK, Selected Papers (Lecture Notes in Computer Science, Vol. 1467)*, Chris Clack, Tony Davie, and Kevin Hammond (Eds.). Springer, 72–92. doi:10.1007/BFb0055425

[22] Sven-Bodo Scholz. 2003. Single Assignment C: efficient support for high-level array operations in a functional setting. *Journal of Functional Programming* 13, 6 (2003), 1005–1059. doi:10.1017/S0956796802004458

[23] Sven-Bodo Scholz and Artjoms Šinkarovs. 2021. Tensor comprehensions in SaC. In *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages* (Singapore, Singapore) *(IFL '19)*. Association for Computing Machinery, New York, NY, USA, Article 15, 13 pages. doi:10.1145/3412932.3412947

[24] Justin Slepak, Olin Shivers, and Panagiotis Manolios. 2014. An Array-Oriented Language with Static Rank Polymorphism. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*. Springer-Verlag, Berlin, Heidelberg, 27–46. doi:10.1007/978-3-642-54833-8_3

[25] Birthe van den Berg, Tom Schrijvers, James McKinna, and Alexander Vandenbroucke. 2024. Forward- or reverse-mode automatic differentiation: What's the difference? *Science of Computer Programming* 231 (2024), 103010. doi:10.1016/j.scico.2023.103010

[26] Artjoms Šinkarovs, Hans-Nikolai Vießmann, and Sven-Bodo Scholz. 2021. Array languages make neural networks fast. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming* (Virtual, Canada) *(ARRAY 2021)*. Association for Computing Machinery, New York, NY, USA, 39–50. doi:10.1145/3460944.3464312