Structuring Arrays with Algebraic Shapes

Jakub Bachurski
kbachurski@gmail.com
University of Cambridge
United Kingdom

Alan Mycroft
alan.mycroft@cl.cam.ac.uk
University of Cambridge
United Kingdom

Dominic Orchard
d.a.orchard@kent.ac.uk
University of Cambridge
University of Kent
United Kingdom

Abstract

Static type systems help prevent errors, improve abstractions, and enable optimisations. There is a whole spectrum of type systems for general-purpose languages, covering a wide range of safety guarantees and expressivity. Despite this, type systems for array programming languages are usually at one of two extremes. In the majority of cases they are nearly untyped, only distinguishing array types by element type or number of dimensions. Otherwise, they tend to reach for powerful dependent types. However, it is difficult to extend existing solutions with a dependent type system, and some problems become undecidable when we do so. Practical array programming – in data science, machine learning and the like – sticks to the bliss of dynamic typing.

We propose a novel calculus for array programming: Star. Array indices and shapes in Star make use of *structural* record and variant types equipped with subtyping. We prevent indexing errors not by resolving arithmetic problems, but by enabling richer types for arrays, allowing programmers to capture their structure explicitly. While we present Star with only subtype polymorphism, we sketch how *algebraic subtyping* promises efficient ML-style polymorphic type inference.

CCS Concepts: • **Software and its engineering → Functional languages**; *Parallel programming languages*; *Data types and structures.*

Keywords: pointful array programming, structural subtyping, type system

ACM Reference Format:
Jakub Bachurski, Alan Mycroft, and Dominic Orchard. 2025. Structuring Arrays with Algebraic Shapes. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming (ARRAY '25), June 17, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3736112.3736141

ARRAY '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-1927-1/25/06
https://doi.org/10.1145/3736112.3736141

1 Introduction

Array programming currently has no satisfactory typing discipline. The untyped[1] *array programming model*, introduced in APL by Iverson [12], currently dominates in languages like Python or MatLab. It is universal in machine learning and data science through libraries like NumPy [9] (and its extensions, e.g. Xarray [11]) and PyTorch [15].

General-purpose programming languages often keep array typing simple, reducing arrays to containers indexed by integers. This fails to capture even simple structures of data, like multiple regular dimensions. At worst, the programmer represents multidimensional arrays as nested arrays of arrays – losing out on performance and useful type information – or applies a handwritten indexing scheme. At best, arrays end up generalised to *tensors* indexed by tuples of integers. Such multidimensional arrays have been the standard since the times of Fortran.

Untyped programming invites primitives with excessive flexibility, so the situation is difficult to amend. In such cases, past work resorts to dependent types – one such project is Remora [25], but there has also been more recent work in Futhark [3, 10]. Such powerful systems allow types to involve arithmetic on array sizes, modelling the complex behaviours of array operations. This approach enables safe indexing (boundary checking) to be statically enforced (with potential proof obligations). However, this also makes types difficult to check – let alone infer. In the general case, the programmer is given the burden of proof [3].

We hit an impasse – we either have just simple array types, or have to reach for dependent types. Our main contribution is a novel calculus, **Star**, with a type system that provides useful and expressive types, while also admitting type inference. Hence, we reveal a new area in the design space of array programming languages and their type systems.

Furthermore, thanks to recent developments in *algebraic subtyping*, we conjecture Star admits ML-style type inference with parametric polymorphism and principal types. In practice, this would reduce the annotation burden on the programmer, bringing Star's style closer to dynamic languages.

The key idea we put forward is to use *structural record and variant types* for array indices and shapes – we were inspired by *algebraic data types*, hence our titular *algebraic shapes*.

[1] We use untyped in a slightly loose sense: array programming languages usually feature a single array type (e.g. NumPy's `ndarray`) with little nuance.

Then, we use structural subtyping to model ubiquitous array programming patterns. Our subtyping order underlies a distributive lattice algebra – in line with *algebraic subtyping* – further reinforcing the algebraic nature of our shape types.

Our contributions are as follows:

- A novel design for an array calculus – Star – which features subtyping and structural types for array shapes. We motivate it in Section 2 and elaborate in Section 3.
- Formalisation of Star's operational semantics (Section 3.5), type system (Section 4; *without parametric polymorphism*), and type safety (Section 4.3).
- Worked examples where Star shines, showing its potential (Section 5) and identifying future work (Section 6).

2 Motivation

In this work, we seek to alleviate the status quo – of arrays as simply containers indexed by (tuples of) integers – which we described in the introduction. We believe this leads to the unfortunate present dogma:

> *Your array can be any type, as long as it is n-dimensional.*

This has become a standard assumption for array programmers, even though we have more general frameworks for indexable structures – like dependently typed containers [1] or Naperian functors [8]. We argue that the current lack of nuance to array types can be addressed by constructing an analogue of algebraic data types for arrays.

We now motivate the design of Star by showing how common array patterns are modelled when we use algebraic data types – records and variants – for array indices.

Terminology. Arrays have a *dimensionality* (i.e. number of dimensions), also called a *rank*. The *shape* of an array defines the extent (*size*) of each dimension. Dimensions are sometimes referred to as *axes*. A *batch* refers to a sequence of something, and we speak of a batch dimension (batch axis).

2.1 Record Indices Label Axes

Consider an array representing a batch of images. Usually, we would model such a structure as a 4-dimensional array, with a batch dimension, two coordinate dimensions, and a channel dimension. For instance, a size 100 batch of 32×32 RGB images might have shape $(100, 32, 32, 3)$.

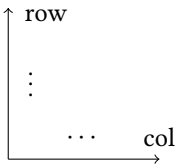
We could instead consider a different interpretation for this array. Notice that a value of the record type:

$$\{\text{batch} : \text{int}, \text{row} : \text{int}, \text{col} : \text{int}, \text{channel} : \text{int}\}$$

could clearly be used as an index into our batch of images. In fact, this perspective has its benefits. For example, a simple 4-dimensional structure does not differentiate dimensions beyond their position in the sequence – the programmer would usually need to keep track of the order of dimensions with e.g. unenforceable annotations in comments.

Thus, we propose that for any array, shape types $\sigma, \ldots$ and axis labels $\ell, \ldots$ we should have a *product shape* $\{\!|\ell : \sigma, \ldots|\!\}$ indexed by records of type $\{\ell : \sigma, \ldots\}$. Records provide a **product** on shapes.

We might illustrate $\{\!|\text{row} : \text{int}, \text{col} : \text{int}|\!\}$ as:



One might find this idea reminiscent of named tensors [4] or Xarray [11] (among others), and this similarity is not accidental. We believe that the practical usefulness of prior work leads us to generalising this technique.

2.2 Variant Indices Concatenate

Having seen that indexing arrays with records models (labelled) multidimensional arrays, we turn our attention to the dual of records – variants.

Consider an array indexed by values of variant type

$$[\text{L} : \text{int}, \text{R} : \text{int}]$$

This corresponds to a composition of array shapes where we can access any *one* of the components – **concatenation**.

Why is this perspective useful? Normally, concatenating arrays can require the programmer to manipulate ranges corresponding to each of the components. For instance, padding an array of size n with a elements at the front and b at the back creates a vector composed of three ranges: $[0, a)$ (front), $[a, n + a)$ (centre), $[n + a, n + a + b)$ (back). Composing and decomposing a concatenated vector – e.g. setting and getting its components – requires reconstructing these ranges. Instead, we shall use an index of variant type:

$$[\text{Front} : \text{int}, \text{Centre} : \text{int}, \text{Back} : \text{int}]$$

which allows us to rely on tagging and pattern matching for composing and decomposing concatenations.

We propose that for shapes $\sigma, \ldots$ and tags $T, \ldots$ we have a shape type $[\![T : \sigma, \ldots]\!]$ indexed by variants of type $[T : \sigma, \ldots]$.

Looking ahead, Table 1 gives examples of shape values and index values in Star along with their types.

3 Calculus

We now introduce the key contribution of this work – the design of **Star** (**st**ructured **ar**rays), a functional, pointful array programming calculus with a novel type system. We begin by introducing expressions (Section 3.1) and our novel structured array shapes (Section 3.2) in Star. We then sketch an operational semantics (Section 3.5). We give a complete description of our type system, which features structural types and subtyping, in Section 4.

$$
\begin{array}{rl}
e ::= & n \mid f \quad \text{(integer and float literal)} \\
\mid & \pi(\overline{e}) \quad \text{(scalar operations)} \\
\mid & x \mid \text{let } x = e \text{ in } e \quad \text{(variable, let-binding)} \\
\mid & \lambda x.\, e \mid e\ e \quad \text{(function, application)} \\
\mid & \{\overline{\ell = e}\} \quad \text{(record construction)} \\
\mid & e.\ell \quad \text{(record projection)} \\
\mid & T\ e \quad \text{(variant construction)} \\
\mid & \text{match } e \text{ with } \overline{\mid T\ x \implies e} \quad \text{(pattern matching)} \\
\mid & \Phi\, x[e].\, e \quad \text{(array comprehension)} \\
\mid & \textsc{Arr}(s, J) \quad \text{(unevaluated array)} \\
\mid & e[e] \quad \text{(array indexing)} \\
\mid & |e| \quad \text{(array shape)} \\
\mid & e^{\mathrm{S}} \quad \text{(shape expression)}
\end{array}
$$

$$
\begin{array}{rl}
e^{\mathrm{S}} ::= & \#e \quad \text{(sized shape)} \\
\mid & \{\!|\overline{\ell = e}|\!\} \quad \text{(product shape)} \\
\mid & e_{\circ}\ell \quad \text{(product dimension projection)} \\
\mid & [\![\overline{T = e}]\!] \quad \text{(concatenation shape)} \\
\mid & e_{\diamond}T \quad \text{(concatenation component projection)} \\
\mid & e \sqcap e \quad \text{(shape broadcasting)}
\end{array}
$$

Figure 1. Expressions e in Star, incl. shape expressions e^{S}.

Notation. Throughout this section, we write n for integers, f for floats, ℓ for (record) labels, and T for (variant) tags. We also write $\overline{t}$ for a repetition of terms of metatype t.

3.1 Language

The basis of Star is a λ-calculus with structurally typed records and variants. We define the grammars of expressions e and values v in Star in Figures 1 and 2. We start with some informal explanations:

- We write π for scalar operations, which consume and produce integers and floats. Examples include $+ : \text{int} \times \text{int} \to \text{int}$, $\sin : \text{float} \to \text{float}$, or $\lfloor\cdot\rfloor : \text{float} \to \text{int}$.
- Functions and let-bindings are standard. For simplicity, we do not include recursive let-bindings.
- We include structural records and variants, with record projection $e.\ell$ and pattern matching (match) as elimination forms.
- Star's array operations are similar to the likes of $\tilde{F}$ [24]. Our primitives are array comprehensions $\Phi\, x[e].\, e'$ (constructing an array of shape e and elements at index x given by e') and indexing $e[e']$ (accessing index e' of array e). The shape of an array e is $|e|$.

$$
\begin{array}{rl}
v ::= & n \mid f \quad \text{(integer, float)} \\
\mid & \lambda x.\, e \quad \text{(function)} \\
\mid & \{\overline{\ell = v}\} \quad \text{(record)} \\
\mid & T\ v \quad \text{(variant)} \\
\mid & \textsc{Arr}(s, I) \quad \text{(array)} \\
\mid & s \quad \text{(shape value)}
\end{array}
$$

$$
\begin{array}{rl}
s ::= & \#n \quad \text{(sized shape)} \\
\mid & \{\!|\overline{\ell = s}|\!\} \quad \text{(product shape)} \\
\mid & [\![\overline{T = s}]\!] \quad \text{(concatenation shape)}
\end{array}
$$

Figure 2. Values v in Star, including shape values s. All values are expressions.

We give the operational semantics for Star in Figure 6, but first explore details of shapes, indices, and finally arrays.

3.2 Array Shapes

Our values (and types, see Section 4) for array shapes are novel. In Star, array comprehensions require providing a shape. We informally summarise them now and elaborate later. We have three constructors for shape values:

- A sized shape $\#e$ is given to a usual flat array. #10 is the shape of an array indexed by integers in the range $[0, 10)$.
- Product shapes are used for defining arrays with multiple *named* dimensions. For instance:

  $$\{\!|\text{row} = \#5, \text{col} = \#4|\!\}$$

  is the shape of a 5×4 matrix (with dimensions named as rows and columns). To index into an array of a product shape, we provide a record – e.g. $\{\text{row} = 2, \text{col} = 3\}$. Products capture *rectangular* (regular) multidimensional arrays. Note that an array of shape $\{\!|\}\!\}$ has a single element.
- Concatenation shapes are given by a sequence of *named* component shapes. For example,

  $$[\![\text{Left} = \{\!|\}\!\}, \text{Centre} = \#8, \text{Right} = \{\!|\}\!\}]\!]$$

  describes an 8-element array with an extra element at the start and end (i.e. a halo/padding/boundary of size 1). We index into arrays of concatenation shapes with variants, e.g. Left $\{\}$ or Centre 2. An array of shape $[\![]\!]$ is empty.

Projections on shapes extract dimensions/components of products/concatenations. For example:

$$\{\!|\text{row} = \#5, \text{col} = \#4|\!\}_{\circ}\text{row} \rightsquigarrow \#5$$

$$[\![\text{Left} = \#1, \text{Right} = \#0]\!]_{\diamond}\text{Left} \rightsquigarrow \#1$$

Lastly, Star's *shape broadcasting* operation $e \sqcap e'$ is used to *align* compatible shapes e and e' by finding their common *sub-shape* (cf. broadcasting in NumPy in Appendix A). We define this operation in Figure 3 and elaborate in Section 4.2.

$$\#n \bowtie \#m = \#n \qquad \text{if } n = m$$

$$\left\{\overline{\ell' = s'_\ell}\right\} \bowtie \left\{\overline{\ell'' = s''_\ell}\right\} = \left\{\overline{\ell = s'_\ell \bowtie s''_\ell}\right\} \qquad \text{for } \ell = \ell' \cup \ell''$$

$$\left[\!\left[\overline{T = s'_T}\right]\!\right] \bowtie \left[\!\left[\overline{T = s''_T}\right]\!\right] = \left[\!\left[\overline{T = s'_T \bowtie s''_T}\right]\!\right]$$

Figure 3. Semantics $\bowtie$ of shape broadcasting $\sqcap$. We take the union of sets of dimensions ℓ (defaulting $s'_\ell = s''_\ell$ is either is absent), but equate components T. If $s' \bowtie s''$ is undefined, we say s' and s'' are *incompatible*, and $s' \sqcap s''$ is a *stuck state*.

$$\frac{}{n \lhd \#m} \quad \text{if } 0 \le n < m$$

$$\frac{\forall \ell'.\, v_{\ell'} \lhd s_{\ell'}}{\left\{\overline{\ell = v_\ell}\right\} \lhd \left\{\overline{\ell' = s_{\ell'}}\right\}} \qquad \frac{v \lhd s_T}{T\, v \lhd \left[\!\left[\overline{T = s_T}\right]\!\right]}$$

Figure 4. Definition of the in-bounds relation $v \lhd s$, stating that an index v can be used to index into a shape s.

$$n \oslash \#m = n$$

$$\left\{\overline{\ell = v_\ell}, \overline{\ell' = v'_\ell}\right\} \oslash \left\{\overline{\ell = s_\ell}\right\} = \left\{\overline{\ell = v_\ell \oslash s_\ell}\right\}$$

$$T\, v \oslash \left[\!\left[\overline{T' = v_{T'}}\right]\!\right] = T\,(v \oslash v_T)$$

Figure 5. Definition of the index-shape cast $v \oslash s$, which removes dimensions from v which do not appear in s.

3.3 Bounds of Shapes

We now make precise what indices v can be used to index into a given shape s. We use the *in-bounds* relation $v \lhd s$ to formalise this. We define $\lhd$ in Figure 4.

For integers, in-bounds behaves as expected. However, structural indices have more interesting behaviour due to subtyping: for products, an index with more dimensions than the shape can be used for indexing; for concatenations any component works. To reason about these extra dimensions, we need a way to manipulate them – we define a *cast* operator $v \oslash s$ in Figure 5, which removes these extra dimensions from v, coercing an index to fit the structure of a given shape. We then define the *within-bounds* relation $v \trianglelefteq s$ as:

$$v \trianglelefteq s \iff v \lhd s \text{ and } v = v \oslash s$$

In the operational semantics, $\trianglelefteq$ determines the indices v for which we should compute and store the elements when building an array from a comprehension. $v \oslash s$ is used instead of v when accessing those stored elements.

Example. Both $\{x = 3\} \lhd \{\!|x = \#4|\!\}$ and $\{x{=}3\} \trianglelefteq \{\!|x{=}\#4|\!\}$ hold. While we have $\{x = 3\} \lhd \{\!|\,|\!\}$, $\{x = 3\} \trianglelefteq \{\!|\,|\!\}$ is false (since $\{x = 3\} \oslash \{\!|\,|\!\} = \{\}$) and only $\{\} \trianglelefteq \{\!|\,|\!\}$ holds.

$$\boxed{e \rightsquigarrow e}$$

$$(\lambda x.\, e)\, v \rightsquigarrow [v/x]e \qquad \textsc{StepApply}$$

$$\text{let } x = v \text{ in } e \rightsquigarrow [v/x]e \qquad \textsc{StepLet}$$

$$\left\{\overline{\ell = v_\ell}\right\}.\ell \rightsquigarrow v_\ell \qquad \textsc{StepRecordProj}$$

$$\text{match } T\, v \text{ with } T\, x \Rightarrow e \mid \cdots \rightsquigarrow [v/x]e \qquad \textsc{StepMatch}$$

$$\Phi\, x[s].\, e \rightsquigarrow \textsc{Arr}(s, \{v \mapsto [v/x]e \mid v \trianglelefteq s\}) \qquad \textsc{StepArray}$$

$$\textsc{Arr}(s, I)[v] \rightsquigarrow I(v \oslash s) \quad (\text{if } v \lhd s) \qquad \textsc{StepIndex}$$

$$|\textsc{Arr}(s, I)| \rightsquigarrow s \qquad \textsc{StepShape}$$

$$\left\{\!\left|\overline{\ell = v_\ell}\right|\!\right\}_\circ \ell \rightsquigarrow v_\ell \qquad \textsc{StepProductProj}$$

$$\left[\!\left[\overline{T = v_T}\right]\!\right]_\diamond T \rightsquigarrow v_T \qquad \textsc{StepConcatProj}$$

$$s' \sqcap s'' \rightsquigarrow s' \bowtie s'' \text{ (see Fig. 3)} \qquad \textsc{StepBroadcast}$$

Figure 6. Small-step operational semantics for Star. We write $e \rightsquigarrow e'$ when e reduces to e'. We formalise congruence (a.k.a. context) rules in Appendix C – we never evaluate under λ or Φ; but we evaluate expressions in $\textsc{Arr}(s, J)$. Indexing and broadcasting may result in a *stuck state*.

3.4 Representation of Array Values

In Star, array comprehensions $\Phi\, x[e].\, e'$ eventually evaluate to array values $\textsc{Arr}(s, I)$. These consist of a shape value s along with a partial function (taken as a set) I from index values to element values. The domain of I consists of exactly the indices v which are within bounds of s: $v \trianglelefteq s$.

However, evaluating $\Phi\, x[e].\, e'$ generally requires multiple evaluations of e'; our small-step operational semantics achieves this using *unevaluated array expressions* $\textsc{Arr}(s, J)$. These generalise array values above, allowing the codomain of J to hold expressions instead of values. Operationally, array comprehensions Φ yield array expressions $\textsc{Arr}(s, J)$, which are evaluated until they become values $\textsc{Arr}(s, I)$. Note that $\textsc{Arr}(s, J)$ is not part of the user-facing expression syntax – it exists solely to enable a *small-step* operational semantics.

3.5 Semantics

We present a call-by-value operational semantics for Star in Figure 6. Run-time errors, e.g. out-of-bounds indexing, manifest as *stuck states*, which do not step further. Successful executions eventually reduce to a value v (Figure 2).

4 Typing

We now introduce our type system for Star. We first describe notable aspects of the syntax of Star types, given in Figure 7. A complete description of the subtyping relation and typing judgements is provided. We then focus on typing array shapes and indices. Finally, we briefly consider how we could provide ML-style type inference, admitting polymorphism.

$$
\begin{array}{lll}
\tau ::= \top \mid \bot & & \text{(top, bottom)} \\
\quad \mid \text{int} \mid \text{float} & & \text{(integer, float)} \\
\quad \mid \tau \to \tau & & \text{(function)} \\
\quad \mid \{\overline{\ell : \tau}\} & & \text{(record)} \\
\quad \mid [\overline{T : \tau}] & & \text{(variant)} \\
\quad \mid [\sigma_1..\sigma_2]\tau & \text{for } \sigma_1 \leqslant \sigma_2 & \text{(array)} \\
\quad \mid \sigma & & \text{(shape)} \\
\\
\sigma ::= \# & & \text{(sized)} \\
\quad \mid \{\!|\overline{\ell : \sigma}|\!\} & & \text{(product)} \\
\quad \mid [\![\overline{T : \sigma}]\!] & & \text{(concatenation)} \\
\\
\eta ::= \text{int} \mid \{\overline{\ell : \eta}\} \mid [\overline{T : \eta}] & & \text{(indices in } \tau\text{)}
\end{array}
$$

Figure 7. Types τ in Star, showing their subtypes: shape types σ and index types η. Note that σ and η are in bijection (see Figure 10). Array types $[\sigma_1..\sigma_2]\tau$ are defined only when $\sigma_1 \leqslant \sigma_2$ (with $\leqslant$ as in Figure 8) – we elaborate in Section 4.

Table 1. Shapes and corresponding indices. Given a shape of type σ on the left, we have an index of type $\iota(\sigma)$ on the right.

Shape		Index	
Type	Sample Value	Type	Sample Value
#	#5	int	4
$\{\!\|\overline{\ell : \sigma}\|\!\}$	$\{\!\|\text{r} = \#5, \text{c} = \#4\|\!\}$	$\{\overline{\ell : \iota(\sigma)}\}$	$\{\text{r} = 4, \text{c} = 3\}$
$[\![\overline{T : \sigma}]\!]$	$\{\!\|L = \#10, R = \#20\|\!\}$	$[\overline{T : \iota(\sigma)}]$	$R\,19$

Splitting σ. We give special attention to our array types, which have the syntax $[\sigma_1..\sigma_2]\tau$ for $\sigma_1 \leqslant \sigma_2$, where $\leqslant$ is our subtyping order. We do this to provide satisfactory subtyping for array types, following a technique used by both Dolan [6] and Pottier [19] for reference types. The crux is to split the shape σ in $[\sigma]\tau$ into a contravariant part σ_1 and covariant part σ_2, so that indexing uses σ_1, while accessing the shape returns a σ_2. With a simpler array type like $[\sigma]\tau$ (which we write to abbreviate $[\sigma..\sigma]\tau$) array types would have to be invariant (see Appendix B). By requiring $\sigma_1 \leqslant \sigma_2$ there is a σ such that $\sigma_1 \leqslant \sigma \leqslant \sigma_2$: we have an *actual* shape σ into which we index with $\iota(\sigma_1) \leqslant \iota(\sigma)$ (where $\iota(\sigma)$ is the index type for σ), and accessing the shape we only use it as $\sigma \leqslant \sigma_2$.

Subtyping. Our type system for Star relies on structural subtyping to handle certain useful patterns, and we present it as such. We define the subtyping relation $\leqslant$ in Figure 8.

It is interesting to note the correspondence of subtyping for shapes and their indices. In particular, variants have their width subtyping rule flipped, while concatenations do not. Since we have dimension $e_\circ\ell$ and component $e_\diamond T$ projections, both product and concatenation shapes subtype like records.

In accordance with algebraic subtyping, we use a subtyping relation which forms a distributive lattice – we have meets $\wedge$ (intersections) and joins $\vee$ (unions) on types τ. These agree with the subtyping relation, i.e.

$$\tau \leqslant \tau' \iff \tau = \tau \wedge \tau' \iff \tau' = \tau \vee \tau'$$

Typing Judgements. Finally, we give the complete typing rules for Star in Figure 9. We elaborate on these and the remaining aspects of the calculus in the following subsections.

4.1 Shapes and Indices

In the typing rules, we need to relate shapes and the index types. Let us consider the index type $\eta = \iota(\sigma)$ used with an array of shape type σ – we give examples in Table 1.

- Sized shapes # are indexed with the integer type int. Note that these have to be bounds checked at runtime (which may fail).
- Product shapes $\{\!|\ell : \sigma, \ell' : \sigma', \ldots|\!\}$ are indexed with a **record** of indices for shapes of individual components, i.e. $\{\ell : \iota(\sigma), \ell' : \iota(\sigma'), \ldots\}$
- The case of concatenation shapes $[\![T : \sigma, T' : \sigma', \ldots]\!]$ is particularly interesting. While concatenations are constructed by a *sequence* of component shapes like $[\![T : \sigma, T' : \sigma', \ldots]\!]$, they are indexed by variant types $[T : \iota(\sigma), T' : \iota(\sigma'), \ldots]$. We concatenate a sequence of shapes, but index with one tagged value.

We model this unique relationship between indices and types with ι. Crucially, ι (given in Figure 10) is an isomorphism between shape types σ and index types η.

It is clear that in Star it no longer suffices to pretend that array shapes and indices can be of the same type. This would be a standard assumption[2] with only n-dimensional arrays – where indices and shapes are tuples of integers.

Introducing ι is not as surprising if we consider Star arrays as *containers* [1]. A container has an element type α, a Shape : Type and a function Position : Shape $\to$ Type, so that we have a $\Sigma(s : \text{Shape}).\,\text{Position}\,s \to \alpha$. In Star, arrays have Shapes given by σ and Positions given by ι. There is a similar connection to Naperian functors, which Gibbons [8] has already found useful for expressing array programming.

The type system ensures that we always index into arrays with expected dimensions (in products) and components (in concatenations). However, unlike most dependent type systems for arrays, we still need to perform runtime bounds checks for sized shapes at runtime.

More formally, if $v \lhd s$ – so $v : \eta$ is in-bounds of $s : \sigma$ – we must have $\eta \lhd \iota(\sigma)$. When showing type safety, we show this is an equivalence *up to integer indexing*.

[2] In fact, Haskell's `Data.Array` – polymorphic in the index type – presumes shapes are *ranges* of indices, which cannot express concatenations.

$$\boxed{\tau \leqslant \tau}$$

$$\textsc{SubRefl}\ \frac{}{\tau \leqslant \tau} \qquad \textsc{SubTop}\ \frac{}{\tau \leqslant \top} \qquad \textsc{SubBot}\ \frac{}{\bot \leqslant \tau} \qquad \textsc{SubTrans}\ \frac{\tau \leqslant \tau' \quad \tau' \leqslant \tau''}{\tau \leqslant \tau''}$$

$$\textsc{SubFun}\ \frac{\tau_1' \leqslant \tau_1 \quad \tau_2 \leqslant \tau_2'}{(\tau_1 \to \tau_2) \leqslant (\tau_1' \to \tau_2')} \qquad \textsc{SubArray}\ \frac{\sigma_1' \leqslant \sigma_1 \quad \sigma_2 \leqslant \sigma_2' \quad \tau \leqslant \tau'}{[\sigma_1..\sigma_2]\tau \leqslant \left[\sigma_1'..\sigma_2'\right]\tau'}$$

$$\textsc{SubRecWidth}\ \frac{}{\{\ell : \tau, \rho\} \leqslant \{\rho\}} \qquad \textsc{SubRecDepth}\ \frac{\tau \leqslant \tau'}{\{\ell : \tau, \rho\} \leqslant \{\ell : \tau', \rho\}} \qquad \textsc{SubProdWidth}\ \frac{}{\{\!|\ell : \tau, \rho|\!\} \leqslant \{\!|\rho|\!\}} \qquad \textsc{SubProdDepth}\ \frac{\tau \leqslant \tau'}{\{\!|\ell : \tau, \rho|\!\} \leqslant \{\!|\ell : \tau', \rho|\!\}}$$

$$\textsc{SubVarWidth}\ \frac{}{[\rho] \leqslant [T : \tau, \rho]} \qquad \textsc{SubVarDepth}\ \frac{\tau \leqslant \tau'}{[T : \tau, \rho] \leqslant [T : \tau', \rho]} \qquad \textsc{SubConcatWidth}\ \frac{}{[\![T : \tau, \rho]\!] \leqslant [\![\rho]\!]} \qquad \textsc{SubConcatDepth}\ \frac{\tau \leqslant \tau'}{[\![T : \tau, \rho]\!] \leqslant [\![T : \tau', \rho]\!]}$$

$$\textsc{SubShapeSize}\ \frac{}{\# \leqslant \sigma} \qquad \textsc{SubShapeProd}\ \frac{\tau \leqslant \sigma \quad \{\!|\rho|\!\} \leqslant \sigma}{\{\!|\ell : \tau, \rho|\!\} \leqslant \sigma} \qquad \textsc{SubShapeConcat}\ \frac{\tau \leqslant \sigma \quad [\![\rho]\!] \leqslant \sigma}{[\![T : \tau, \rho]\!] \leqslant \sigma}$$

$$\textsc{SubIndexInt}\ \frac{}{\text{int} \leqslant \eta} \qquad \textsc{SubIndexRecord}\ \frac{\tau \leqslant \eta \quad \{\!|\rho|\!\} \leqslant \eta}{\{\!|\ell : \tau, \rho|\!\} \leqslant \eta} \qquad \textsc{SubIndexVariant}\ \frac{\tau \leqslant \eta \quad [\![\rho]\!] \leqslant \eta}{[\![T : \tau, \rho]\!] \leqslant \eta}$$

Figure 8. Subtyping rules for Star types τ. SubArray is particularly noteworthy. We write ρ for a list of entries in a type (like fields of a record). We consider these to be unordered – $\rho = (\ell : \tau, \ell' : \tau')$ and $\rho' = (\ell' : \tau', \ell : \tau)$ are equal. We include rules for specifying σ to be a supertype of shape types and η a supertype of index types.

$$\boxed{\Gamma \vdash e : \tau}$$

$$\Gamma ::= \cdot \mid \Gamma, x : \tau$$

$$\textsc{Sub}\ \frac{\Gamma \vdash e : \tau \quad \tau \leqslant \tau'}{\Gamma \vdash e : \tau'} \qquad \textsc{Var}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \textsc{Int}\ \frac{}{\Gamma \vdash n : \text{int}} \qquad \textsc{Float}\ \frac{}{\Gamma \vdash f : \text{float}}$$

$$\textsc{Let}\ \frac{\Gamma \vdash e : \tau \quad \Gamma, x : \tau \vdash e' : \tau'}{\Gamma \vdash \text{let}\, x = e \,\text{in}\, e' : \tau'} \qquad \textsc{Lambda}\ \frac{\Gamma, x : \tau \vdash e' : \tau'}{\Gamma \vdash \lambda x.\, e : \tau \to \tau'} \qquad \textsc{Apply}\ \frac{\Gamma \vdash e : \tau' \to \tau \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash e\, e' : \tau}$$

$$\textsc{Record}\ \frac{\forall \ell.\, \Gamma \vdash e_\ell : \tau_\ell}{\Gamma \vdash \left\{\overline{\ell = e_\ell}\right\} : \left\{\overline{\ell : \tau_\ell}\right\}} \qquad \textsc{RecordProj}\ \frac{\Gamma \vdash e : \{\ell : \tau\}}{\Gamma \vdash e.\ell : \tau} \qquad \textsc{Tag}\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash T\, e : [T : \tau]}$$

$$\textsc{Match}\ \frac{\Gamma \vdash e : \left[\overline{T : \tau_T}\right] \quad \forall T.\, \Gamma, x_T : \tau_T \vdash e_T : \tau}{\Gamma \vdash \text{match}\, e\, \text{with}\, \overline{T\, x_T \Rightarrow e_T} : \tau}$$

$$\textsc{Array}\ \frac{\Gamma \vdash e : \sigma \quad \Gamma, x : \iota(\sigma) \vdash e' : \tau}{\Gamma \vdash \Phi\, x[e].\, e' : [\sigma]\tau} \qquad \textsc{ArrayLit}\ \frac{\Gamma \vdash s : \sigma \quad \forall (v \mapsto e) \in J.\, \Gamma \vdash v : \iota(\sigma) \text{ and } \Gamma \vdash e : \tau}{\Gamma \vdash \textsc{Arr}(s, J) : [\sigma]\tau}$$

$$\textsc{Index}\ \frac{\Gamma \vdash e : [\sigma_1..\sigma_2]\tau \quad \Gamma \vdash e' : \iota(\sigma_1)}{\Gamma \vdash e[e'] : \tau} \qquad \textsc{Shape}\ \frac{\Gamma \vdash e : [\sigma_1..\sigma_2]\tau}{\Gamma \vdash |e| : \sigma_2} \qquad \textsc{Broadcast}\ \frac{\Gamma \vdash e : \sigma \quad \Gamma \vdash e' : \sigma'}{\Gamma \vdash e \sqcap e' : \sigma \wedge \sigma'}$$

$$\textsc{Sized}\ \frac{\Gamma \vdash e : \text{int}}{\Gamma \vdash \#e : \#} \qquad \textsc{Product}\ \frac{\forall \ell.\, \Gamma \vdash e_\ell : \sigma_\ell}{\Gamma \vdash \left\{\!\left|\overline{\ell = e_\ell}\right|\!\right\} : \left\{\!\left|\overline{\ell : \sigma_\ell}\right|\!\right\}} \qquad \textsc{ProductProj}\ \frac{\Gamma \vdash e : \{\!|\ell : \sigma|\!\}}{\Gamma \vdash e_\circ \ell : \sigma}$$

$$\textsc{Concat}\ \frac{\forall T.\, \Gamma \vdash e_T : \sigma_T}{\Gamma \vdash \left[\!\left[\overline{T = e_T}\right]\!\right] : \left[\!\left[\overline{T : \sigma_T}\right]\!\right]} \qquad \textsc{ConcatProj}\ \frac{\Gamma \vdash e : [\![T : \sigma]\!]}{\Gamma \vdash e_\diamond T : \sigma}$$

Figure 9. Rules for the typing judgement $\Gamma \vdash e : \tau$ for Star expressions e. We use σ for metavariables of shape types. Note ι in Array and Index maps shape types σ into corresponding index types η. Quantifiers $\forall t. \cdots$ stand for an expanded list of assumptions instantiated for all appropriate t.

$$\iota(\#) = \text{int}$$
$$\iota\left(\{\!|\overline{\ell : \sigma_\ell}|\!\}\right) = \left\{\overline{\ell : \iota(\sigma_\ell)}\right\}$$
$$\iota\left(\left[\!\left[\overline{T : \sigma_T}\right]\!\right]\right) = \left[\overline{T : \iota(\sigma_T)}\right]$$

Figure 10. Definition of the metafunction ι, which given a shape type σ gives its index type η.

Historically, a focus on the safety of integer indices in turn caused a focus on type-level arithmetic in type systems for array programming. We take another approach: enabling a *shape-polymorphic* programming style relying on structural shapes, ensuring the safety of array operations which do not rely on integer indices. We have not yet formalised a type system with parametric polymorphism for Star, so we showcase the idea in the following text on simple examples – taking type schemes as types up to a substitution.

Example. Take

$$e = \Phi\, p[\{\!|\text{row} = \#2, \text{col} = \#3|\!\}].\, p.\text{row} + p.\text{col}$$

Then $\cdot \vdash e : [\{\text{row} : \#, \text{col} : \#\}]\text{int}$ and:

$$e \rightsquigarrow^* \begin{array}{c} \text{col} \\ \begin{bmatrix} 0 & 1 & 2 \\ 1 & 2 & 3 \end{bmatrix} \end{array} \text{row}$$

where index $\{\text{row} = i, \text{col} = j\}$ accesses the cell in the i-th row and j-th column. Thus, $e[\{\text{row} = 1, \text{col} = 1\}] \rightsquigarrow^* 2$.

4.2 Shape Broadcasting

Let us finally consider shape broadcasting – which we use to model NumPy-style broadcasting (explained in Appendix A) – through the lens of our type system.

Shape broadcasting $s' \bowtie s''$ (for shape values $s' : \sigma'$, $s'' : \sigma''$) is a partial operator that finds the shape $s : \sigma$ for which $\sigma = \sigma' \wedge \sigma''$ (where $\wedge$ is the meet – greatest lower bound – on types), so that for any index i:

$$i \lhd s \iff i \lhd s' \text{ and } i \lhd s''$$

Hence, $\sqcap$ is the meet in a lattice spanned by the predicate $\lhd\, s$, and its result type takes the meet of shape types: clarifying how broadcasting finds 'sub-shapes', as said in Section 3.2.

Example. Take a column vector $u : [\{\!|\text{row} : \#|\!\}]\text{float}$ and row vector $w : [\{\!|\text{col} : \#|\!\}]\text{float}$. The expression:

$$e = \Phi\, i[|u| \sqcap |w|].\, u[i] \times w[i]$$

has type $[\{\!|\text{row} : \#, \text{col} : \#|\!\}]\text{float}$, and computes the outer product $u \otimes w$. We can depict this as:

$$\begin{array}{c} \\ \text{row} \end{array} \begin{bmatrix} -1 \\ 0 \\ +1 \end{bmatrix} \quad \begin{array}{c} \text{col} \\ \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \end{array} = \begin{array}{c} \text{col} \\ \begin{bmatrix} -1 & -2 & -3 \\ 0 & 0 & 0 \\ +1 & +2 & +3 \end{bmatrix} \end{array} \text{row}$$

Notice that the most-general type of the function

$$\lambda u.\, \lambda w.\, \Phi\, i[|u| \sqcap |w|].\, u[i] \times w[i]$$

is polymorphic in the specific shapes σ and σ':

$$[\sigma]\text{float} \to [\sigma']\text{float} \to [\sigma \wedge \sigma']\text{float}$$

We notice that thanks to polymorphism, it is impossible to index the argument arrays out-of-bounds at runtime. Shape polymorphism gives us safety for free, akin to Wadler [26].

Example. Consider incompatible shapes $s' = \{\!|\ell = \#2|\!\}$ and $s'' = \{\!|\ell = \#3|\!\}$. There is no clear way in which these shapes could be reconciled – value $\{\!|\ell = 2|\!\}$ lies only in the bounds of s'' – thus $s' \sqcap s''$ fails. We could set $\#n \bowtie \#m = \#\min(n, m)$ in the semantics – but this seems prone to surprising behaviour, as it implicitly clips indices in $[n, m)$ (when $n < m$).

4.3 Type Safety

To show our proposed design is well-behaved, we state and prove type safety for Star in the form of theorems of preservation and progress in the simply-typed case. We leave the case with parametric polymorphism as future work.

Star's type safety does not prevent runtime errors caused by out-of-bounds integer indexing and shape broadcasting on incompatible shapes. To reason about this, we extend our operational semantics with a *raises-error* judgement $e \rightsquigarrow \lightning$:

$$\textsc{Arr}(s, I)[v] \rightsquigarrow \lightning \text{ if } (v \blacktriangleleft s) \wedge \neg(v \lhd s) \qquad \textsc{StepIndexErr}$$
$$s' \sqcap s'' \rightsquigarrow \lightning \text{ if } s' \bowtie s'' \text{ undef.} \qquad \textsc{StepBroadcastErr}$$

where we used a structurally-in-bounds relation $v \blacktriangleleft s$, given by the same rules as $\lhd$ but with $n \blacktriangleleft \#m$ holding for **any** n and m (and not just $0 \le n < m$). Hence, an error is raised exactly when an *integer* index is out of bounds.

We can now state Preservation and Progress theorems.

Theorem 4.1 (Preservation). *If $\Gamma \vdash e : \tau$ and $e \rightsquigarrow e'$, then $\Gamma \vdash e' : \tau$.*

Theorem 4.2 (Progress). *If $\cdot \vdash e : \tau$, then either e is a value, $e \rightsquigarrow e'$ for some e', or $e \rightsquigarrow \lightning$.*

We prove these theorems in Appendix C.

4.4 Type Inference

We hypothesise **algebraic subtyping** can provide principal ML-style type inference. We provide a sketch of our approach, relying on the work of Parreaux and Chau [14]. The key requirement we satisfy is that our subtyping yields a distributive lattice algebra, which is claimed to be sufficient to apply these techniques by Dolan [6]. We are currently implementing this approach as part of a general framework.

The main difficulty in providing type inference for Star is handling the 'type-level function' ι. We claim constraint solvers for algebraic subtyping in the style of Parreaux [13]

Table 2. Table of Star features used for expressing a selection of common array programming patterns.

Pattern	Star Feature
Tensors	Product shapes
Padding	Concatenation shapes
Broadcasting	Shape broadcasting
Axis indexing	Record update & restriction

can be extended to handle applications of type lattice homomorphisms.[3] The mapping ι is such a homomorphism. Otherwise, we could encode the relation between shapes σ and indices ι via qualified types (e.g. a predicate ShapeHasIndex $\sigma\,\iota$), or a GADT (e.g. a polymorphic type (σ, ι, τ) arr). However, these approaches might not compose well with subtyping [7, 22], which is key for some patterns.

If algebraic subtyping proves to be the wrong approach, we note many of its key aspects can be replicated in a system with row polymorphism. Nevertheless, we stick to relying on algebraic subtyping in this paper, though noting that it is less established than the alternatives.

5 Examples

Star does not include primitives that obviously correspond to common array programming patterns. We consider various examples of such patterns in Star, all of which are statically typed and have an elegant presentation. Table 2 summarises the features of Star and the patterns they express.

5.1 Padding

Consider padding a matrix with l and r columns on the left and right respectively, and t and b top and bottom rows. Padding should have value -1 at top, $+1$ at bottom, and 0 elsewhere. For instance, for $l = r = t = b = 1$:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \mapsto \begin{bmatrix} -1 & -1 & -1 & -1 & -1 \\ 0 & 1 & 2 & 3 & 0 \\ 0 & 4 & 5 & 6 & 0 \\ +1 & +1 & +1 & +1 & +1 \end{bmatrix}$$

This can be inconvenient in point-free array programming (e.g. NumPy). An array of shape (n, m) becomes one of shape $(n + t + b, m + l + r)$. The programmer is burdened with keeping track of indices. To pad a NumPy array `x` one might write:

```
n, m = x.shape
p = numpy.empty((n + t + b, m + l + r))
p[t:n+t, l:m+l] = x[:, :]
p[:t, :] = -1;     p[n+t:, :] = +1
p[t:n+t, :l] = 0; p[t:n+t, n+l:] = 0
```

[3] This is part of our ongoing work extending Parreaux and Chau [14]. We push homomorphism applications through joins/meets onto variables, and then we transform inequalities so variables appear without an application.

Not only does the programmer need to do mental arithmetic to find the index ranges, they are also asymmetrical.

Now consider how we would implement padding in Star. Intuitively, the shape of a padded matrix has different *regions*, depending on which part of the padding we are in:

$$\begin{array}{c} \text{col} \\ \begin{bmatrix} \text{top left} & \text{top centre} & \text{top right} \\ \text{centre left} & \text{centre centre} & \text{centre right} \\ \text{bot left} & \text{bot centre} & \text{bot right} \end{bmatrix} \end{array} \text{row}$$

These correspond to index ranges used prior. For instance, centre right is in range $[t, n + t) \times [n + l, n + l + r)$.

This leads us to the shape type σ of a padded array:

$$\sigma = \{\!|\text{row} : [\![\text{Top} : \#, \text{Centre} : \#, \text{Bottom} : \#]\!], \\ \text{col} : [\![\text{Left} : \#, \text{Centre} : \#, \text{Right} : \#]\!]|\!\}$$

We construct padding of an array $a : [\{\!|\text{row} : \#, \text{col} : \#|\!\}]\text{int}$:[4]

$$\begin{aligned} &\text{let } s = \{\!|\text{row} = [\![\text{Top} = \#t, \text{Centre} = |a|_\circ\text{row}, \text{Bottom} = \#b]\!], \\ &\qquad \text{col} = [\![\text{Left} = \#l, \text{Centre} = |a|_\circ\text{col}, \text{Right} = \#r]\!]|\!\} \text{ in} \\ &\Phi\, x[s].\, \text{match } x \text{ with} \\ &\qquad \mid \text{Centre } i, \text{Centre } j \Rightarrow a[\{\text{row} = i, \text{col} = j\}] \\ &\qquad \mid \text{Top } \_, \_ \Rightarrow -1 \;\; \mid \text{Bot } \_, \_ \Rightarrow +1 \;\; \mid \_, \_ \Rightarrow 0 \end{aligned}$$

Satisfyingly, index arithmetic has become pattern matching.

5.2 Broadcasting

As advertised, we use *shape broadcasting* to model NumPy's broadcasting pattern. We agree with Chiang et al. [4] that the broadcasting pattern is easier to reason about when array dimensions are named. For a case study, consider Graph Neural Networks, which are implemented using tensors with four (or more) dimensions, like:

$$(\text{batch}, \text{source}, \text{target}, \text{feature})$$

Suppose we want to add an array with axes $(\text{source}, \text{target})$ to each entry of such a tensor. In NumPy, this is difficult to follow, as it requires introducing dimensions of size 1 using an operation like `numpy.expand_dims(..., axis=-1)`. The use of axis indexing obstructs the new shape of the array, and general type inference is a lost cause.

In Star, we can instead rely on named dimensions (in product shapes) and subtyping of record indices. We might have:

$$h : [\{\!|\text{batch} : \#, \text{source} : \#, \text{target} : \#, \text{feature} : \#|\!\}]\text{float}$$
$$h' : [\{\!|\text{source} : \#, \text{target} : \#|\!\}]\text{float}$$

and then implement the addition of h and h' as

$$\Phi\, p\big[|h| \sqcap |h|'\big].\, h[p] + h'[p]$$

We could take such a routine as the definition of a whole-array element-wise addition operator $\oplus$.

NumPy also allows scalars to be broadcast into an array, e.g. adding 3.14 to every element of a matrix. This effect is

[4] For brevity, we used a '_' to match on any tag, which is not in core Star.

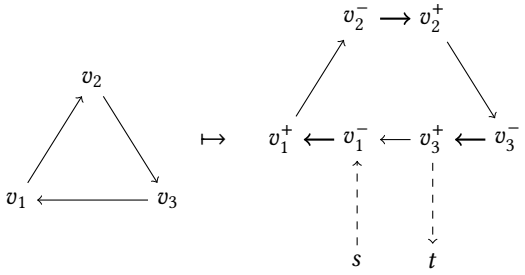


Figure 11. An example transformation from G to G', splitting vertices into inlets & outlets and adding a source & sink.

achieved in Star by wrapping the scalar in a zero-dimensional array (of size 1) – e.g. $\Phi\, i[\{\!|\}]$. 3.14. – and then applying $\oplus$.

5.3 Partial Indexing

To present one more interesting pattern in Star, we extend it with a *record update* primitive $\{\ell = e \mid e'\}$ which adds a field ℓ with value e to the record e'. We also add *record restriction* $e \setminus \ell$ which removes field ℓ from the record e. For brevity, we associate product shapes with records here.

We give a simple though nevertheless useful example of the usage of record updates and restrictions on indexing. Suppose we are given an array a with a product shape with dimension *batch*, and an index i : int of a batch we want to use. We can write the following expression and its type:

$$\lambda a.\, \lambda i.\, \Phi\, p[|a| \setminus \text{batch}].\, a[\{\text{batch} = i \mid p\}]$$
$$[\{\text{batch} : \sigma\} \wedge \sigma']\tau \to \iota(\sigma) \to [\sigma']\tau \quad (\text{batch} \notin \sigma')$$

By including record updates and restrictions, we can express indexing across an axis of a multi-dimensional array in a shape-polymorphic manner.

Typing record extension and restrictions under algebraic subtyping is ongoing work that we believe is also solved with lattice homomorphisms (like ι). However, these operations are known to be expressible with row polymorphism.

5.4 Indexing Schemes

Let us now consider a larger construction that could arise when constructing a graph problem (e.g., a maximum flow network). We show how structural array shapes provide a powerful mechanism for building abstractions.

Given a graph G represented as an adjacency list (array of arrays), we want to construct a graph G' with each vertex v split into ingoing v^- and outgoing v^+ sides, and with an extra source s and sink t. An example of this transformation is given in Figure 11.

Without structurally typed array shapes, to perform this construction (while still using arrays, e.g. for performance) we would need to come up with indexing scheme to use for G'. For instance, given a vertex v at index i in the array G

$$\begin{aligned}
G' = \Phi i\, [\{\!|v = [\![\mathrm{S} = \{\!|\}\!], \mathrm{V} = |G|]\!],\ s = [\![\mathrm{In}=\{\!|\}\!], \mathrm{Out} = \{\!|\}\!]\!|\}\!].\\
\quad \text{match } i \text{ with}\\
\quad | \{v = \mathrm{S}\,\{\}, s = \mathrm{In}\} \Rightarrow [\{v = \mathrm{V}\,1, s = \mathrm{In}\}]\\
\quad | \{v = \mathrm{S}\,\{\}, s = \mathrm{Out}\} \Rightarrow []\\
\quad | \{v = \mathrm{V}\,i, s = \mathrm{In}\} \Rightarrow [\{v = \mathrm{V}\,i, s = \mathrm{Out}\}]\\
\quad | \{v = \mathrm{V}\,i, s = \mathrm{Out}\} \Rightarrow\\
\qquad \text{let } a = (\Phi\, j[|G[i]|].\, \{v = \mathrm{V}\,G[i][j], s = \mathrm{In}\}) \text{ in}\\
\qquad \text{if } i = n - 1 \text{ then cons}(\{v = \mathrm{S}, s = \mathrm{Out}\}, a) \text{ else } a
\end{aligned}$$

Figure 12. Implementation of the graph transformation in Star. We used some utilities – cons and singleton array constructors – that are expressible in the Star calculus.

(written $v \mapsto i$), we could map $v^- \mapsto 2i + 2$, $v^+ \mapsto 2i + 3$, and set $s \mapsto 0$ and $t \mapsto 1$.

In Star, this construction is much simpler, and reminiscent of how a functional programmer would naturally apply algebraic data types in similar problems. Let us presume G : [#][#]int so that for each (u, i) we have an edge $u \to G[u][i]$. Let us write $\mathbf{1} = \{\!|\}$ as a shorthand. The shape of G' – or, really, the type of vertices – might be:

$$\sigma' = [\![\mathrm{Source} : \mathbf{1}, \mathrm{Sink} : \mathbf{1}, \mathrm{V} : \{\!|v : \#, s : [\![\mathrm{In} : \mathbf{1}, \mathrm{Out} : \mathbf{1}]\!]|\}\!]\!]$$

We could also use In and Out for the source and sink:

$$\sigma'' = \{\!|v : [\![\mathrm{S} : \mathbf{1}, \mathrm{V} : \#]\!], s : [\![\mathrm{In} : \mathbf{1}, \mathrm{Out} : \mathbf{1}]\!]|\}\!$$

so that the value of source s is given by $\{v = \mathrm{S}\,\{\}, s = \mathrm{In}\}$, the sink t by $\{v = \mathrm{S}\,\{\}, s = \mathrm{Out}\}$, vertices v_i^+ by $\{v = \mathrm{V}\,i, s = \mathrm{Out}\}$ and v_i^- by $\{v = \mathrm{V}\,i, s = \mathrm{In}\}$.

In either case, we have G' : $[\sigma][\#]\iota(\sigma)$. Clearly, a Star compiler could derive and use the same integer indexing scheme we showed before for G' from both σ' and σ''.

Star's encoding is pleasant, as it allows us to use types to express the structure of our new graph G'. Operating on this encoding is also easier than with raw integers. For instance, consider an operation to get the ingoing side of a vertex:

$$\text{match } e \text{ with } \mathrm{V}\,\{v = i, s = \_\} \Rightarrow \mathrm{V}\,\{v = i, s = \mathrm{In}\}$$

which maps vertices $V\,\{v = i, s = \mathrm{Out}\}$ to $V\,\{v = i, s = \mathrm{In}\}$. Using the proposed integer integer encoding the programmer might instead write `i - i%2` (or otherwise explicitly check parity), which maps $2i + 3$ to $(2i + 3) - [(2i + 3) \bmod 2] = 2i + 3 - 1 = 2i + 2$. Both representations achieve the same goal, but Star's makes productive use of algebraic data types.

The difference becomes stark in a longer example. Let us now specify that any edge $v_i \to v_j$ in G becomes an edge $v_i^+ \to v_j^-$ in G', and we have edges $v_i^- \to v_i^+$, $s \to v_0$ and $v_{n-1} \to t$. In Star, we elegantly implement this via pattern matching the program is presented in Figure 12. We contrast it with an explicit indexing scheme in Figure 13. The latter

```
n = |G|; G1 = [[] for _ in range(2*n+4)]
G1[0].append(2*0+2); G1[2*(n-1)+3].append(1)
for i in range(n):
  G[2*i+2].append(2*i+3)
  for j in range(|G[i]|):
    G1[2*i+3].append(2*G[i][j]+2)
```

Figure 13. Implementation of the same graph transformation with an explicit integer indexing scheme.

implementation lacks separation of concerns – the interface for G' depends on implementation details.[5]

The graph transformation example is interesting for another reason: it involves jagged arrays, which are often second-class citizens in array programming. Type systems might disregard them entirely and assume all arrays are regular. Star cleanly supports both regular arrays (product shapes) and jagged arrays of arrays.

We conclude that Star offers better capabilities for building abstractions in programs where arrays are used as an underlying data structure. Structurally typed array shapes parallel the usefulness of algebraic data types generally.

6 Future Work

Our key contribution is the design of Star. We are in the process of implementing a prototype, including our approach to type inference. There is still more work needed in order to show that polymorphism in Star is well-behaved, particularly in conjunction with subtyping – for now we solely rely on the work of Dolan [6] and Parreaux and Chau [14].

In this paper, we only described the crux of the idea of structural array shapes. In this section we describe further useful extensions that we can make to Star.

6.1 Shape Inference

A common desideratum for pointful array calculi is the ability to omit array shapes, which makes them much less verbose. In Star, this *shape inference* could be implemented as a type-directed program transformation.

We first explain shape inference by example. A prototypical pointful calculus is *Einstein notation*, which provides index-oriented specification of operations generalising matrix multiplication. Matrix multiplication $C = AB$ is written:

$$C_{i,k} = \sum_j A_{i,j} B_{j,k}$$

Note that we have omitted bounds on (i, j, k) – we presume them to span the full range of dimensions of A and B.

[5] One could say this could be improved by using functions for computing indices. This is not a complete argument – by the same logic one could argue that a `struct` type is not necessary in C, as it can be implemented by setter and getter functions on void pointers into buffers. By implementing indexing functions, the programmer copes with the language's limitations.

Let us consider omitting the shape in a Star array comprehension. Recall the outer-product example, where we are given $u : [\{\!|\text{row} : \sigma|\!\}]\text{float}, w : [\{\!|\text{col} : \sigma'|\!\}]\text{float}$ in the environment. Suppose we write:

$$\Phi\, i[].\, u[i] \times w[i]$$

The least upper bound on the type of i is $\{\text{row} : \sigma, \text{col} : \sigma'\}$. Since σ and σ' are now unconstrained shape types, the only way to obtain a (shape) value of type σ is to use $|u|_\circ\text{row}$. We can thus recover the shape value as:

$$\{\!|\text{row} = |u|_\circ\text{row}, \text{col} = |w|_\circ\text{col}|\!\}$$

In a more general case, for $u : [\sigma]\text{float}$ and $w : [\sigma']\text{float}$, we find $i \leqslant \iota(\sigma) \wedge \iota(\sigma')$. Since the index bound is found to be a meet of shape types, we use shape broadcasting to reconcile them. We would infer the shape:

$$|u| \sqcap |w|$$

so the most-general type of $\lambda w.\, \lambda u.\, \Phi\, i[].\, w[i] \times u[i]$ is:

$$[\sigma]\text{float} \to [\sigma']\text{float} \to [\sigma \wedge \sigma']\text{float}$$

(temporarily treating σ and σ' as shape type variables).

As in most similar approaches, we cannot infer the shape when index arithmetic is involved. Thus, $\Phi\, i[].\, a[i] + a[i+1]$ is illegal without a shape, as we only find $i \leqslant \text{int}$.

It is worth noting inference of index bounds is seen as a case of dependent type inference in Dex [16]. In Dex, index types are associated with shapes, and thus they can be inferred. Since Star is not dependently typed, we cannot mingle types and values like that, and we instead model the relationship via the ι type isomorphism. Thankfully, type inference relying on algebraic subtyping is sufficient for us to provide a similar mechanism. Most other approaches to shape inference are simple syntax-directed heuristics.

6.2 Nameless Shapes

Just as record types correspond to tuples, Star's (named) product shapes could correspond to unnamed product shapes. We might have shape type of form $(\!|\sigma, \ldots|\!)$, indexed by tuples $(\iota(\sigma), \ldots)$. In contrast to records, tuples do not have width subtyping – we only relate tuples of the same arity.

Unnamed dimensions allow building slightly less verbose abstractions – appropriate in cases where the dimensions of an array are strictly known and non-extensible.

6.3 Recursive Types

Consider the usual definition of the Fibonacci sequence:

$$F_{n+2} = F_{n+1} + F_n \quad F_1 = 1 \quad F_0 = 0$$

When expressing this as an array program, we would rely on a *scan* combinator, replacing the recursion with an iteration. It is an interesting question if we can preserve the recursive nature of the computation using *recursive index types*. Using a Peano-style encoding, we could have an index type like:

$$\mu\, t.\, [Z : \{\}, S : t]$$

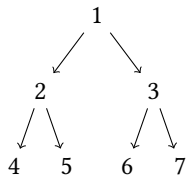


Figure 14. Traditional indexing scheme for a complete binary tree – node i has left child $2i$ and right child $2i + 1$.

We would like to define an array $F : [\mu\, t.\, [\![Z : \{\!|\,|\!\}, S : t]\!]]\text{int}$ of Fibonacci numbers using the standard recursive definition. We might do so by referring to its elements with structurally smaller indices, in an imaginary extension of Star:

$$\begin{array}{l}\text{let } k = S\,(S\,(S\,(S\,(S\,(Z\,\{\!|\,|\!\})))))\text{ in}\\ \text{let rec } F = \Phi\, m[k].\ \text{match } m \text{ with}\\ \qquad\qquad\qquad\qquad\quad \mid \mathrm{S}\,(\mathrm{S}\,n) \mapsto F[S\,n] + F[n]\\ \qquad\qquad\qquad\qquad\quad \mid \mathrm{S}\,\mathrm{Z} \mapsto 1 \mid \mathrm{Z} \mapsto 0\end{array}$$

which would compute F_0 through F_5, so that

$$F[S\,(S\,(S\,(S\,(Z\,\{\}))))] = F_4 = 5$$

However, it is unclear how this should work out in practice.

Finding a way to construct recursively defined arrays could help safe implementation of dynamic programming algorithms, which are inherently recursive. Another direction would be representation of complete n-ary trees. It would be interesting to generalise the construction of the standard indexing scheme for binary trees (Fig. 14), with index type:

$$\mu\, t.\, [\text{Leaf} : \{\}, \text{Branch} : \{\text{Left} : t, \text{Node} : \{\}, \text{Right} : t\}]$$

6.4 Stencils

Stencils are a common pattern that is not addressed well by the proposed shape types. For example, the operation:

$$a[i] \mapsto (a[i-1] + a[i] + a[i+1])/3$$

degenerates to setting $i : \text{int}$ in Star, forcing the programmer to e.g. explicitly perform checks for out-of-bounds values. Addressing this shortcoming is left as future work.

6.5 Implementation

We briefly consider whether arrays using Star's structurally typed shapes can be implemented efficiently.

All array shapes in Star are regular, hence any shape value s contains a fixed number of elements $\varepsilon(s)$. We claim that we can construct affine integer indexing schemes $\tilde{\imath}$ as such:

- Sizes $\#n$ are indexed by integers in $[0, n)$, i.e. $\tilde{\imath}(n) = n$.
- Product shapes $\{\!|\overline{\ell = s}|\!\}$ can use a row-major (column-major) representation, with axes ℓ of size $\varepsilon(s)$. For $\{\!|a = s_a, b = s_b, c = s_c|\!\}$ we might use the indexing:

$$\begin{array}{l}\tilde{\imath}(\{a = i_a, b = i_b, c = i_c\}) =\\ \quad \varepsilon(s_b)\varepsilon(s_c)\tilde{\imath}_a(i_a) + \varepsilon(s_c)\tilde{\imath}_b(i_b) + \tilde{\imath}_c(i_c)\end{array}$$

- Concatenations $[\![\overline{T = s}]\!]$ require an offset map for each component T. For $[\![S = s_s, T = s_t]\!]$ we could set:

$$\tilde{\imath}(S\ i) = \tilde{\imath}_s(i) \qquad \tilde{\imath}(T\ i) = \varepsilon(s_e) + \tilde{\imath}_t(i)$$

This sketches a compositional construction for indexing schemes, generalising *strides* (linear memory access descriptors) for n-dimensional arrays. A compiler with static knowledge of shapes could generate code applying these. Otherwise, strides and offsets would be part of array values as in NumPy.

Star's semantics do not differentiate between the ordering of labels (axes) in a product shape, and similarly for tags (components) in a concatenation shape. One possibility is to consider component order in shape *values* as a required implementation choice, differentiating between equivalent array memory representations. For example, a row-major representation of a 5×4 matrix could be written $\{\!|\text{row} = 5, \text{col} = 4|\!\}$, while column-major could be $\{\!|\text{col} = 4, \text{row} = 5|\!\}$. While the shape *values* could express ordering, ordering in shape *types* would likely reduce their usability – standing at odds with efficient separate compilation.

6.6 Real-World Applications

We consider the limitations that might arise when applying Star in practice.

While the type system relies on algebraic subtyping, much of the same features could be recovered with an HM-style system with classic row polymorphism [20, 27].

The type system is presented for a pointful array calculus – reminiscent of $\tilde{F}$ [24] or elements of Single Assignment C [23]. Our technique is thus not directly applicable to popular *point-free* array programming libraries applied in machine learning workloads and deriving from APL, such as NumPy or PyTorch. However, prior work [2] has shown that pointful array programs can be compiled to point-free ones. The programmer can use a statically typed pointful DSL, that executes using an established backend library.

We believe functional array languages like Futhark could benefit from elements of our system. Such work might uncover interesting difficulties in a performant implementation.

7 Related Work

We elaborate on the most important lines of prior work that inspired the design of Star.

7.1 Generalising Arrays

Named tensors [4] propose the notion of naming array axes. This provides a refined treatment of operations on multi-dimensional arrays, with more intuitive definitions of both broadcasting and operations along axes. These can be seen as a special case of our product shapes $\{\!|\overline{\ell : \#}|\!\}$.

Note that the ideas of naming axes have existed in several software packages, particularly `xarray`, but also `tsalib`. Similar ideas were present in `einops` [21].

7.2 Typing Arrays

Dex [16] is a dependently typed, *pointful* array programming language. Arrays are seen as functions from a domain – the index type – into the element type. For instance, `n => d => Float` are $n \times d$ matrices of floats. A crucial aspect of Dex's dependent types is that an array's shape (value) is given by its index type (since types can be used as values) – and this type can be, for instance, inferred. In Star, our shapes are simply values, and we relate shapes and indices via an isomorphism in the type lattice.

A system with product and concatenation shapes is interestingly close to the work of Colaço et al. [5] focusing on handling polynomial array sizes – product shapes multiply sizes, concatenations add them. This approach handles bounds checking statically, in contrast to Star. However, this does not help the programmer build abstractions so that such bounds checking is not necessary in the first place. We argue that the structure of data stored in arrays is richer than an expression giving its number of elements, and arrays should be typed in this nature.

A common motif in array type systems is whether they support *rank polymorphism*. In the case of Star, it is a special case of shape polymorphism.

7.3 Subtyping

There has been successful research on typing dataframe programs with structural record types and row polymorphism [17]. These are another source of inspiration for using a similar approach to typing array programs.

Algebraic subtyping, introduced by Dolan and Mycroft [7], along the recent developments to remove the *polarity restriction* due to Parreaux and Chau [14], are the key developments thanks to which Star admits ML-style type inference. This discipline also allows clean integration of parametric polymorphism and subtyping, the detailed investigation of which for Star we leave as future work.

8 Conclusions

We have presented a novel design for an array programming calculus, emphasising the use of structural types for array shapes. Though we do not statically ensure bounds checking for integer indices like dependent type systems, we circumvent the problem by enabling the programmer to use richer structures for their arrays' shapes (and indices), which we statically type check. Thanks to our approach, it is much easier to provide useful static types to common array programming patterns. Our novel type system for array programs fills in the gap between dynamically and dependently typed array programming.

Acknowledgments

We would like to sincerely thank the reviewers for their helpful advice on improving the paper. We are also grateful to Ines Wright and Charlie Lidbury for their feedback on draft versions of this paper. Orchard received support through Schmidt Sciences, LLC. and thanks the Institute of Computing for Climate Science for their support.

A Broadcasting

Broadcasting – as called in NumPy, though the pattern appears in many forms throughout array programming – can be seen as an instance of *rank polymorphism*.

It is common that we operate element-wise on data that does not have the same dimensions. A basic instance is scalar-matrix multiplication:

$$2 \times \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix}$$

Broadcasting generalises this pattern. Consider two arrays a and b with shapes s and z respectively (given by tuples of integers). Whenever a dimension's size s_i is 1, we instead 'promote' it to be equal to z_i, repeating elements ('broadcasting') along that dimension. For example, for shapes $s = (n, m, 1)$ and $z = (n, 1, k)$ we define $a + b$ as:

$$(a + b)_{i,j,k} = a_{i,j,\mathbf{0}} + b_{i,\mathbf{0},k}$$

Axes in **bold** were broadcast. The shape of $a + b$ is (n, m, k).

This is notoriously difficult to type statically, as e.g. adding matrices of shapes (n, m) and (n', m') yields one of shape $(\max(n, n'), \max(m, m'))$ but only if

$$(n = n' \vee n = 1 \vee n' = 1) \wedge (m = m' \vee m = 1 \vee m' = 1)$$

Mixing conjunctions and disjunctions of constraints like this is difficult to resolve in general.

B Variance of Array Types

We now explain why array types of the form $[\sigma]\tau$ are necessarily shape invariant (i.e. invariant in σ). Take an array $a : [\{\!|x : \#|\!\}]\text{float}$. Then:

- Suppose we have shape contravariance. Then

  $$[\{\!|x : \#|\!\}]\text{float} \leqslant [\{\!|x : \#, y : \#|\!\}]\text{float}$$

  since $\{x : \text{int}, y : \text{int}\} \leqslant \{x : \text{int}\}$. $a[\{x = 0, y = 0\}]$ is well-typed as wanted (we can always use an index subtype), but we also find that $|a| : \{\!|x : \#, y : \#|\!\}$, which is nonsense (where would a size for y come from?) as clearly we have $|a| : \{\!|x : \#|\!\}$ at most.
- Suppose we have shape covariance. Then

  $$[\{\!|x : \#|\!\}]\text{float} \leqslant [\{\!|\,|\!\}]\text{float}$$

  But while we safely get $|a| : \{\!|\,|\!\} \geqslant \{\!|x : \#|\!\}$, we would also admit $a[\{\}]$ as well-typed, which is unsound.

$$
\begin{aligned}
C\langle\Box\rangle ::= {} & \Box \\
& \mid \pi(\overline{v}, C\langle\Box\rangle, \overline{e}) \\
& \mid \text{let } x = C\langle\Box\rangle \text{ in } e \\
& \mid C\langle\Box\rangle\ e \mid v\ C\langle\Box\rangle \\
& \mid \left\{\overline{\ell = v}, \ell = C\langle\Box\rangle, \overline{\ell = e}\right\} \mid C\langle\Box\rangle.\ell \\
& \mid T\ C\langle\Box\rangle \mid \text{match } C\langle\Box\rangle \text{ with } \overline{\mid T\ x \implies e} \\
& \mid \Phi\, x[C\langle\Box\rangle].\ e \\
& \mid \textsc{Arr}(s, \{v \mapsto C\langle\Box\rangle\} \cup J) \\
& \mid C\langle\Box\rangle[e] \mid v[C\langle\Box\rangle] \mid |C\langle\Box\rangle| \\
& \mid \#C\langle\Box\rangle \\
& \mid \left\{\!\left|\overline{\ell = v}, \ell = C\langle\Box\rangle, \overline{\ell = e}\right|\!\right\} \mid C\langle\Box\rangle_{\circ}\ell \\
& \mid \left[\!\left[\overline{\ell = v}, \ell = C\langle\Box\rangle, \overline{\ell = e}\right]\!\right] \mid C\langle\Box\rangle_{\diamond} T \\
& \mid C\langle\Box\rangle \sqcap e \mid v \sqcap C\langle\Box\rangle
\end{aligned}
$$

Figure 15. Evaluation contexts C for Star, necessary for specifying the congruence rules for $\rightsquigarrow$.

Finally, notice that using Star's split array types $[\sigma_1..\sigma_2]\tau$:

$$[\{\!|x : \#|\!\}]\text{float} \leqslant [\{\!|x : \#, y : \#|\!\}..\{\!|\}]\text{float}$$

we face neither of the issues – we can freely index the array with indices $\{x : \text{int}, y : \text{int}\}$ and access the shape $\{\!|\}$.

C Proof of Type Safety

In this section, we give the proof of type safety theorems in the simply-typed case without parametric polymorphism, as claimed in Section 4.3. We begin by completing some necessary details of our system, give some lemmas, and finally prove the Progress and Preservation theorems.

These are largely an extension of the proofs given by Pierce [18] for $\lambda_{<:}$ (simply-typed lambda calculus with subtyping and records). We add variants and Star's arrays.

C.1 Formal Details

In order to give a complete proof of type safety, we first need to complete some details in our formalisation of Star.

For our operational semantics to be complete, we need congruence rules. We define evaluation contexts C in Fig. 15. We can then give the congruence rules – including one for the error-raising judgement $e \rightsquigarrow \lightning$ from Section 4.3:

$$\textsc{Cong}\ \frac{e \rightsquigarrow e'}{C\langle e\rangle \rightsquigarrow C\langle e'\rangle} \qquad \textsc{CongErr}\ \frac{e \rightsquigarrow \lightning}{C\langle e\rangle \rightsquigarrow \lightning}$$

Note we consider $\lightning$ to be neither an expression nor a value.

In the following text, $\Gamma \vdash e :: \tau$ states that the *strongest* (least in the type lattice) type of e is τ, defined:

$$\Gamma \vdash e :: \tau \iff (\forall \tau'.\, \Gamma \vdash e : \tau' \implies \tau \leqslant \tau')$$

For a value, any typing derivation that does not apply Sub yields the strongest type.

C.2 Lemmas

We start by formulating a standard substitution lemma – we also use it for array comprehensions.

Lemma C.1 (Substitution). *If $\Gamma \vdash e : \tau$ and $\Gamma, x : \tau \vdash e' : \tau'$, then $\Gamma \vdash [e/x]e' : \tau'$.*

Proof. By straightforward induction on the derivation of $\Gamma, x : \tau \vdash e' : \tau'$. □

We also include lemmas about values – their canonical form given their type and that they do not reduce.

Lemma C.2 (Canonical forms). *We have the following canonical forms for well-typed values v:*

- *If $\Gamma \vdash v : \text{int}$, then $v = n$.*
- *If $\Gamma \vdash v : \text{float}$, then $v = f$.*
- *If $\Gamma \vdash v : \tau \to \tau'$, then $v = \lambda x.\, e$.*
- *If $\Gamma \vdash v : \left\{\overline{\ell : \tau}\right\}$, then $v = \left\{\overline{\ell = v_\ell}, \ldots\right\}$ (up to ordering).*
- *If $\Gamma \vdash v : \left[\overline{T : \tau_T}\right]$, then $v = T\, v_T$. In particular, we have $\Gamma \vdash v_T : \tau_T$ and $\Gamma \vdash v :: [T : \tau_T]$.*
- *If $\Gamma \vdash v : [\sigma_1..\sigma_2]\tau$, then $v = \textsc{Arr}(s, I)$. In particular, there exist σ such that $\sigma_1 \leqslant \sigma \leqslant \sigma_2$ and $\Gamma \vdash v :: [\sigma]\tau$.*
- *If $\Gamma \vdash v : \#$, then $v = \#n$.*
- *If $\Gamma \vdash v : \left\{\!\left|\overline{\ell : \tau}\right|\!\right\}$, then $v = \left\{\!\left|\overline{\ell = v_\ell}, \ldots\right|\!\right\}$ (up to ordering).*
- *If $\Gamma \vdash v : \left[\!\left[\overline{\ell : \tau}\right]\!\right]$, then $v = \left[\!\left[\overline{\ell = v_\ell}, \ldots\right]\!\right]$ (up to ordering).*

Proof. By inversion on the typing derivation $\Gamma \vdash v : \tau$, eliminating impossible cases which do not match the syntax of values. Where the implicit subtyping rule is applied at the end of a derivation, we proceed from the subtype.

Note that as noted canonical forms might have a stronger type than advertised by the well-typedness assumption – in particular, records (analogously products and concatenations) might contain more fields than in their type (we denote these by $\ldots$). This follows from the subtyping rules. □

Lemma C.3 (Values are normal forms). *For any value v there exist no e such that $v \rightsquigarrow e$, nor $v \rightsquigarrow \lightning$.*

Proof. By routine check of axioms and rules of $\rightsquigarrow$. □

We then assert that our subtyping relation is well-behaved.

Lemma C.4 (Subtyping lattice). *The relation $\leqslant$ together with appropriate meets $\wedge$ and joins $\vee$ forms a distributive lattice $(\leqslant, \top, \bot, \wedge, \vee)$.*

Proof. By routine check of all the axioms of a distributive lattice. We also need to construct meets $\wedge$ and joins $\vee$ and show that they agree with the subtyping order.

We briefly consider just the cases specific to Star for the construction of $\wedge$ and $\vee$. Product and concatenation shapes behave exactly as records under subtyping, so their their

meet takes the union of fields and the elementwise meet of field types. For arrays:

$$\left[\sigma_1'..\sigma_2'\right]\tau' \wedge \left[\sigma_1''..\sigma_2''\right]\tau'' \stackrel{\text{def}}{=} \left[(\sigma_1' \vee \sigma_1'')..(\sigma_2' \wedge \sigma_2'')\right](\tau' \wedge \tau'')$$

Joins are constructed analogously. □

Due to transitivity of $\leqslant$ (by Trans), we can replace repeated application of Sub with one. Subtyping forming a distributive lattice is also a necessary assumption for algebraic subtyping to apply.

Lastly, we specify crucial lemmas about indexing – particularly the correspondence of the structurally-in-bounds relation $\blacktriangleleft$ with typing.

Lemma C.5 (Typing agrees with indexing). *$\cdot \vdash s :: \sigma$ and $\cdot \vdash v : \iota(\sigma)$ if and only if $v \blacktriangleleft s$.*

Proof. Right to left is a straightforward check by structural induction on v and s. The other direction follows by structural induction on the typing derivations of s and v, checking that appropriate cases of $\blacktriangleleft$ follow. Avoiding loss of information about s by considering its strongest type ensures that v contains all the necessary dimensions (and possibly more). □

It is worth noting that given an array of type $[\sigma_1..\sigma_2]\tau$, for its shape's strongest type σ it follows $\sigma_1 \leqslant \sigma \leqslant \sigma_2$. Thus, for any index of type $\eta \leqslant \iota(\sigma_1)$ we also have $\eta \leqslant \iota(\sigma)$ and the lemma holds for v and the array's shape – this is key to our proof of Progress.

Lemma C.6 (Casting is idempotent). *If $v \lhd s$, then*

$$(v \oslash s) \oslash s = v \oslash s$$

Lemma C.7 (In-bounds relations agree with indexing). *If $v \lhd s$, then $(v \oslash s) \lhd s$ – in particular, $(v \oslash s) \trianglelefteq s$.*

As a consequence, if an index v occurs in the bounds of the shape s ($v \lhd s$) of an array $\textsc{Arr}(s, I)$, then StepIndex is sound (as $v \oslash s$ is indeed in the domain of I).

Lemma C.8 (Covariance of ι). *If $\sigma \leqslant \sigma'$, then $\iota(\sigma) \leqslant \iota(\sigma')$.*

Proof. By structural induction on σ and σ', with the proof following by rules of $\leqslant$ and definition of ι. □

C.3 Theorems

Theorem C.9 (Preservation). *If $\Gamma \vdash e : \tau$ and $e \rightsquigarrow e'$, then $\Gamma \vdash e' : \tau$.*

Proof. We perform induction on $\Gamma \vdash e : \tau$. A common case is where we find the reduction $e \rightsquigarrow e'$ derived from Cong, we can usually finish by inductive hypothesis – which states the subject of Cong preserves the type. The most interesting case is Array, where we have to carry through the type of each array element via the Substitution lemma.

Case Sub We have $\Gamma \vdash e : \tau'$ for some $\tau' \leqslant \tau$. Since $e \rightsquigarrow e'$ we can invoke the inductive hypothesis and obtain $\Gamma \vdash e' : \tau'$ immediately, and finish by Sub.

Case Var Impossible – no $\rightsquigarrow$ rule applies to variables.

Case Int, Float Cannot happen – e is a value.

Case Let By inversion on $\rightsquigarrow$, either StepLet or Cong apply. In the former case, we finish by Substitution.

Case Lambda Cannot happen – e is a value.

Case Apply By inversion on $\rightsquigarrow$, we reduce by StepApply – and can finish by Substitution.

Case Record By inversion on $\rightsquigarrow$, only Cong applies and we finish immediately.

Case RecordProj By inversion on $\rightsquigarrow$, StepRecordProj or Cong apply to our expression $e = e^*.\ell$. In the former, we must have $e^* = v^* = \overline{\{\ell = v_\ell\}}$, so that $\Gamma \vdash e : \{\ell : \tau_\ell\}$, $\tau = \tau_\ell$ and $e' = v_\ell$. Since e is well-typed, we must have $\Gamma \vdash v_\ell : \tau_\ell$ as required.

Case Tag By inversion on $\rightsquigarrow$, only Cong applies.

Case Match Similarly to the dual RecordProj. By inversion, we either have Cong (immediate) or StepMatch, in which case $e = \text{match}\, T\, v \,\text{with}\, \overline{T \Rightarrow e_T}$ for some value v (canonical form $T\, v$ of the match target). Since $e \rightsquigarrow e'$, it must be that $e' = [v/x]e_T$ (StepMatch) and we finish by Substitution at $\Gamma, x : \tau_T \vdash e_T : \tau$ (assumption of Match).

Case Array We have $\tau = [\sigma]\tau^*$. By inversion on $\rightsquigarrow$, either we evaluate the shape via Cong (and finish immediately), or return an array via StepArray. We check the latter – $e = \Phi\, x \left[e^{\mathrm{S}}\right].\, e^*$, $e' = \textsc{Arr}(s, J)$, and we have the inductive hypothesis at each element $v \mapsto e_v$ in J. By the inductive case e is typed via Array, so we have a well-typed shape value $\Gamma \vdash s : \sigma$ and know $\Gamma, x : \iota(\sigma) \vdash e^* : \tau^*$. StepArray reduces via indices v such that $v \lhd s$, from which we know it follows $\Gamma \vdash v : \iota(\sigma)$. By the inductive hypothesis and Substitution at $e_v = [v/x]e^*$ it thus follows that $\Gamma \vdash e_v : \tau^*$ at all v. We thus ascribed the right types to both the shape s and all elements $v \mapsto e_v$ of J and we are done via ArrayLit.

Case Index By inversion on $\rightsquigarrow$ we either have Cong or StepIndex, checking the latter. In that case $e = v[e^\circ]$ where $\Gamma \vdash v : [\sigma_1..\sigma_2]\tau$ and $\Gamma \vdash e^\circ : \iota(\sigma_1)$. we know $v = \textsc{Arr}(s, I)$ (canonical form). Since v must be well-typed (by inversion, with ArrayVal), each value in I – including v' – must have type τ, finishing the case.

Case Shape Similarly to Index – in StepShape we must have the canonical form $e = v = \textsc{Arr}(s, I)$, with $\Gamma \vdash e : [\sigma_1..\sigma_2]\tau$, $\Gamma \vdash v : [\sigma]\tau$ for some $\sigma_1 \leqslant \sigma \leqslant \sigma_2$, and $e' = s$. But then we immediately have $\Gamma \vdash s : \sigma$ and $\sigma \leqslant \sigma_2$ as required so $\Gamma \vdash s : \sigma_2$ by Sub.

Case Broadcast We check StepBroadcast. By the definition of broadcasting we find that if it does not get stuck, the result must be of type $\sigma \wedge \sigma'$.

Case Sized By inversion on $\rightsquigarrow$, only Cong applies.

Case Product, Concat Analogously to Record.

Case ProductProj, ConcatProj Like RecordProj.

□

Theorem C.10 (Progress). *If $\cdot \vdash e : \tau$, then either e is a value, $e \rightsquigarrow e'$ for some e', or $e \rightsquigarrow \lightning$.*

Proof. As before, we apply induction on $\cdot \vdash e : \tau$. Note that e is necessarily closed, since Γ is empty. Where we say we can proceed by Cong, CongErr might instead apply if an error is raised – this still meets the condition in the statement of the theorem. The most interesting case is **Index**, where we make use of the lemma that typing agrees with the structurally-in-bounds relation $\blacktriangleleft$.

Case Sub Finish by immediate application of the inductive hypothesis at the assumption of Sup, $\cdot \vdash v : \tau' \geqslant \tau$.

Case Var Impossible – no $\rightsquigarrow$ rule applies to variables.

Case Int, Float Immediate, since e must be a value.

Case Let If the let-bound expression is not a value, then we finish by the inductive hypothesis and Cong. Otherwise, we finish by StepLet.

Case Lambda Immediate, since $e = \lambda x.\, e'$ is a value.

Case Apply Let $e = e'\ e''$. If either e' or e'' are not values, then Cong applies. Otherwise $e = v'\ v''$. But $\cdot \vdash v' : \tau'' \to \tau$ (assumption of Apply) and we know v' is in canonical form $v' = \lambda x.\, e^*$, so we can finish by StepApply.

Case Record If not all expressions occurring in fields are values, we can proceed by Cong. Otherwise, e is already a value.

Case RecordProj Let $e = e^*.\ell$. If e^* is not a value, we can apply Cong. Otherwise, $e^* = v^*$ and by assumption of RecordProj, $\cdot \vdash v^* : \{\ell : \tau\}$, so $v^* = \{\ell = v, \ldots\}$. Thus ℓ occurs in v^* and we finish by StepRecordProj.

Case Tag Similarly to Record: if the tagged expression is not a value, we Cong. Otherwise, e is a value.

Case Match Dually to RecordProj: since the expression is well-typed, the target of the match is a tagged value that does occur in the case list, and we can finish by StepMatch.

Case Array Let us assume shape s is already a value (otherwise we have a Cong) in $e = \Phi\, x[s].\, e^*$ with $\tau = [\sigma]\tau$ and $\cdot \vdash s : \sigma$, $\cdot \vdash e : \tau$. In that case, we can immediately finish by StepArray.

Case ArrayLit Either the array literal is already a value – if all the elements in J have been evaluated – or we can evaluate by Cong.

Case Index Without loss of generality, let us take both operands to be values – otherwise, Cong applies. Let $e = v[v']$ so that $\cdot \vdash v : [\sigma_1..\sigma_2]\tau$ and $\cdot \vdash v' : \iota(\sigma_1)$ (assumptions of Index). Then $v = \textsc{Arr}(s, I)$ (canonical), so that in particular $\cdot \vdash s :: \sigma$ and $\cdot \vdash v : [\sigma]\tau$ where $\sigma_1 \leqslant \sigma \leqslant \sigma_2$. But since $\sigma_1 \leqslant \sigma$ and thus $\iota(\sigma_1) \leqslant \iota(\sigma)$, by Sub we also have $\cdot \vdash v' : \iota(\sigma)$. Invoking the agreement of typing and in-bounds lemma on v' and s, we immediately get that $v' \blacktriangleleft s$. There remain two cases: either $v' \lhd s$, in which case we finish by StepIndex as v' must be in the domain of I (each index v^* in the domain has $v^* \lhd s$ thus $v^* \blacktriangleleft s$). Otherwise, v' is not in-bounds up to integer indices, i.e. $\neg(v' \lhd s) \wedge (v' \blacktriangleleft s)$, and we get StepIndexErr.

Case Shape Let $e = |e^*|$. If e^* is not a value, Cong. Otherwise, StepShape.

Case Broadcast If both operands are not already values, Cong. Otherwise, StepBroadcast or StepBroadcastErr apply – with the two cases exclusive, depending on whether $\sqcap$ is defined.

Case Sized If e is not already a value, we apply Cong.

Case Product, Concat Like Record.

Case ProductProj, ConcatProj Like RecordProj.

□

References

[1] Michael Gordon Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. 2003. Derivatives of Containers. In *Typed Lambda Calculi and Applications, 6th International Conference, TLCA 2003, Valencia, Spain, June 10-12, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2701)*, Martin Hofmann (Ed.). Springer, 16–30. doi:10.1007/3-540-44904-3_2

[2] Jakub Bachurski and Alan Mycroft. 2024. Points for Free: Embedding Pointful Array Programming in Python. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming, ARRAY 2024, Copenhagen, Denmark, 25 June 2024*, Aaron Hsu and Artjoms Sinkarovs (Eds.). ACM, 1–12. doi:10.1145/3652586.3663312

[3] Lubin Bailly, Troels Henriksen, and Martin Elsman. 2023. Shape-Constrained Array Programming with Size-Dependent Types. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing, FHPNC 2023, Seattle, WA, USA, 4 September 2023*, Gabriele Keller and Sam Westrick (Eds.). ACM, 29–41. doi:10.1145/3609024.3609412

[4] David Chiang, Alexander M. Rush, and Boaz Barak. 2023. Named Tensor Notation. *Trans. Mach. Learn. Res.* 2023 (2023). https://openreview.net/forum?id=hVT7SHlilx

[5] Jean-Louis Colaço, Baptiste Pauget, and Marc Pouzet. 2023. Polymorphic Types with Polynomial Sizes. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming, ARRAY 2023, Orlando, FL, USA, 18 June 2023*, Troels Henriksen and Artjoms Sinkarovs (Eds.). ACM, 36–49. doi:10.1145/3589246.3595372

[6] Stephen Dolan. 2017. *Algebraic subtyping.* PhD thesis. University of Cambridge.

[7] Stephen Dolan and Alan Mycroft. 2017. Polymorphism, subtyping, and type inference in MLsub. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 60–72. doi:10.1145/3009837.3009882

[8] Jeremy Gibbons. 2017. APLicative Programming with Naperian Functors. In *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 (Lecture Notes in Computer Science, Vol. 10201)*, Hongseok Yang (Ed.). Springer, 556–583. doi:10.1007/978-3-662-54434-1_21

[9] 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. 2020. Array programming with NumPy. *Nat.* 585 (2020), 357–362. doi:10.1038/S41586-020-2649-2

[10] Troels Henriksen and Martin Elsman. 2021. Towards size-dependent types for array programming. In *ARRAY 2021: Proceedings of the 7th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming, Virtual Event, Canada, 21 June, 2021*, Tze Meng Low and Jeremy Gibbons (Eds.). ACM, 1–14. doi:10.1145/3460944.3464310

[11] Stephan Hoyer and Joe Hamman. 2017. xarray: ND labeled arrays and datasets in Python. *Journal of Open Research Software* 5, 1 (2017), 10–10.

[12] Kenneth E. Iverson. 1962. A programming language. In *Proceedings of the 1962 spring joint computer conference, AFIPS 1962 (Spring), San Francisco, California, USA, May 1-3, 1962*, G. A. Barnard III (Ed.). ACM, 345–351. doi:10.1145/1460833.1460872

[13] Lionel Parreaux. 2020. The simple essence of algebraic subtyping: principal type inference with subtyping made easy (functional pearl). *Proc. ACM Program. Lang.* 4, ICFP (2020), 124:1–124:28. doi:10.1145/3409006

[14] Lionel Parreaux and Chun Yin Chau. 2022. MLstruct: principal type inference in a Boolean algebra of structural types. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 449–478. doi:10.1145/3563304

[15] 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 Z. Yang, Zachary 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. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 8024–8035. https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html

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

[17] Adam Paszke and Ningning Xie. 2023. Infix-Extensible Record Types for Tabular Data. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Type-Driven Development, TyDe 2023, Seattle, WA, USA, 4 September 2023*, Youyou Cong and Pierre-Évariste Dagand (Eds.). ACM, 29–43. doi:10.1145/3609027.3609406

[18] Benjamin C. Pierce. 2002. *Types and programming languages*. MIT Press.

[19] François Pottier. 1998. *Type Inference in the Presence of Subtyping: from Theory to Practice*. Research Report RR-3483. INRIA. https://inria.hal.science/inria-00073205

[20] Didier Rémy. 1989. Type checking records and variants in a natural extension of ML. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 77–88.

[21] Alex Rogozhnikov. 2022. Einops: Clear and Reliable Tensor Manipulations with Einstein-like Notation. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net. https://openreview.net/forum?id=oapKSVM2bcj

[22] Gabriel Scherer and Didier Rémy. 2013. GADTs Meet Subtyping. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7792)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer, 554–573. doi:10.1007/978-3-642-37036-6_30

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

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

[25] Justin Slepak, Olin Shivers, and Panagiotis Manolios. 2014. An Array-Oriented Language with Static Rank Polymorphism. In *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 (Lecture Notes in Computer Science, Vol. 8410)*, Zhong Shao (Ed.). Springer, 27–46. doi:10.1007/978-3-642-54833-8_3

[26] Philip Wadler. 1989. Theorems for Free!. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*, Joseph E. Stoy (Ed.). ACM, 347–359. doi:10.1145/99370.99404

[27] Mitchell Wand. 1987. Complete Type Inference for Simple Objects. In *Proceedings of the Symposium on Logic in Computer Science (LICS '87), Ithaca, New York, USA, June 22-25, 1987*. IEEE Computer Society, 37–44. http://www.ccs.neu.edu/home/wand/papers/wand-lics-87.pdf

Received 2025-04-01; accepted 2025-04-19