Ch2 — Introduction

Contents

Chapter 1

Structural Under: A Catamorphic Lens for GPU Subgroup Operations

> "It is true that a recursive data structure which is held in a conventionally addressed main store will usually be represented by means of machine addresses, but it seems a good idea that the programmer should be encouraged to ignore the machine-oriented details of the representation (just as he ignores details of the implementation of integers and of recursive procedures), and should concentrate on the more pleasant abstract properties of the structure." (Hoare 1975)

This chapter explores the abstraction of "machine-oriented details" of device operations of GPUs for general-purpose computing. The approach used is based on recursion schemes. The motivation is, in part, due to how list-homomorphisms translate into parallel divide-and-conquer computation. Such computation views work in smaller pieces, applies an update under such view, and then inverts the view. As GPUs store the mapped memory in the heap, this pattern is a shape-preserving bidirectional transformation. These transformations are constrained by hardware constants that vary between manufacturers. One such constraint is the so-called subgroup size, the width

of the SIMD bundle. Abstracting programs to account for subgroup width variability is thus a step towards heterogeneous computation on GPUs.

Subgroup operations are the GPU equivalent of the single-instruction, multiple-data (SIMD) paradigm of CPUs. These instructions yield better data-parallelism at the cost of more complex instructions. Thus, the abstraction is about closing the gap between optimal programs for the *computer to execute* and optimal programs for the *programmer to write*. What we strive for is an algorithmic treatment of a GPU:

> "It is important to note that the notation is executed directly, and the user need learn nothing about the computer itself." (p. 337 (Iverson 1972))

In the thesis context, this reading is about the *homogeneity* of GPU operations. To achieve this, we draw further motivation regarding abstraction of recursive patterns:

> "The computer is an important tool of mathematics. Nevertheless, it is essential that it be treated as a tool, and that the details of its use not be permitted to dominate or obscure the treatment of mathematical topics. For example, it is important in algebra to introduce matters such as iteration and function definition as fundamental mathematical notions and not as ingenious tricks for making use of a computer." (p.346 (Iverson 1972))

The tools to bridge this gap do have algebraic correspondences: a catamorphism paired with its own input (Meertens 1992), lens calculus (Foster et al. 2005), and a shape-preserving inverse combinator known as structural Under (Lochbaum 2020).

A **paired fold** is a catamorphism that carries untouched input alongside its result. It folds a structure while keeping the unconsumed remainder in hand. Such folds address descending a memory hierarchy: addressing a slice of a buffer, or a lane within a subgroup, is structural recursion over a nested carrier.

Bidirectional programming is the study of data transformations that can be run forwards and backwards. Its canonical reference point is the lens calculus, which rests on the constant-complement view-update theory (Bancilhon and Spyratos 1981). A lens focuses on a part of a whole, edits it, and reassembles the rest unchanged. The *complement* is everything outside the focus. In our context, this means that safe subgroup operations should not interfere with each other's memory slices.

The heap-based memory model of GPUs refines subgroup operations into instantiations of **structural Under**. This is a variant of an inverse-combinator called Under (Kromberg 2023), introduced in BQN (Lochbaum 2020). Structural Under is shape-preserving. Our approach proves shape-preservation extrinsically per operation rather than building it into the combinator. To our knowledge, structural Under has not been formalised in a dependently typed setting so far.

The contribution of this chapter rests on the identification of iterated structural Under as the form of an addressed SIMD operation. From it follow the use of the lens complement as the memory-preservation invariant, an instruction factorisation, and a derivation of the frame and parallel rules of disjoint concurrent separation logic (O'Hearn 2007; Brookes 2007) from the lens laws. The combination has properties that we mechanise in Lean 4:

1. **Memory-safety.** An operation in a given context cannot touch out-of-scope values. At the lens level, the **Complement-preservation theorem** (Theorem 1.11) ensures an update through a lawful decomposition cannot write its complement for any instruction. At the iterate level, the **effect-locality theorem** (Theorem 1.16) ensures the user instruction applies only at its position. The remainder is carried by `id`. All other positions are returned unchanged.

2. **Content-independence separates schedule from instruction.** The schedule specifies which slice an operation addresses. The instruction specifies what operation runs on that slice. The **Factorisation theorem**

(Theorem 1.17) formalises this separation. The typed dispatch of Section 1.4 implements it by treating schedules as selections and instructions as admitted operations. This yields a useful separation of schedules and instructions for program optimisation.

3. **Disjoint concurrency from the complement.** Two selections are independent when a joint decomposition exhibits both over a shared frame. This makes independence a proof witness. Independent operations commute for every pair of instructions (Theorem 1.13). The parallel rule of disjoint concurrent separation logic follows with side conditions discharged structurally (Theorem 1.14). A race is refuted by the absence of an independence witness. As our scope is the disjoint fragment, subgroups run in lockstep over total functions and there is no interleaving to schedule. Resource invariants and ownership transfer remain out of scope.

The first two rest on a Lambek-isomorphic core. Here, the factorisation is generic over any safe inductive carrier, while effect-locality is proved at a list carrier. The third is stated one level more abstractly: over the lens decompositions themselves. This construction is instantiated at the chunked carrier, where sibling subgroups are independent by construction. Lane, subgroup, and workgroup, the levels of the GPU execution hierarchy, are therefore instances of one framework rather than separate developments.

Three neighbouring lines of work exist. Section 1.5 compares against each in detail.

Inverse-based GPU memory mapping constructs the reassembling inverse manually for each layout (Janssen and Scholz 2021). Here the reassembly is the lens's backward direction. It is correct by the discharged law.

Isabelle/UTP models program variables as lenses and states lens-based frame rules (Foster et al. 2016; Huerta y Munive et al. 2024). Here independence is derived from the named complement rather than defined by commutation. This yields the parallel rule.

GPU memory safety has been discharged per kernel by SMT-backed concurrent separation logic (Martínez et al. 2026). It has been enforced in the type system by ownership and borrow checking (Köpcke et al. 2024). Here separation is inherited from the type-driven denotational structure.

The rest of the chapter is organised as follows.

Section 1.1.1 gives the device referent and the operations to be modelled. This includes SIMD peer access, the SPIR-V execution hierarchy, and four subgroup idioms on an example buffer. Section 1.1.2 introduces structured recursion with a paired fold's access to the original substructure, and the safe Lambek encoding. Section 1.1.3 presents lens calculus and its laws, defines constant complement, and elaborates BQN's Under read as conjugation.

Section 1.2 types Under by stratifying the Undo pair it conjugates by. A three-level taxonomy of Undo is formalised and connected to lens laws. The section closes by deriving the frame and parallel rules of disjoint concurrent separation logic from the constant complement.

Section 1.3 presents iterated Under as a single fold with a content-independent step, generic over the carrier, delivering effect-locality (Theorem 1.16) as the iterate-spelling counterpart of complement preservation, and closes with the schedule/instruction split.

Section 1.4 assembles the device story: the shape-preservation certificate that admits an operation, the admitted operations classified by channel budget, the typed dispatch of selections and admitted operations, independence instantiated at sibling subgroups with its race refutation, the read-footprint boundary, and a workgroup reduction composed of structural Unders.

Section 1.5 situates the work against related work, and states the limitations and conclusion.

1.1 Background: subgroups, recursion, and lenses

1.1.1 GPU subgroup operations

Programming with a GPU follows the device-host programming model (Section **??**). The host layer covers CPU-side ceremony and scheduling such as device enumeration, queue management, buffer allocation and transfers, and fences. It is an ordered, synchronised world of system calls, monadic in the sense of Moggi (1991). This layer could be modelled using T-algebras, but we bracket it for future work, and instead focus on the device side.

The device side is the total world of SIMD operations. A crucial capability of these vector instructions is peer access. Within a SIMD bundle, lanes can read and combine one another's registers in a single group operation. This makes a subgroup a special unit that shares memory by warping registers into a single operation.

The SPIR-V intermediate language organises execution into four nested levels:

$$\text{invocation} \subset \text{subgroup} \subset \text{workgroup} \subset \text{dispatch}$$

An invocation is a single thread indexed by a global identifier. A subgroup is the SIMD bundle of invocations that admits non-uniform group operations (peer access). A workgroup is a set of subgroups sharing local memory. The dispatch is the whole launch.

The four levels can be read as rising dimensions. An invocation is a scalar. A subgroup is a vector. A workgroup is a matrix. The dispatch is a cube. The hierarchy is nested, and an operation can be addressed to any level: a single lane, one subgroup, or one workgroup. The buffer ranges over all of them.

Next, we introduce some examples from the viewpoint of a subgroup. A particular property of subgroups is that the width is set by the hardware target instead of the program. E.g., an NVIDIA "warp" is 32 lanes wide, an AMD

"wavefront" 32 or 64, and an Intel subgroup 8, 16, or 32[1]. Following the SPIR-V specification, we call all these variations a subgroup. This is also indicative of why SPIR-V is device-agnostic: the intermediate representation is designed to accommodate hardware variance rather than assume constants.

A common challenge in working with subgroups is that the execution hierarchy raises a preservation concern. A device buffer is a flat array. An operation addressed to one level of the hierarchy writes its result into that level's slice and must leave the rest of the buffer intact. For instance, a subgroup reduction should produce its sum within the addressed subgroup. The neighbouring subgroups that share the buffer should remain undisturbed. The grouping is logical store. This means that a group operation in the subgroup context shares memory between lanes, yet each lane writes its result back into the flat memory buffer governed by its coordinate identity. Read access and write footprint diverge here: a group operation may read across a level to combine peer registers, but its effect on memory must stay confined to the addressed slice. This is the constant complement property (Bancilhon and Spyratos 1981) read as a hardware obligation: the unaddressed remainder of the buffer is a lens complement C. Preserving it is the put-side law.

The operations the chapter models are the standard SIMD device idioms. As an example, consider a pre-formatted buffer of four four-lane subgroups

$$[[0,1,2,3],\ [4,5,6,7],\ [8,9,10,11],\ [12,13,14,15]]$$

read as *subgroups* $\times$ *lanes*: the outer list indexes subgroups, the inner indexes the lanes within one. Next, consider operations that span different idioms. Indexing starts at zero.

Single-lane write. Adding `100` to lane `2` of subgroup `1` touches one cell and leaves the others untouched:

$$[[0,1,2,3],\ [4,5,\mathbf{106},7],\ [8,9,10,11],\ [12,13,14,15]]$$

[1]For a comprehensive table, see: https://vulkan.gpuinfo.org/displaycoreproperty.php?name=subgroupProperties.subgroupSize&platform=all

The coordinate $(1, 2)$ is fixed in advance; the only free ingredient is the cell instruction.

Subgroup-leader broadcast. Broadcasting subgroup `2`'s leader (lane `0`) across its lanes reads one lane and writes all four of the same subgroup:

$$[[0, 1, 2, 3],\ [4, 5, 6, 7],\ [\mathbf{8}, \mathbf{8}, \mathbf{8}, \mathbf{8}],\ [12, 13, 14, 15]]$$

This is peer access: the operation inspects a neighbour's value within the subgroup before writing.

Position-aware offset. Adding each subgroup's own index to its lanes varies with position: subgroup `0` gets `+0`, subgroup `1` gets `+1`, and so on:

$$[[0, 1, 2, 3],\ [5, 6, 7, 8],\ [10, 11, 12, 13],\ [15, 16, 17, 18]]$$

The transformation depends on *where* it is applied, not only on the values it sees.

Uniform broadcast. When the same operation applies to *every* subgroup there is no chosen index. Broadcasting every leader gives:

$$[[0, 0, 0, 0],\ [4, 4, 4, 4],\ [8, 8, 8, 8],\ [12, 12, 12, 12]]$$

and replacing every subgroup's lane `3` with its leader gives:

$$[[0, 1, 2, 0],\ [4, 5, 6, 4],\ [8, 9, 10, 8],\ [12, 13, 14, 12]]$$

Notice that this is uniform within the context of a subgroup, and no subgroup consults another.

Shared properties of these examples include:

- every operation writes only its addressed slice
- *which* slice an operation touches is fixed independently of *what* it computes there: the coordinates come from the launch configuration
- the operations vary in what a lane must *read* to compute its result:
  - nothing beyond its own cell (the single-lane write)

  - a peer's register within the subgroup (the leader broadcasts, addressed or uniform)
  - a coordinate of the launch configuration (the position-aware offset), which is not a data read at all
- operations addressed to different slices do not interfere: the single-lane write (subgroup `1`) and the leader broadcast (subgroup `2`) yield the same buffer in either order
- the initial shape of the buffer is kept intact.

The chapter formalises each feature in turn: the any-order issuance in Section 1.2, the fixed placement in Section 1.3, the write confinement in Section 1.3.2, and the typed discipline of the subgroup operations with shape preservation in Section 1.4.

1.1.2 Total structured recursion

Recall that a *recursion scheme* (Section **??**) is a combinator that captures a pattern of recursion as a single higher-order function, parameterised by the step that varies between instances (Meijer et al. 1991). For an endofunctor F describing the one-layer shape of a datatype, *catamorphism* folds a structure to a value by an algebra $F\,A \to A$. *Anamorphism* unfolds a seed into a structure by a *coalgebra* $A \to F\,A$. *Hylomorphism* is the composite of the two by unfolding and folding without materialising the intermediate structure. These are staples in *algebra of programming* (Bird and Moor 1997).

The form this chapter uses is catamorphism with a paired carrier. We instantiate it such that the fold has a tupled carrier `(rebuilt original × result)`. Here, an operation is the second projection. The step it sees is the recursively computed result and the *original substructure* that produced it. This has a type $F\,(\mu F \times A) \to A$ rather than $F\,A \to A$. Where a plain catamorphism has forgotten the input by the time its algebra runs, the paired fold retains it, available to be carried through untouched. An operation can thus edit one slice while the surrounding structure passes by verbatim. This pairing is the *tupling trick* that

implements **paramorphism** (Meertens 1992). This is also called the primitive-recursive strengthening of the fold.

In our construction, only the type property of paramorphism is exercised. This is because in operations that model subgroups, the addressing step consults only the carried original substructure. The recursively computed result is ignored, but the type signature is used as a lens complement. In this approach, the complement is the paired carrier's reconstruction identity. The paramorphic type nevertheless carries weight, because it bridges the scheme with the lens laws. This retains expressivity to distinguish patterns where each step consults structure only (content-independent) or both structure and result (content-dependent). Addressing of a subgroup operation is always content-independent. Later sections show that subgroup *instructions* are catamorphisms or cheap zygomorphisms, but these are distinct patterns from how the *addressing* happens.

Two encodings of the carrier μF must be distinguished, because only one is admissible in a proof assistant. Existing works often use a literal fixpoint with constructor in : $F\,(\mu F) \to \mu F$. Here, the schemes are written as self-reference, but in a total type theory this encoding is inadmissible. The constructor is non-positive and the recursion is unbounded, so the definitions type-check but may not appear in a theorem. The chapter therefore works throughout with Lambek isomorphism as the safe alternative. Lambek's lemma states that the carrier of an initial algebra is isomorphic to one layer of itself, $\mu F \cong F\,(\mu F)$. A Lambek isomorphism for F is a carrier C with a fold/unfold pair, fold : $F\,C \to C$ and unfold : $C \to F\,C$, whose two composites are the identity. The development calls this structure `FIso`:

Listing 1.1 The Lambek iso: a carrier with mutually inverse fold and unfold.

```
structure FIso (F : Type → Type) [Functor F] where
  carrier : Type
  fold    : F carrier → carrier
  unfold  : carrier → F carrier
  left_inv  : ∀ x, fold (unfold x) = x
  right_inv : ∀ x, unfold (fold x) = x
```

In the development, `FIso` extends the one-layer algebra and coalgebra structures `FAlgebra` and `FCoalgebra`, so its two halves can be named as `toFAlgebra` and `toFCoalgebra`; the flattened form above lists the same fields.

`MonadicF` realises lists as the Lambek iso of the functor $F\,X = 1 + E \times X$. Write `Monadic E X` for that one-layer functor `Option (E × X)`. Then `fold` is cons and `unfold` is uncons:

Listing 1.2 Lists as the Lambek iso of the list functor.

```
def MonadicF : FIso (Monadic E) where
  carrier := List E
  fold   := fun | none ⇒ [] | some (x, xs) ⇒ x :: xs
  unfold := fun | [] ⇒ none | x :: xs ⇒ some (x, xs)
  left_inv  := by intro x; cases x <;> rfl
  right_inv := by intro x; cases x with | none ⇒ rfl | some p ⇒ cases p; rfl
```

Termination is supplied separately by **recursive coalgebras** (Hinze et al. 2015). A coalgebra is a carrier with an `unfold : C → F C`. It is *recursive* when the hylomorphism equation $h = a \circ F\,h \circ \text{unfold}$ has a unique solution for every algebra $a : F\,A \to A$, which guarantees the recursion converges without restricting the ambient category. The development records this as the typeclass `IsRecursive`, whose `recHylo` field is that unique solution, $[\![a]\!]_{\text{unfold}}$:

Listing 1.3 Recursive coalgebras: the unique hylomorphism solution as a typeclass.

```
-- c is the unfold side: a carrier with unfold : carrier → F carrier
class IsRecursive (c : FCoalgebra F) where
  recHylo        : (F A → A) → c.carrier → A
  -- recHylo a solves the equation
  recHylo_eq     : recHylo a x = a (Functor.map (recHylo a) (c.unfold x))
  -- and is the only solution
  recHylo_unique : (∀ x, h x = a (Functor.map h (c.unfold x))) → h = recHylo a
```

The recursion schemes used downstream are built from these two ingredients: the Lambek iso supplies the layers, and the recursive coalgebra the termination. This means that each such instance carries a termination guarantee and admits inductive proof.

1.1.3 Lenses and Under

This subsection fixes two prior-work vocabularies: the lens calculus with its constant complement, and the Under combinator read as conjugation. The lens calculus of Foster et al. (2005) gives the bidirectional vocabulary.

Definition 1.1 (Lens)**.** A *lens* from a source type S to a view type V is a getter together with a setter, where put writes a new view back into a source and reassembles the rest:

$$\text{get} : S \to V$$

$$\text{put} : V \times S \to S$$

Definition 1.2 (Well-behaved lens)**.** A lens is *well-behaved* when it satisfies two round-trip laws:

$$\text{put}(\text{get}\ s,\ s) = s \quad (\mathit{GetPut})$$

$$\text{get}(\text{put}(v, s)) = v \quad (\mathit{PutGet})$$

Definition 1.3 (Lawful lens)**.** A well-behaved lens is called *lawful* or "very well-behaved" when it additionally satisfies:

$$\text{put}(v',\ \text{put}(v, s)) = \text{put}(v',\ s) \quad (\mathit{PutPut})$$

GetPut and PutGet make getter and setter mutually coherent. PutPut adds statelessness where a later write supersedes an earlier one. The semantic content of a lawful lens is the **constant-complement** condition of Bancilhon and Spyratos (1981). A lawful lens exhibits a decomposition

$$S \ \cong\ V \times C$$

in which the complement C is everything in the source not in the view, get is the projection onto V, and put replaces the V factor while holding C fixed. The complement is usually left existential:

$$\exists C.\ S \cong V \times C$$

Making it concrete, and choosing the *minimal* such C, is the answer Foster et al. (2005) give to the view-update problem. The whole chapter reads C as the memory a focused operation must leave intact, so that "recording the complement" and "naming the preserved memory" become one act (Theorem 1.11).

When it comes to Under, the data of a do-transform-undo operation is what we call an obverse. Obverse can be seen as lawless Undo: a pair of functions run forwards and backwards.

Definition 1.4 (Obverse)**.** An obverse is a forward/backward pair on two types with no law relating them:

$$\text{fwd} : A \to B$$

$$\text{bwd} : B \to A$$

Undo is used in BQN to define Under. Under is written with a donut ◎ between its two functions: F ◎ G.

Definition 1.5 (Monadic Under)**.** Transform the argument x by g, act with f, then undo g:

$$f \circledcirc g\ x = g^{-1}(f(g\ x))$$

Once the literal inverse g^{-1} is relaxed to the backward map of an obverse, this is the conjugation of an endomorphism f by the pair:

$$\text{bwd} \circ f \circ \text{fwd}$$

Conjugation is the profunctor action dimap on the function arrow.

Definition 1.6 (dimap)**.** For $p : A' \to A$ and $q : B \to B'$:

$$\text{dimap}\ p\ q\ f = q \circ f \circ p$$

Under is the diagonal case of a dimap.

Definition 1.7 (Under as dimap)**.** Given an obverse g, and some transformation f, then Under is a dimap of the form:

$$f \circledcirc g = \text{dimap } g.\text{fwd } g.\text{bwd } f = g.\text{bwd} \circ f \circ g.\text{fwd}$$

So, Under adds nothing categorical beyond the profunctor action it instantiates. What gives it content is *which* obverse g is, and which round-trip laws g obeys.

1.2 A typed taxonomy of Under

This section implements Undo in three tiers. These are classified by the round-trip laws each tier obeys. We then extend this to cover the Under combinator with intrinsic typing. Structural Under arrives later in the Section 1.4.1 via extrinsic typing.

Under classically comes in two forms called mathematical and structural (Kromberg 2023). Using the concept of lenses, we draw a further distinction between bare and lawful lenses.

The lawfulness of Under is determined by the existence of a right inverse. This is closely related to a named complement type. Naming the complement turns Under into a total lawful lens. In the complement type view, Under becomes a type polymorphic construct capable of expressing the frame and par-rule from concurrent separation logic.

The contribution of this section for the array programming community is, to our knowledge, the first dependently typed definition of Under. A more general contribution is the bits of correspondence between complement types and concurrent separation logic.

1.2.1 Undo: Obverse, Section, and Isomorphism

We model Undo as a class definition (Listing 1.4) for invertible operations.

In a typed setting, we get three meaningful structures out of this class. First is the Obverse (Definition 1.4) as a Lean 4 structure (Listing 1.5).

For example, `Obverse.mul` can be defined as an `Obverse` over two natural numbers.

Listing 1.4 The `Undo` class[2]: the algebra of inverse operations with `fwd` and `bwd` projections and extensionality.

```
class Undo (P: Type → Type → Type) where
  id: P A A                          -- id⁼ = id
  symm: P A B → P B A                -- f⁼⁼ = f
  comp: P B C → P A B → P A C        -- (f∘g)⁼ = g⁼∘f⁼
  each: P A B → P (List A) (List B)  -- f¨⁼ = f⁼¨
  fwd: P A B → (A → B)
  bwd: P A B → (B → A)
  ext: fwd f = fwd g → bwd f = bwd g → f = g
```

Listing 1.5 An `Obverse`: a lawless forward/backward pair.

```
structure Obverse (A B: Type) where
  fwd: A → B
  bwd: B → A
```

```
def Obverse.mul (k: Nat): Obverse Nat Nat where
  fwd := (· * k)
  bwd := (· / k)
```

A `Section` (Listing 1.6) adds a left inverse bwd ∘ fwd = id. Thus fwd is injective with a chosen retraction. This is a selection that remembers where its items came from.

Listing 1.6 A `Section`: an `Obverse` with a left inverse.

```
structure Section (A B: Type) extends Obverse A B where
  left_inv: ∀ x, bwd (fwd x) = x
```

As an example, `Obverse.mul` becomes a `Section.mul` when `k > 0`. This is because division left-inverts multiplication.

```
def Section.mul (k: Nat) (hk: k > 0): Section Nat Nat where
  toObverse := Obverse.mul k
  left_inv x := Nat.mul_div_cancel x hk
```

An `Iso` (Listing 1.7) adds a right inverse fwd ∘ bwd = id. This makes fwd a bijection with inverse bwd.

For example, integer addition is an instance of `Iso`.

Listing 1.7 An `Iso`: a `Section` that is also right-invertible.

```
structure Iso (A B: Type) extends Section A B where
  right_inv: ∀ x, fwd (bwd x) = x
```

```
def Iso.add (k: Int): Iso Int Int where
  fwd := (· + k)
  bwd := (· - k)
  left_inv  _ := by omega
  right_inv _ := by omega
```

Integer subtraction can then be defined as the symmetry of `Iso.add`. This realises the `symm` law of Listing 1.4 at the `Iso` tier.

```
def Iso.sub (k: Int): Iso Int Int := (add k).symm
```

Basic arithmetic can be exemplified here. 7 undone with +3 is 4.

```
example: (Iso.add 3).symm 7 = (Iso.sub 3) 7 := by rfl
```

Recall that Under is defined via Undo (Definition 1.5). We can implement the dimap view of Under (Definition 1.7) as follows:

Listing 1.8 `Undo.under`: conjugate `f` by any `Undo`, `bwd ∘ f ∘ fwd`.

```
def Undo.under [Undo P] (p: P A B) (f: B → B) (x: A): A :=
  Undo.bwd p (f (Undo.fwd p x))
```

The three Undo tiers correspond to the two flavours (Kromberg 2023) of Under. A mathematical Under has a transform that is an invertible value function. It conjugates by an `Iso`. This is conjugation in a transformed domain, $\text{fwd}^{-1} \circ f \circ \text{fwd}$.

A structural Under is either a `Section` or `Iso`. In both cases, the `fwd` and `bwd` form a selection. For example, rearrangements that lose nothing such as reverse or transpose are `Iso`s. Selections that discard part of their input such as take, drop, or filter are `Section`s.

In a `Section`, fwd has a left inverse that scatters the kept items back. However, the converse law fwd ∘ bwd = id fails on inputs the selection could never have produced. There the operation is unconstrained.

The lawless `Obverse` is the common root.

BQN's specification resolves Under by a three-way dispatch. Its invertible case occurs when the view transform is uniquely invertible on the transformed value. This is the `Iso` tier. Its structural case is the selection whose inverse scatters the kept items back. This is the `Section` tier. Its computational case is the lawless `Obverse`. This covers imprecise cases such as mean under square and a strong profunctor variant [3].

Shape preservation is a distinct property of structural Under. In this system, it is a property of the instruction rather than the combinator itself. The later sections supply that discipline by type and by theorem (Section 1.3.2, Section 1.4.1).

1.2.2 Lens view of Under

To make the complement explicit in the type for Under, we turn to lens laws. A lens (Listing 1.9) is a slight upgrade to Undo, where `put` is binary.

Listing 1.9 Typed `Lens`.

```
structure Lens (S V: Type) where
  get: S → V
  put: V × S → S
```

Any Undo structure fulfills `put` by discarding, i.e., being constant, in the second argument (Listing 1.10).

[3] Recall that the transform is an endomorphism `f : B → B`. Its functorial ("strong" profunctor) variant `underF p f x = Functor.map p.bwd (f (p.fwd x))` takes `f : B → G B` for a functor `G` and lifts `bwd` through that shape. This is the van Laarhoven modify `(B → G B) → (A → G A)` (a lens for `G` a functor, a traversal for `G` applicative), and BQN's element-wise Under when the transform changes shape (`bwd` mapped through `G`, cf. the `f˜˜ = f⁼˜` law of Listing 1.4). It recovers `under` at `G = Id`.

Listing 1.10 `Obverse.toLens` is a constant-put lens.

```
def Obverse.toLens (p: Obverse A B): Lens A B where
  get := p.fwd
  put := λ(b, _) ⇒ p.bwd b
```

Various interpretations of lenses exist. A particularly relevant formulation is via the so-called costate comonad (Listing 1.11). This formulation allows us to represent the lenses in terms of F-coalgebras.

Listing 1.11 The `Store` functor of the costate comonad.

```
def Store (V X: Type): Type := V × (V → X)

instance: Functor (Store V) where
  map f := fun (v, k) ⇒ (v, f ∘ k)
```

With the above instantiation, lenses correspond bijectively to coalgebras of the `Store` comonad (Listing 1.12).

Listing 1.12 `Lens` defined as the costate comonad.

```
def Lens.equivFCoalg: Lens S V ≃ { c: FCoalgebra (Store V) // c.carrier = S } where
  toFun l := ⟨{
    carrier := S
    unfold := fun s ⇒ (l.get s, fun v ⇒ l.put (v, s))
  }, rfl⟩
  invFun := fun ⟨⟨_, unfold⟩, h⟩ ⇒ by
    subst h
    exact {
      get := fun s ⇒ (unfold s).1
      put := fun (v, s) ⇒ (unfold s).2 v
    }
  left_inv l := rfl
  right_inv := fun ⟨⟨_, unfold⟩, h⟩ ⇒ by
    subst h
    rfl
```

Given our `FCoalgebra` with an explicit carrier, we can move a lens to an `FCoalgebra` indexed by the `Store` comonad (Listing 1.13).

Now, given some function `f` over the view, the Under structure unfolds the getter and setter (Listing 1.14). These are in principle the forward and backward transformations.

Listing 1.13 Transporting `Lens` to an F-coalgebra.

```
def Lens.toFCoalg (l: Lens S V): FCoalgebra (Store V) where
  carrier := S
  unfold := λs ⇒ (l.get s, λv ⇒ l.put (v, s))
```

Listing 1.14 `Lens.over`.

```
def Lens.over (l: Lens S V) (f: V → V) (s: S): S :=
  let (fwd, bwd) := l.toFCoalg.unfold s
  bwd (f fwd)
```

With this approach, the lens `over` is Under repackaged (Theorem 1.8).

Theorem 1.8 (Under is the lens over)**.** *For an obverse p and a view transform f, running f under p is the* over *of p's constant-put lens:*

$$\text{over}\,(\text{toLens}\ p)\ f\ =\ \text{under}\ p\ f$$

Proof. Both sides unfold to the same conjugation $\text{bwd} \circ f \circ \text{fwd}$.

The constant-put lens has getter fwd and setter $(v, \_) \mapsto \text{bwd}\ v$. Thus over reads the fwd-view, applies f, and reassembles with bwd, discarding the source. The equality is definitional. □

Being one operation, `over` and `under` raise a single question about composition. This splits in two. Composing the conjugating pairs is free.

Theorem 1.9 (Under composes along the pair)**.** *For obverses p and q and a transform f, conjugating by the composite obverse equals nesting the conjugations:*

$$\text{under}\,(\text{comp}\ q\ p)\ f\ =\ \text{under}\ p\,(\text{under}\ q\ f)$$

Proof. Both sides unfold to $\text{bwd}_p(\text{bwd}_q(f(\text{fwd}_q(\text{fwd}_p\ x))))$. The composite's forward map is $\text{fwd}_q \circ \text{fwd}_p$. Its backward map is $\text{bwd}_p \circ \text{bwd}_q$. The equality is definitional. □

Composing two transforms on one lens is not free. Fusing two updates with different transforms into a single write is $\text{over}\ g \circ \text{over}\ f = \text{over}\,(g \circ f)$. At a

general lens the fusion spends PutGet and PutPut (Definition 1.3) together; at a constant-put pair PutPut is free, so PutGet alone carries it (Section 1.3.1). A bare lens lacks these laws. The lens on `Nat × Nat` with view `Nat` whose `put` special-cases a view already equal to the first component is a witness (Listing 1.15).

Listing 1.15 `over` need not compose without lens laws.

```
def counterexample: Lens (Nat × Nat) Nat where
  get := Prod.fst
  put := fun (v, s) ⇒ if v == s.1 then s else (v, v)

-- write 3 then 1, against the single fused write (1 ∘ 3)
example: (counterexample.over (fun _ ⇒ 1) ∘
          counterexample.over (fun _ ⇒ 3)) (1, 2) = (1, 1) := by decide
example:  counterexample.over ((fun _ ⇒ 1) ∘ (fun _ ⇒ 3)) (1, 2) = (1, 2) := by decide
```

Writing 3 then 1 lands on `(1, 1)`. The single fused write lands on `(1, 2)`. The two disagree. Thus PutPut fails. Recovering it is what the lawfulness ladder of the next subsections supplies.

1.2.2.1 Structural section

Recovering PutPut begins by adding the first two lens laws. A well-behaved lens (Definition 1.2) adjoins GetPut and PutGet to a bare one (Listing 1.16).

Listing 1.16 A `WellBehavedLens`.

```
structure WellBehavedLens (S V: Type) extends Lens S V where
  getput: ∀ s, put ((get s), s) = s
  putget: ∀ v s, get (put (v, s)) = v
```

These give over id = id and get(over f s) = f(get s). The section tier does not inhabit this rung. Its unconditional home is the bare lens (Listing 1.18). Here the left inverse discharges GetPut outright as bwd(fwd x) = x. No further law follows.

On the image of fwd the section is well-behaved. fwd(bwd(fwd x)) = fwd x holds there. However, writing back a view the source could never have produced is left unconstrained.

A structural section (Listing 1.17) carries its complement explicitly.

Listing 1.17 A `StructuralSection`: a section carrying its complement type.

```
structure StructuralSection (S V: Type): Type 1 where
  Complement: Type
  decomp: S → V × Complement
  recomp: V × Complement → S
  left_inv: ∀ s, recomp (decomp s) = s
```

This is the intrinsic type of BQN's structural Under selector plus complement preservation, in the constant-complement form of Bancilhon and Spyratos (1981).

It is a `Section` on the paired view. It projects to a bare `Lens` whose `put` reads both the new view and the retained complement. This is unlike the constant-put lens of an obverse (Listing 1.10).

Listing 1.18 `StructuralSection` as a `Section`, a `Lens`, and its `under`.

```
def StructuralSection.toSection (ss: StructuralSection S V): Section S (V × ss.Complement) where
  fwd := ss.decomp
  bwd := ss.recomp
  left_inv := ss.left_inv

def StructuralSection.toLens (ss: StructuralSection S V): Lens S V where
  get s := (ss.decomp s).1
  put := λ(v, s) ⇒ ss.recomp (v, (ss.decomp s).2)

def StructuralSection.under (ss: StructuralSection S V) (f: V → V) (s: S): S :=
  ss.toLens.over f s
```

The complement is still reused in the rebuild. `over` re-embeds (decomp s).2.

However, a section need not let it be read back. The converse round trip decomp ∘ recomp = id may fail off the image of `decomp`.

Chunking a buffer that need not divide evenly is one such instance. Concretely, that instance is `chunk` (Listing 1.19). `decomp` splits a list into rows of `n`. `recomp` flattens them. The complement is trivial (`Unit`), and nothing is discarded. What fails is the converse direction: ragged row-lists inhabit the paired type without being any list's decomposition, so re-chunking their flattening is not the identity.

At a `Vector`-typed carrier the shape moves into the type, and the same chunking becomes an isomorphism, which is the form Section 1.4 issues through.

Listing 1.19 The chunking structural section.

```
def StructuralSection.chunk (n: Nat): StructuralSection (List E) (List (List E)) where
  Complement := Unit
  decomp xs := (listChunk n xs, ())
  recomp := fun (rows, _) ⇒ rows.flatten
  left_inv xs := flatten_listChunk n xs

example: (StructuralSection.chunk 3).under (List.map List.reverse)
  [1,2,3,4,5,6,7,8,9] = [3,2,1, 6,5,4, 9,8,7] := by native_decide

def addPositional (offset: Nat) (xs: List Nat): List Nat :=
  xs.zipWith (fun x i ⇒ x + offset + i) (List.range xs.length)

example: (StructuralSection.chunk 4).under (List.map (addPositional 0))
  [1,2,3,4, 5,6,7,8, 9,10,11,12, 13,14,15,16]
  = [1,3,5,7, 5,7,9,11, 9,11,13,15, 13,15,17,19] := by native_decide
```

Its round-trip law via `left_inv` is a theorem. The proof term is assembled from F-algebra constructions. This is similar to how BQN has pre-set inverses for each structural operation.

Theorem 1.10 (Chunking round-trip)**.** *Flattening the chunks recovers the list: for every n and xs:*

$$\text{flatten}\,(\text{listChunk}\ n\ xs) = xs$$

Proof. listChunk is a hylomorphism. It unfolds the safe-chunking coalgebra. It refolds through the initial algebra on List (List E). Thus its recursion is well-founded even when n does not divide the length.

flatten is the unique algebra homomorphism from that initial algebra to the list-monoid algebra on List E. Naturality of the hylomorphism in its algebra carries flatten through the fold. The hylomorphism into the list-monoid algebra is the identity. Their composite is flatten $\circ$ listChunk $=$ id. □

`filterSection` (Listing 1.20) shows a case where the complement is redundant.

In these cases, the complement records what the filter forgets. Here it is the whole original list. In its methods, `decomp` pairs the filtered view with the original. `recomp` scatters the possibly modified selection back into the positions where `pred` held, filling the rest from the original. Both directions are folds: `filterAlg` keeps the passing elements, and `interleaveAlg` merges the selection with the rejected ones. The round trip recomp (decomp x) $= x$ holds on real inputs.

This is a section, not an isomorphism. The redundancy is what costs the converse law. A pair (sel, orig) whose `orig` is unrelated to `sel` inhabits $V \times C$. It is not in the image of `decomp`. Thus writing it back is unconstrained. PutGet fails off-image. This matches the section's on-image lawfulness.

Listing 1.20 `filterSection`: filter with the original list as complement.

```
def filterSection (pred: A → Bool): StructuralSection (List A) (List A) where
  Complement := List A
  decomp x := (listFold (filterAlg pred).fold x, x)
  recomp := λ(sel, orig) ⇒
    listFold (interleaveAlg pred).fold orig
      (sel, listFold (filterAlg (λx ⇒ !pred x)).fold orig)
  left_inv x := interleave_partition pred x
```

In other words, its Under method requires a masked map. This transforms only the entries passing the predicate. For example, this can double even entries and leave odd ones untouched (Listing 1.21).

Listing 1.21 A masked map via `filterSection`: double the evens, leave the odds.

```
example: (filterSection (· % 2 == 0)).under (List.map (· * 2))
  [3, 8, 1, 6, 5, 2, 7, 4] = [3, 16, 1, 12, 5, 4, 7, 8] := by decide
```

Whether `filterSection` promotes to a structural isomorphism is about its complement.

Here, its section-ness comes from a redundant complement. Keeping the whole original makes the view recoverable from it. Thus the selected elements are stored twice. That slack is the junk that costs PutGet. The redundancy is removable. The view already fixes the selected elements. All it leaves free is which positions were selected and what the unselected elements are.

Recording only the mask `x.map pred` and the elements that failed `pred` gives the minimal complement. Against it the view is independent. The junk states vanish. Once the view and complement types forbid mismatched masks, `decomp ∘ recomp = id` holds too. Thus filter becomes an isomorphism.

1.2.2.2 Structural isomorphism

Adjoining the converse law decomp ∘ recomp = id promotes a structural section to a structural isomorphism (Listing 1.22) $S \simeq V \times C$ (Listing 1.24). The complement can also be read back. Thus `decomp` and `recomp` become mutually inverse.

Listing 1.22 A `StructuralIso`: a `StructuralSection` whose decomposition is a bijection.

```
structure StructuralIso (S V: Type): Type 1 extends StructuralSection S V where
  right_inv: ∀ vc, decomp (recomp vc) = vc
```

Where a section's lens is the unlawful version and well-behaved only on its image, the isomorphism is unconditionally lawful. I.e., a lawful lens (Theorem 1.3, Listing 1.23) adds PutPut to a well-behaved one. This is the law that makes writes compose.

Listing 1.23 A `LawfulLens`.

```
structure LawfulLens (S V: Type) extends WellBehavedLens S V where
  putput: ∀ v v' s, put (v', (put (v, s))) = put (v', s)
```

The `under` of a structural isomorphism is the one inherited from its section (Listing 1.18). `StructuralIso.toLawfulLens` (Listing 1.24) threads the complement through `put`.

The `FIso` case (Listing 1.25) is the fold view. The Lambek isomorphism makes Under total, composing (Listing 1.23), and complement-preserving (Theorem 1.11). Totality is a divergence from BQN worth naming. Where BQN's structural Under raises an error when the written view does not fit the source, the total formulation silently computes with whatever view it is handed. The fit obligation returns as the shape certificate of Section 1.4.1.

Listing 1.24 `StructuralIso` as an equivalence and a `LawfulLens`.

```
def StructuralIso.toEquiv (si: StructuralIso S V): S ≃ V × si.Complement where
  toFun := si.decomp
  invFun := si.recomp
  left_inv := si.left_inv
  right_inv := si.right_inv

def StructuralIso.toLawfulLens (si: StructuralIso S V): LawfulLens S V where
  toLens := si.toStructuralSection.toLens
  getput := λ s ⇒ si.left_inv s
  putget := λ v s ⇒ congrArg Prod.fst (si.right_inv (v, (si.decomp s).2))
  putput := λ v v' s ⇒ by
    show si.recomp (v', (si.decomp (si.recomp (v, (si.decomp s).2))).2)
       = si.recomp (v', (si.decomp s).2)
    rw [si.right_inv]
```

Listing 1.25 A bijection is a constant-put lawful lens.

```
def Iso.toLawfulLens (iso : Iso A B) : LawfulLens A B where
  get := iso.fwd
  put := fun (b, _) ⇒ iso.bwd b
  getput := fun a ⇒ iso.left_inv a
  putget := fun b _ ⇒ iso.right_inv b
  putput := fun _ b' _ ⇒ by show iso.bwd b' = iso.bwd b'; rfl

def FIso.toLawfulLens [Functor F] (iso: FIso F): LawfulLens iso.carrier (F iso.carrier) where
  get := iso.unfold
  put := λ (x, _) ⇒ iso.fold x
  getput := λ s ⇒ iso.left_inv s
  putget := λ v _ ⇒ iso.right_inv v
  putput := λ _ v' _ ⇒ by show iso.fold v' = iso.fold v'; rfl
```

Theorem 1.11 (Complement preservation)**.** *For a structural* isomorphism *and a view operation f, decomposing the result of an update splits as the updated view and the* original *complement:*

$$\mathsf{si.decomp}\,(\mathsf{si.under}\ f\ s)\ =\ (f\,(\mathsf{si.decomp}\ s).1,\ (\mathsf{si.decomp}\ s).2)$$

so in particular the complement read back from the result equals the original:

$$(\mathsf{si.decomp}\,(\mathsf{si.under}\ f\ s)).2\ =\ (\mathsf{si.decomp}\ s).2$$

Proof. `under` rebuilds from the original complement. $\mathsf{si.under}\ f\ s\ =\ \mathsf{recomp}\,(f\,(\mathsf{decomp}\ s).1,\ (\mathsf{decomp}\ s).2)$. Decomposing again is one rewrite by

the isomorphism's converse law $\texttt{decomp} \circ \texttt{recomp} = \texttt{id}$. The second equation is the first projected onto the complement. □

The out-of-scope component is therefore invariant under the operation. This holds for all `f`. Preservation is observational at the iso tier. That observational invariance has a separation-logic reading. This is the frame rule (Theorem 1.12).

Read a decomposition decomp $s = (\text{view}, \text{frame})$ as a constant-complement separating conjunction $s \vDash \text{view} * \text{frame}$. The view is the footprint an operation may touch. The frame becomes a polymorphic complement type.

Theorem 1.12 (Frame rule)**.** *Let* si *be a structural isomorphism, and write* view $s = (\texttt{si.decomp}\ s).1$ *and* frame $s = (\texttt{si.decomp}\ s).2$ *for the components of its decomposition. For a view operation f with a local specification $\forall v,\ P\,v \to Q\,(f\,v)$, any predicate R on the complement, and any source s: from $P\,(\text{view}\ s)$ and $R\,(\text{frame}\ s)$:*

$$Q\,(\text{view}\,(\texttt{si.under}\ f\ s)) \ \wedge\ R\,(\text{frame}\,(\texttt{si.under}\ f\ s))$$

Proof. By complement preservation, $\texttt{si.under}\ f\ s$ decomposes as $(f\,(\text{view}\ s),\ \text{frame}\ s)$. The first conjunct is $Q\,(f\,(\text{view}\ s))$. The local specification supplies this from $P\,(\text{view}\ s)$. The second is $R\,(\text{frame}\ s)$. This is the frame hypothesis unchanged. No premise relates f to R. The side condition $\text{mod}(f) \cap \text{fv}(R) = \emptyset$ is discharged structurally. This holds since $\texttt{si.under}\ f$ provably never writes the frame. □

This is the denotational, constant-complement lens form of the frame rule (Reynolds 2002). It is an instance of its semantic frame property (Yang and O'Hearn 2002). Because $\texttt{si.under}\ f$ is a total function over the product $V \times C$, the general soundness theorem's safety- and termination-monotonicity side conditions are vacuous. A fault-free or terminating run stays so as the state grows.

The frame rule becomes more useful once two selections are independent (Listing 1.26).

Listing 1.26 Independence of two selections: one joint constant-complement decomposition exhibits both views.

```
structure Independent (sel₁: StructuralSection S V₁) (sel₂: StructuralSection S V₂) (C: Type) where
  joint  : S ≃ (V₁ × V₂) × C
  view₁  : ∀ s, (sel₁.decomp s).1 = (joint s).1.1
  view₂  : ∀ s, (sel₂.decomp s).1 = (joint s).1.2
  under₁ : ∀ (f: V₁ → V₁) (s: S),
    sel₁.under f s = joint.symm ((f (joint s).1.1, (joint s).1.2), (joint s).2)
  under₂ : ∀ (g: V₂ → V₂) (s: S),
    sel₂.under g s = joint.symm (((joint s).1.1, g (joint s).1.2), (joint s).2)
```

Independence is thus a property of the two selections' complements. This holds over arbitrary operations. Commutation therefore holds for every pair of instructions at once (Theorem 1.13).

Theorem 1.13 (Independent selections commute)**.** *For independent selections* $\mathrm{sel}_1, \mathrm{sel}_2$ *and any view operations* f *and* g*,*

$$\mathrm{sel}_1.\mathrm{under}\ f \circ \mathrm{sel}_2.\mathrm{under}\ g = \mathrm{sel}_2.\mathrm{under}\ g \circ \mathrm{sel}_1.\mathrm{under}\ f.$$

Proof. On the joint decomposition (v_1, v_2, c) of the source, the independence equations rewrite both orders. They rewrite to the same product update $\mathrm{joint}^{-1}((f\,v_1,\ g\,v_2),\ c)$. □

This machinery allows the frame rule to be lifted. This applies to parallel composition over independent complements (Theorem 1.14).

Theorem 1.14 (Parallel rule)**.** *Let* sel_1 *and* sel_2 *be independent with joint complement* C*, and write* $\mathrm{view}_i\ s = (\mathrm{sel}_i.\mathrm{decomp}\ s).1$ *and* $\mathrm{frame}\ s = (\mathrm{joint}\ s).2$*. For view operations* f, g *with local specifications* $\forall v,\ P_1\,v \to Q_1\,(f\,v)$ *and* $\forall v,\ P_2\,v \to Q_2\,(g\,v)$*, any predicate* R *on* C*, and any source* s*: from* $P_1\,(\mathrm{view}_1\ s)$*,* $P_2\,(\mathrm{view}_2\ s)$*, and* $R\,(\mathrm{frame}\ s)$*, the parallel composite* $p = \mathrm{sel}_1.\mathrm{under}\ f \circ \mathrm{sel}_2.\mathrm{under}\ g$ *satisfies:*

$$Q_1\,(\mathrm{view}_1\,(p\ s)) \ \wedge\ Q_2\,(\mathrm{view}_2\,(p\ s)) \ \wedge\ R\,(\mathrm{frame}\,(p\ s))$$

Proof. By the independence equations the composite decomposes as $((f\,(\text{view}_1\ s),\ g\,(\text{view}_2\ s)),\ \text{frame}\ s)$. The first two conjuncts are the local specifications applied to the views. The third is the frame hypothesis unchanged. No premise relates f to P_2 or Q_2, or g to P_1 or Q_1, or either operation to R. The concurrency rule's side conditions are discharged structurally. This is as in Theorem 1.12. □

This is the parallel composition rule of concurrent separation logic (O'Hearn 2007; Brookes 2007) in constant-complement form. Again, it is a denotational equation rather than an operational Hoare triple. Thus it speaks only to order-independence, i.e., it models neither interleaving nor ownership transfer. Despite this, the theorem is useful; it can be used to *derive* barriers from composition of structural Unders (Section 1.4).

This expressivity comes from naming the complement. The write fusion this needs, over $g \circ \text{over}\ f = \text{over}\,(g \circ f)$, is available only at the lawful tier, where PutGet and PutPut hold together. The named (C explicit) and existential presentations of the complement are reconciled by a quotient.

For a lawful lens, relate two sources when every write to them agrees:

$$s_1 \sim s_2 \ :\Longleftrightarrow\ \forall v,\ \text{put}(v, s_1) = \text{put}(v, s_2)$$

Take Foster's complement to be the quotient $S/\!\sim$. This is the canonical residual the lens laws already determine. Promoting the implicit complement to this explicit one yields a structural isomorphism. Projecting that isomorphism back to a lawful lens recovers the original.

The reverse round trip takes a structural isomorphism with a hand-chosen complement C. It passes it to its Foster complement and back. This round trip is the identity only up to isomorphism.

$$C \ \simeq\ S/\!\sim \qquad (V \text{ inhabited})$$

I.e., C may carry ornamentation that the lens laws cannot observe, e.g., an index or a vector length. Hence, with $S/\sim$ as the normal form:

$$\text{LawfulLens } S\ V \ \simeq\ \text{StructuralIso } S\ V \qquad \text{modulo complement isomorphism}$$

Both round trips are mechanised: the lens-to-isomorphism direction recovers the original lens definitionally, and the reverse recovers the hand-chosen complement up to the isomorphism above.

In practice, this means that a construction may be stated in the array-language vocabulary of Under whilst reasoned in the vocabulary of lenses. The type (obverse, section, or iso) records exactly which laws are in force. This includes whether its complement is named.

1.3 Iterated Under as a single fold

Under iterates in three ways. This section develops one interface per row: the lens action, a bare conjugation, and an algebra fold:

Table 1.1: Three interfaces of iterated Under.

Construct	Interface to the pair	Laws consulted	Terminates on
`mapHead`/`mapTail` (Section 1.3.1)	the lens: `over` at the list pair	all three lens laws, free at a bijection	one layer, nothing recurses
`applyAt` (Section 1.3.2)	conjugation: `dimap` of a bare `(fold, unfold)`	none to compute; the round trips upgrade each step to a lawful `over`	the position `n`
`pairedFold` (Section 1.3.3)	the algebra: `fold` paired with a step, run by the recursive coalgebra	recursivity, the round trips, functor laws	the structure

1.3.1 Lens at the list pair

Iterating lenses to a fixed position means descending the tails and acting at the bottom. The list carrier's Lambek pair supplies both moves (Listing 1.27), via the lens bridge (Listing 1.25):

Listing 1.27 `mapHead` and `mapTail`: the iterated lens primitives of the list carrier.

```
def mapHead (f: E → E) (xs: List E): List E := MonadicF.toLawfulLens.over (bimap f id) xs
def mapTail (f: List E → List E) (xs: List E): List E := MonadicF.toLawfulLens.over (Functor.map f) xs

mapHead (· + 100) [1, 2, 3, 4] = [101, 2, 3, 4]   -- position 0, first element
mapTail (mapHead (· + 100)) [1, 2, 3, 4] = [1, 102, 3, 4]   -- position 1, second element
```

`bimap f g` maps the element and tail slots of one layer separately, so `bimap f id` acts on the element and passes the tail through. Composing the primitives descends the structure. `mapTail (mapHead f)` leaves the head and modifies the head of the tail, i.e., the second element. `mapTail (mapTail (mapHead f))` reaches the third element. In general, `n` tails deep followed by a head modifies the `(n+1)`-th element.

Leaving a layer unchanged is the round trip of the pair. GetPut makes the round trip the identity, so `mapTail id = id`. The tier table below calls this guarantee *pass-through*. Descending `n` layers untouched spends that law `n` times. The guarantees of the two definitions correspond to the tier of the pair they conjugate by (Section 1.2.1):

Table 1.2: The primitives at each Undo tier: conjugation enables computation at every tier; GetPut ensures pass-through; PutGet ensures write fusion.

Tier	computes	pass-through	write fusion
Obverse	✓	✗	✗
Section	✓	✓	✗
Iso	✓	✓	✓

The *computes* column of Table 1.2 is uniform. Conjugation never consults a law. Thus, `mapHead` and `mapTail` compute the same function at every tier.

The *pass-through* column concerns surviving a layer untouched. At a lawless pair, even the identity instruction corrupts the carrier. GetPut is exactly what prevents this.

The *write fusion* column concerns consecutive writes through the same view: `mapHead f ∘ mapHead g = mapHead (f ∘ g)`. This is PutGet: at the constant-put lens of a bijection PutPut holds for free, so the right inverse alone carries the fusion (Section 1.2.2).

The list pair in Listing 1.27 is a Lambek isomorphism. Thus, the primitives sit on the bottom row with every guarantee in force. The pass-through column is the guarantee that Theorem 1.16 spends once per layer.

1.3.2 The iterated conjugation

The second interface needs no lens, only the pair itself; the iterate is the fold of a Peano algebra over the position:

Listing 1.28 `applyAt`: the conjugation iterated by `natFold`.

```
def natFold (alg: Option R → R): Nat → R
  | 0     ⇒ alg none
  | n + 1 ⇒ alg (some (natFold alg n))

def optionAlg {A : Type} (default : A) : TAlgebra Option where
  carrier := A
  fold := fun | some x ⇒ x | none ⇒ default
  fold_pure := fun _ ⇒ rfl
  fold_join := by intro tt; cases tt <;> rfl

def peanoAlg (z: R) (s: R → R): FAlgebra Option where
  carrier := R
  fold := fun layer ⇒ (optionAlg z).fold (layer.map s)

def applyAt [Functor F] (C: Type) (fold: F C → C) (unfold: C → F C) (g: F C → F C) (n: Nat): C → C :=
  natFold (peanoAlg (dimap unfold fold g) (fun k ⇒ dimap unfold fold (Functor.map k))).fold n

example: applyAt (List Int) MonadicF.fold MonadicF.unfold
  (bimap (· + 100) id) 1 ([1, 2, 3, 4] : List Int)
  = [1, 102, 3, 4] := rfl
```

`applyAt` (Listing 1.28) is a form of iterated Under over a bare `(fold, unfold)` pair.

It applies layer transform `g` at position `n` of the recursive structure exposed by the pair. The definition is functor-generic, but the polymorphism the chapter spends is in the element type, which Section 1.4 instantiates at `E := Vector M w`. Every carrier exhibited is a linear spine, and that is what makes the count `n` a position; at a branching functor it would name a whole level. The zero case is the conjugated instruction. The successor case is the conjugated functorial action. Both cases are a `dimap` (Theorem 1.6), which is the form Under takes in general (Theorem 1.7).

Theorem 1.15 (Iterate bridge)**.** *For every element operation $op : E \to E$ and position n:*

$$\texttt{mapTail}^{[n]}\ (\texttt{mapHead}\ op) \;=\; \texttt{applyAt}\ (\texttt{bimap}\ op\ \texttt{id})\ n$$

Read `mapHead` and `mapTail` as the pair's conjugated instruction and conjugated functorial action; under that reading the equality holds at every carrier. Thus, `applyAt` iterates the functorial action on the instruction.

Proof. One induction establishes the tower form: the fold of a Peano algebra is an iterate, $\texttt{natFold}\ (\texttt{peanoAlg}\ z\ s)\ n = s^{[n]}\ z$; the zero case is definitional, and the successor case flips the iterate by one step and closes definitionally. Instantiating z as the conjugated instruction and s as the conjugated functorial action yields the equality at every carrier, and the displayed list form is its instance at the Lambek pair, where the two conjugations are `mapHead` and `mapTail`. □

Section 1.2.2.2 derived the frame rule of separation logic (Theorem 1.12) from the constant complement. The iterate has a counterpart, proved by its own induction:

Theorem 1.16 (Effect-locality)**.** *For every element operation $op : E \to E$, position n, list xs, and every position j:*

$$(\texttt{applyAt}\,(\texttt{bimap}\ op\ \texttt{id})\ n\ xs)[j]? \;=\; \begin{cases} (xs[j]?).\texttt{map}\ op & \text{if } j = n \\ xs[j]? & \text{otherwise} \end{cases}$$

Proof. Induction on n, cases on xs; every leaf reduces definitionally. The operation *op* is universally quantified with no side condition and occurs on the right only as $(xs[j]?)$.`map` *op* in the $j = n$ branch. □

Note that for lists, j is an index; when the elements are themselves rows, j names a row. Effect-locality bounds the *write*-footprint: whatever `op` computes, its output lands at position n and nowhere else. Because `op` is a free variable, locality is a property of the skeleton.

1.3.3 The single fold: the paired algebra

Supplying a Lambek pair upgrades the iterate. At a `FIso`, the pair is total and its lens is lawful. Each `natFold` step is then definitionally the lens operation, for every layer transform g:

$$\texttt{applyAt}\ g\ 0 \;=\; \texttt{fiso.toLawfulLens.over}\ g$$

$$\texttt{applyAt}\ g\ (n+1) \;=\; \texttt{fiso.toLawfulLens.over}\ (\texttt{map}\ (\texttt{applyAt}\ g\ n))$$

Both equations hold by `rfl`, and at the list pair they are the primitives (Listing 1.27).

To express the same operation as one structural recursion, we pair the carrier with itself. The first component rebuilds the input, the second computes the result, and the operation is the second projection of a single fold.

Listing 1.29 `pairedFold` is a catamorphism at the paired carrier.

```
def pairedFold [Functor F] [LawfulFunctor F] (iso: FIso F) [rec: IsRecursive iso.toFCoalgebra]
    (step: F (iso.carrier × A) → A): iso.carrier → A :=
  Prod.snd ∘ rec.recHylo ((iso.fold ∘ Functor.map Prod.fst) ▵ step)
```

The algebra `(iso.fold ∘ Functor.map Prod.fst) ▵ step`, where `▵` splits two maps into one product-valued map, is a plain F-algebra at the carrier `iso.carrier × A`. The first component folds the first projections back together, rebuilding the input layer by layer, while the second is the supplied `step`. The `step` therefore receives, at every layer, the rebuilt substructure alongside the recursively computed result.

This pairing is classically the tupling that implements paramorphism (Meertens 1992).

To unfold the position we need the dual of `natFold`; with it, the step at position `n` is:

Listing 1.30 `stepAt` is `applyAt` as a one-layer step.

```
def natUnfold: Nat → Option Nat
  | 0     ⇒ none
  | n + 1 ⇒ some n

def stepAt [Functor F] (iso: FIso F) (g: F iso.carrier → F iso.carrier) (n: Nat):
    F (iso.carrier × iso.carrier) → iso.carrier :=
  match (natUnfold n).map (applyAt iso.carrier iso.fold iso.unfold g) with
  | none   ⇒ fun layer ⇒ iso.fold (g (Functor.map Prod.fst layer))
  | some f ⇒ fun layer ⇒ iso.fold (Functor.map (f ∘ Prod.fst) layer)
```

Both branches use only `Prod.fst`, the rebuilt substructure. The recursively computed component is never consulted, and the rebuilt component is acted on by a fixed function chosen by the position alone – a *content-independent step*.

Theorem 1.17 (Factorisation)**.** *For every `FIso` at a lawful functor with a recursive coalgebra, layer transform g, iterate count n, and carrier element xs:*

$$\texttt{applyAt}\ g\ n\ xs\ =\ \texttt{pairedFold}\ iso\ (\texttt{stepAt}\ iso\ g\ n)\ xs$$

The iterate and the single fold at the paired carrier compute the same function.

Proof. Case analysis on n; both cases follow one chain. Expanding the fold once by its fixed-point equation leaves a goal containing the composite `map Prod.fst` ∘ `map` (`recHylo` *paired*) applied to the unfolded layer, which collapses to the identity by the key lemma: the first component of the paired algebra's fold is the identity on the carrier. The key lemma needs no induction over the carrier. `Prod.fst` is an algebra homomorphism from the paired algebra to `iso.fold`, so by naturality `Prod.fst` ∘ `recHylo` *paired* = `recHylo iso.fold`, and `recHylo iso.fold` = `id` by the fold's uniqueness together with the Lambek round-trip. □

At the list pair, Theorem 1.15 and Theorem 1.17 make the running example of Listing 1.27 read the same in all three spellings, with the carrier arguments elided:

Listing 1.31 The three spellings at the running example.

```
mapTail (mapHead (· + 100))                [1,2,3,4] = [1, 102, 3, 4]
applyAt (bimap (· + 100) id) 1             [1,2,3,4] = [1, 102, 3, 4]
pairedFold (stepAt (bimap (· + 100) id) 1) [1,2,3,4] = [1, 102, 3, 4]
```

Theorem 1.17 is stated at any recursive coalgebra, not only the list pair. Thus, the fold interface survives carriers whose positions the iterate cannot name – the fold carries no `Nat` at all. Its datum is the step, whose type is written in the functor. The addressing vocabulary therefore reshapes with the carrier. At the list functor, a layer exposes one child. At a branching functor, a layer exposes every child. Addressing becomes the choice of which child to descend into, and `stepAt` would accommodate that choice by peeling one edge of a path where `natUnfold` peels one `Nat`.

The factorisation is also a separation of concerns, and it arrives as a theorem: the position fixes *where* the operation acts, the instruction fixes *what* runs there. Scheduling languages maintain this *schedule/instruction split* by convention; here it is forced.

The laws-consulted ledger of Table 1.1 also settles why the three spellings are not instances of one richer interface. An interface is the object together with the laws a client may consult, and the two law systems here constrain different maps: the lens laws constrain the pair itself, while recursivity constrains the unfolding against every algebra at once. Neither implies the other: the iterate bridge held with no law at all, the step equations charged the round trips, and the factorisation charged recursivity on top of them. The unification on offer is the agreement of independently defined interfaces rather than a common signature.

This section establishes addressing, and nothing more than addressing. On a linear carrier, an addressed operation is iterated Under in three agreeing spellings.

The whole iterate is one fold with a content-independent step (Theorem 1.17). Within the elementwise fragment, the write-footprint is the addressed position and nothing else (Theorem 1.16).

1.4 Application: GPU subgroup operations

This section applies the chapter's machinery to GPU subgroup operations.

1.4.1 Shape preservation with ornaments

The previous sections suggested but did not prove that iterated Under operations apply to `Vector` data types. The obstacle is that an operation's output length is an obligation a fold's carrier cannot inherently capture. We therefore rely on extrinsic types.

One might attempt to fold directly over the `Vector` type, which would require an F-algebra with a fixed-width carrier `Vector E n`. This intrinsic approach contradicts the principle folds are built on: a fold assembles structure incrementally.

To elaborate, the carrier is the invariant of assembly. It absorbs one layer's worth of structure at every step. For the list functor, one layer adds one element. Thus every algebra at a constant vector carrier is forced to discard one element's worth of information per layer. The F-algebra must choose what to sacrifice. For example, it could evict the oldest cell, overwrite at a tracked position, or ignore the new element.

Furthermore, the key obligation with structural Under is that output length must equal input length. This cannot be expressed within the algebra itself.

Instead, we adopt an approach familiar from the graded monoids of the previous chapter. A separate indexing algebra must land on the fibre determined by the input type. This means that the `op` in `bimap op id` gains an additional refinement type.

Given an algebra with carrier `A` and a projection `project : A → I`, the fibre over `i : I` is the subtype `{ a : A // project a = i }`. This fibre comprises all elements of `A` that project to `i`.

The key observation is that our algebras operate over the unindexed `List E`. An `n`-producing indexing algebra turns the result into `Vector E n`.

In the context of arrays, such an indexing algebra is an F-algebra that counts the elements that pass through the fold. Showing that an operation's output projects to the same count as its input is how we enforce shape preservation. This allows reuse of the existing list-based F-algebras.

As a reminder from the previous chapter, the obligation is that the following square commutes.

$$\begin{array}{ccc} F(A) & \xrightarrow{alg} & A \\ {\scriptstyle F(\pi)}\downarrow & & \downarrow{\scriptstyle \pi} \\ F(I) & \xrightarrow[idx]{} & I \end{array}$$

Here, alg is the F-algebra of the operation, $idx is the counting algebra, and π is the projection. The square states that π is an F-algebra homomorphism from alg to idx. This enables the fold to land in the ornamented result.

While the previous chapter used this fact for parsing, here it serves as a post-condition. Starting from a `Vector E n` and running alg over the data as a `List E` must produce a result in the fibre over the same `n`. The set of GPU subgroup operations is bounded in number by standards such as SPIR-V. One admission condition can therefore serve the whole instruction set. The shape-preserving operations the chapter models all pass it. The length-changing ones, such as compaction, fail it.

In code, the square is packaged as an ornament (Listing 1.32):

Vector ornamentation thus enables a structural Under in a dependently typed setting: it enforces the shape preservation property apparent in BQN. The Under interface remains the same. Structural Under's definition becomes a

Listing 1.32 The shape certificate: an ornament from the operation's step to the counting algebra.

```
def vectorOrnament (step: Monadic M C → C) (buf: C → List M)
  (hom: ∀ x, (buf (step x)).length = countAlg M (Functor.map (List.length ∘ buf) x)):
    Ornament ⟨C, step⟩ ⟨Nat, countAlg M⟩ where
      project := List.length ∘ buf
      coherence := hom
```

structural `Section` or structural `Iso`, which are classified over ornaments. The definition appears later in Section 1.4.4. Before that, we need to build the class of admitted operations, i.e., the `op` in `bimap op id`.

1.4.2 The admitted operation

An admitted operation for subgroups is an F-algebra homomorphism with the type of a zygomorphism (Listing 1.33).

Listing 1.33 An admitted operation: a channel algebra, a buffer step, and the pinned length ornament.

```
structure SubgroupOp (M B: Type) where
  aux  : Monadic M B → B
  main : Monadic M (B × List M) → List M
  orn  : Ornament ⟨B × List M, zygoStep aux main⟩ ⟨Nat, countAlg M⟩
  proj : orn.project = List.length ∘ Prod.snd
```

In context, the channel algebra `aux` computes a companion value. The buffer step `main` rebuilds the lanes while consulting `aux`. The ornament `orn` is the shape certificate. The pin `proj` fixes that the algebra homomorphism is about the buffer's length. This certifies that an indexed data type such as `Vector` can be computed by a `List`-based algebra.

Note that while the structure is called a subgroup operation, it models a generic shape-preserving buffer operation up to the expressivity of a zygomorphism. A zygomorphism is an extension of an F-algebra (Listing 1.34).

In the extension, the inherited `carrier` and `fold` become the channel. This is an auxiliary value computed alongside the recursion. The buffer step `main` may consult this channel.

Listing 1.34 A one-layer algebra, and its extension with an auxiliary channel.

```
structure FAlgebra (f: Type u → Type v) [Functor f] where
  carrier: Type u
  fold: f carrier → carrier

structure ZygoFAlgebra (f: Type → Type) [Functor f] extends FAlgebra f where
  out: Type
  main: f (carrier × out) → out
```

The channel is forced by typing. A bare catamorphic algebra is handed only the recursively computed result. Thus a peer's original value is absent from its type. It can enter only through this explicit companion.

Running both components together is a fork at the paired carrier. This has the same shape as the paired algebra of Theorem 1.17. The rebuilding fold is replaced by an arbitrary channel algebra (Listing 1.35).

Listing 1.35 The tupled step of a zygomorphism.

```
abbrev zygoStep [Functor f] (aux: f B → B) (main: f (B × A) → A): f (B × A) → B × A :=
  (aux ∘ Functor.map Prod.fst) △ main
```

This construct yields the two constructors for zygo- and catamorphic subgroup instructions (Listing 1.36).

Listing 1.36 Subgroup operations are either zygo or catamorphisms.

```
def SubgroupOp.zygo
  (aux: Monadic M B → B)
  (main: Monadic M (B × List M) → List M)
  (hom: ∀ fx, (main fx).length = countAlg M (Functor.map (List.length ∘ Prod.snd) fx)):
    SubgroupOp M B := ⟨aux, main, vectorOrnament (zygoStep aux main) Prod.snd hom, rfl⟩

def SubgroupOp.cata
  (fold: Monadic M (List M) → List M)
  (hom: ∀ x, (fold x).length = countAlg M (Functor.map List.length x)): SubgroupOp M Unit :=
  .zygo (fun _ ⇒ ()) (fun fx ⇒ fold (Functor.map Prod.snd fx)) (fun
    | none ⇒ hom none
    | some (x, (_, rest)) ⇒ hom (some (x, rest)))
```

Issuance is the level where vectors are converted to lists and back (Listing 1.37). We term this `run`.

Listing 1.37 Issuance: run the fold in the fibre over the width, read the buffer back at the pin.

```
def SubgroupOp.run (o: SubgroupOp M B) (w: Nat): Vector M w → Vector M w :=
  fun v ⇒
    let
      y := o.orn.lift (rec := monadicF_recursive M) (i := w)
      ⟨v.toList, by
        rw [recHylo_is_listFold, ← congrFun length_eq_listFold]
        simp
      ⟩
    ⟨y.val.2.toArray, by
      have h : y.val.2.length = w := (congrFun o.proj y.val).symm.trans y.property
      simp [h]⟩
```

Listing 1.37 is the operational form of the commuting square from Section 1.4.1. `Ornament.lift` runs the fold inside the fibre over the width `w`. The entry proof rewrites the counting algebra's fold to `List.length`. The exit proof reads the fibre property back through the pin.

Issuance is also where the admitted operation meets the theorems of Section 1.3. An issued operation enters the instruction slot of the iterate of Section 1.3.2 at element type `Vector M w`. This is the position-addressed spelling of issuance. There is one subgroup per layer. Theorem 1.15 was stated for every carrier and every instruction. Instantiating it at the chunked carrier with the issued operation costs no new proof.

$$\texttt{mapTail}^{[i]}\,(\texttt{mapHead}\ (o.\texttt{run}\ w))\ =\ \texttt{applyAt}\ (\texttt{bimap}\ (o.\texttt{run}\ w)\ \texttt{id})\ i$$

The same reading turns effect-locality into the device guarantee the chapter promised.

Theorem 1.18 (Grid frame)**.** *For every admitted operation o, width w, buffer $cs : \texttt{List}\,(\texttt{Vector}\ M\ w)$, addressed index i, and position $j \neq i$:*

$$(\texttt{mapTail}^{[i]}\,(\texttt{mapHead}\ (o.\texttt{run}\ w))\ cs)[j]?\ =\ cs[j]?$$

Proof. The statement is the off-focus branch of Theorem 1.16 at element type `Vector` M w, spelled through the iterate. The mechanised proof replays that

branch's induction on the addressed index, peeling one `mapTail` per step. The instruction slot is universally quantified. Thus the issued operation enters with no side condition. □

The quantifiers deliver the key benefit. Theorem 1.15 and Theorem 1.16 hold for an arbitrary instruction at an arbitrary element type. The certificate allows an issued operation to inhabit that quantifier. The slot accepts any endofunction at `Vector M w`. This is a type the operation holds only through `run`. Without the ornament it exists only at `List M`.

Instantiation then returns shape preservation at both levels. Within a subgroup, the ornament pins the width. `run` lands in the fibre over `w` regardless of what the algebra computes. Across the grid, the content-independent step rebuilds the spine it walked. Thus the buffer keeps its extent. The grid frame (Theorem 1.18) keeps the off-focus chunks intact.

The frame is uniform over the operation. No matter how an instruction rebuilds the lanes inside its subgroup, the outer theorem remains agnostic to its internals. The outer layer addresses a subgroup. The inner layer is the admitted operation, generic in its instruction and a morphism of its own.

1.4.3 The operation families

Next, we model four families that cover the subgroup operations introduced in Section 1.1.1.

Table 1.3: The operation families. The channel type displays each specification's budget, and every family executes as one traversal.

Family	Device operation	Channel `B`	One-layer step
`reduce`	group arithmetic, `Reduce` mode	`Unit`	constant write-back (`mapAlg`)
`scan`	`InclusiveScan`, `ExclusiveScan` modes	`Unit`	prefix step (`scanAlg`)

Family	Device operation	Channel `B`	One-layer step
`address`	single-lane write at a coordinate	`Nat`	one-cell update (`modifyAlg`)
`broadcast`	`Broadcast`, `BroadcastFirst`	`Nat × Option M`	capture and zoned write-back (`broadcastAlg`)

In the channel column of Table 1.3, `Unit` marks the families that need no companion value. These are definitionally catamorphisms. Conversely, those with richer carriers are zygomorphic. They cover cases where a value must be located within the list.

Thus the channel type classifies the specification: we call it the family's *channel budget*. It displays how much context the operation's defining equations need. Meanwhile, the lockstep hardware executes every family the same way. It performs a map over lanes once the context is materialised.

1.4.3.1 Catamorphisms

The reduce family writes one value to every lane. Its algebra is elementwise map. This is packaged as a one-layer fold (Listing 1.38).

Listing 1.38 The reduce family: the write-back algebra at a constant.

```
def mapAlg (f: E → A): FAlgebra (Monadic E) where
  carrier := List A
  fold := fun
    | none ⇒ []
    | some (x, xs) ⇒ f x :: xs

def SubgroupOp.reduce (c: M): SubgroupOp M Unit :=
  .cata (mapAlg (fun _ ⇒ c)).fold (mapAlg.hom _)
```

The scan family accumulates along the lanes (Listing 1.39).

`first := id` yields an inclusive scan. `first := fun _ ⇒ I` yields an exclusive scan at identity `I`. No algebraic constraints are demanded of `op`. Thus the family ranges over the whole arithmetic, bitwise, and logical instruction set.

Listing 1.39 The scan family.

```
def scanAlg (op: M → M → M) (first: M → M): FAlgebra (Monadic M) where
  carrier := List M
  fold := fun
    | none ⇒ []
    | some (x, xs) ⇒ first x :: xs.map (op x)

def SubgroupOp.scan (op: M → M → M) (first: M → M): SubgroupOp M Unit :=
  .cata (scanAlg op first).fold (scanAlg.hom _ _)
```

A `GroupOperation` operand selects the family. Its three constructors are the core `GroupOperation` modes of SPIR-V; extended modes such as clustered reduction are outside the model. An arithmetic operand is a bare one-layer reduction algebra `Monadic M M → M`. Its two cases encode the operation and its identity (Listing 1.40).

Listing 1.40 Group instructions that model execution contexts of SPIR-V.

```
inductive GroupOperation | Reduce | InclusiveScan | ExclusiveScan

def groupInstr (fold: Monadic M M → M) (gop: GroupOperation) (sgLen: Nat)
    (v: Vector M w) (_h: w = sgLen): Vector M w :=
  SubgroupOp.run
    (match gop with
     | .Reduce        ⇒ .reduce (listFold fold v.toList)
     | .InclusiveScan ⇒ .scan (fun x r ⇒ fold (some (x, r))) id
     | .ExclusiveScan ⇒ .scan (fun x r ⇒ fold (some (x, r))) (fun _ ⇒ fold none))
    w v
```

We can exercise the dispatcher by instantiating the arithmetic operand at the addition algebra. Empty layer maps to `0` and cons layer maps to `+`. We can also use the maximum algebra. Empty layer maps to `0` and cons layer maps to `max` (Listing 1.41).

Listing 1.41 The dispatcher at the addition and maximum algebras.

```
#eval (groupInstr (sum _).fold .Reduce 4 ⟨#[1, 2, 3, 4], rfl⟩ (by simp)).toList
-- expect [10, 10, 10, 10]
#eval (groupInstr (sum _).fold .ExclusiveScan 4 ⟨#[1, 2, 3, 4], rfl⟩ (by simp)).toList
-- expect [0, 1, 3, 6]
#eval (groupInstr (maxAlg _).fold .Reduce 4 ⟨#[3, 9, 2, 7], rfl⟩ (by simp)).toList
-- expect [9, 9, 9, 9]
```

1.4.3.2 Zygomorphisms

The addressed family implements the single-lane write from Section 1.1.1. Its algebra is the one-cell update. This is a channel-consulting fold whose channel is the counting algebra itself (Listing 1.42).

Listing 1.42 The addressed family: the one-cell update over the count channel.

```
def modifyAlg (f: M → M) (j: Nat): ZygoFAlgebra (Monadic M) where
  carrier := Nat
  fold := countAlg M
  out := List M
  main := fun
    | none               ⇒ []
    | some (x, (n, rec)) ⇒ (if n = j then f x else x) :: rec

def SubgroupOp.address (f: M → M) (j: Nat): SubgroupOp M Nat :=
  .zygo (modifyAlg f j).fold (modifyAlg f j).main (modifyAlg.hom f j)

def addressVec (f: M → M) (sgLen: Nat) (v: Vector M w) (i: Nat)
    (_h: w = sgLen) (_hi: i < sgLen): Vector M w :=
  (SubgroupOp.address f (w - 1 - i)).run w v

#eval (addressVec (· + 100) 4 ⟨#[1, 2, 3, 4], rfl⟩ 1 (by simp) (by simp)).toList
-- expect [1, 102, 3, 4]
```

The broadcast family moves one lane's value to all lanes in a single traversal (Listing 1.43). In SPIR-V, `Broadcast` and `BroadcastFirst` are standalone instructions rather than `GroupOperation` modes. `BroadcastFirst` reads the lowest active lane, which in the all-active lockstep model here coincides with an explicit leader lane.

1.4.4 Issuance through selections

With the classification in place, we can model the class of issuable operands (Listing 1.44).

These instances cover both zygo- and catamorphisms, as well as their group-operation variants. We then arrive at the definition of a **typed Structural Under** (Listing 1.45).

Listing 1.43 The broadcast family: the channel captures the source value in-band.

```
def broadcastAlg (j: Nat): ZygoFAlgebra (Monadic M) where
  carrier := Nat × Option M
  fold := fun
    | none               ⇒ (0, none)
    | some (x, (n, val)) ⇒ (n + 1, if n == j then some x else val)
  out := List M
  main := fun
    | none                      ⇒ []
    | some (x, ((n, val), rec)) ⇒
        if n < j then x :: rec
        else if n == j then x :: rec.map (fun _ ⇒ x)
        else (optionAlg x).fold val :: rec

def SubgroupOp.broadcast (j: Nat): SubgroupOp M (Nat × Option M) :=
  .zygo (broadcastAlg j).fold (broadcastAlg j).main (broadcastAlg_hom j)

def broadcastVec (sgLen: Nat) (v: Vector M w) (l: Nat)
    (_hl: l < sgLen) (_hw: w = sgLen): Vector M w :=
  (SubgroupOp.broadcast (w - 1 - l)).run w v

#eval (broadcastVec 4 ⟨#[7, 2, 3, 4], rfl⟩ 0 (by simp) (by simp)).toList
-- expect [7, 7, 7, 7]
#eval (broadcastVec 4 ⟨#[7, 2, 3, 4], rfl⟩ 2 (by simp) (by simp)).toList
-- expect [3, 3, 3, 3]
```

This is the chapter's second spelling of addressed issuance. Issuance through the slot (Section 1.4.2) addresses through a position, and its frame is the grid frame (Theorem 1.18). The dispatch here addresses through a selection, and its frame is the selection's complement (Theorem 1.12).

Two points warrant emphasis. First, selection occurs through the lens framing of Under. This uses the `under` defined for structural sections (Listing 1.18). An isomorphism's Under is that same action with the converse law adjoined. Thus a selection built from a structural isomorphism runs the section's `under` while the iso-tier theorems apply to it. Because the selection happens through the lens route, **Structural Under is type-polymorphic**.

The second point is that the dispatch also completes the schedule/instruction

Listing 1.44 The class of issuable operands and its instances.

```
class Issuable (τ: Type) (M: outParam Type) (w: Nat) where
  issue : τ → Vector M w → Vector M w

instance : Issuable (SubgroupOp M B) M w where
  issue o := o.run w

instance : Issuable ((Monadic M M → M) × GroupOperation) M w where
  issue iw v := groupInstr iw.1 iw.2 w v rfl
```

Listing 1.45 Typed Structural Under: issuance through a selection.

```
def StructuralUnder [Issuable τ M w] (sel: StructuralSection S (Vector M w)) (o: τ): S → S :=
  sel.under (Issuable.issue o)
```

split of Theorem 1.17 as types. Here, the schedule is the selection. This is a lens fixed before any data arrives. The instruction is the admitted operation. This receives the view and nothing else.

A scheduling language keeps the two in separate syntactic categories by convention. Here they sit in separate types with separate laws. The selection cannot inspect the buffer. The instruction cannot read or write outside its view. Placement across selections belongs to the schedule. Placement within a view is the instruction's purview. This is how the addressed family carries its lane operand without breaching the split.

1.4.5 Subgroup operations: revisited

The selections demonstrated are structural isomorphisms over the vector carrier, built from Mathlib equivalences (Listing 1.46). `chunk` reshapes a flat buffer into subgroups, and `strided` is the transposed reshape at a stride. `slotAt` selects one element, and `laneAt` re-views it as a width-1 subgroup. `chunkAt` selects one subgroup of the flat buffer, with the address carried by the selection rather than by any lane's data.

With the vocabulary in place, we can issue the families through selections (Listing 1.47).

Listing 1.46 The selection vocabulary: reshaping and addressing isomorphisms over the vector carrier.

```
def StructuralIso.chunk (n m: Nat): StructuralIso (Vector E (m * n)) (Vector (Vector E n) m)
def StructuralIso.strided (w m: Nat): StructuralIso (Vector E (m * w)) (Vector (Vector E m) w)
def StructuralIso.slotAt (i: Fin n): StructuralIso (Vector α n) α
def StructuralIso.laneAt (i: Fin n): StructuralIso (Vector α n) (Vector α 1)
def StructuralIso.chunkAt (w m: Nat) (i: Fin m): StructuralIso (Vector E (m * w)) (Vector E w)
```

Listing 1.47 The map family, issued through the chunked and the addressed selections.

```
def SubgroupOp.map (f: M → M): SubgroupOp M Unit :=
  .cata (mapAlg f).fold (mapAlg.hom _)

#eval (StructuralUnder
        (StructuralIso.chunk 4 3).toStructuralSection
        (SubgroupOp.map ((SubgroupOp.scan (· + ·) id).run 4))
        ⟨#[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12], rfl⟩).toList
-- every subgroup scanned: [1, 3, 6, 10, 5, 11, 18, 26, 9, 19, 30, 42]

#eval (StructuralUnder
        (StructuralIso.chunkAt 4 3 1).toStructuralSection
        (SubgroupOp.scan (· + ·) id)
        ⟨#[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12], rfl⟩).toList
-- subgroup 1 scanned, others untouched (the addressed flat dispatch):
-- [1, 2, 3, 4, 5, 11, 18, 26, 9, 10, 11, 12]
```

Both variants of Structural types, sections and isos, compose (Listing 1.48).

`chunkAt` is itself such a composite: `chunkAt w m i` is `comp (chunk w m) (slotAt i)`.

For example, consider disjoint footprints: lane (0,2) and lane (2,0) (Listing 1.49).

Both evaluations return `[1, 2, 103, 4, 5, 6, 7, 8, 18, 10, 11, 12]`. This commutation property licenses any-order parallel issuance.

The failure of commutation can itself be formalised as a theorem (Listing 1.50).

If we run two operations over the same buffer, we can use Mathlib's tooling to show that such a composition cannot be run in parallel. The independence structure of Section 1.2 turns this into complement-level reasoning.

Distinct subgroup selections of the flat buffer are independent, and the witness is mechanised. One joint decomposition exhibits both subgroups side by side

Listing 1.48 Composition of structural isomorphisms and structural sections.

```
def StructuralIso.comp (outer: StructuralIso S V) (inner: StructuralIso V W): StructuralIso S W :=
  StructuralIso.ofProdEquiv ◁
    outer.toEquiv
      ▷.trans (Equiv.prodCongr inner.toEquiv (Equiv.refl outer.Complement))
      ▷.trans (Equiv.prodAssoc W inner.Complement outer.Complement)

def StructuralSection.comp (outer: StructuralSection S V) (inner: StructuralSection V W):
    StructuralSection S W where
  Complement := inner.Complement × outer.Complement
  decomp s :=
    ((inner.decomp (outer.decomp s).1).1,
     ((inner.decomp (outer.decomp s).1).2, (outer.decomp s).2))
  recomp := fun (w', (ci, co)) ⇒ outer.recomp (inner.recomp (w', ci), co)
  left_inv s := by
    show outer.recomp (inner.recomp (inner.decomp (outer.decomp s).1), (outer.decomp s).2) = s
    rw [inner.left_inv]
    exact outer.left_inv s
```

over the shared frame of the remaining ones. Thus Theorem 1.13 delivers the commutation observed above for every pair of operands. And Theorem 1.14 carries local specifications across the parallel composite. The refutation composes in the opposite direction, reusing the non-commuting pair above (Listing 1.51).

In separation logic terms, no separating conjunction splits the buffer into these two views at any complement. This has one particularly interesting application, which is to derive memory barriers from composition of structural Unders. I.e., whenever compositions do not commute, a barrier is entered. Barrier minimisation then becomes a scheduling objective.

The read-footprint boundary. The complement also bounds what an instruction may read. By Theorem 1.11, the view of an updated source is the instruction applied to the view of the original. Thus the instruction receives the view and nothing else. The sandbox is refutable, which makes explicit the boundary effect-locality could not see: Theorem 1.16 bounds only the write-footprint. Consider a transformation with the same write-footprint as a lane selection (Listing 1.52). Its update nevertheless consults a neighbouring lane.

Listing 1.49 Two operations at disjoint lane footprints, issued in both orders.

```
def sampleBuf : Vector Int (3 * 4) := ⟨#[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12], rfl⟩

#eval (StructuralUnder
        (StructuralIso.comp (StructuralIso.chunkAt 4 3 0) (StructuralIso.laneAt 2)).toStructuralSection
        (SubgroupOp.map (· + 100))
        (StructuralUnder
          (StructuralIso.comp (StructuralIso.chunkAt 4 3 2) (StructuralIso.laneAt 0)).toStructuralSection
          (SubgroupOp.map (· * 2))
          sampleBuf)).toList
#eval (StructuralUnder
        (StructuralIso.comp (StructuralIso.chunkAt 4 3 2) (StructuralIso.laneAt 0)).toStructuralSection
        (SubgroupOp.map (· * 2))
        (StructuralUnder
          (StructuralIso.comp (StructuralIso.chunkAt 4 3 0) (StructuralIso.laneAt 2)).toStructuralSection
          (SubgroupOp.map (· + 100))
          sampleBuf)).toList
```

Listing 1.50 A non-commuting pair as a theorem: the footprints overlap.

```
theorem scan_bump_not_commute : ¬ Function.Commute
    (StructuralUnder (StructuralIso.chunkAt 4 3 1).toStructuralSection
      (SubgroupOp.scan (· + ·) id : SubgroupOp Int Unit))
    (StructuralUnder
      (StructuralIso.comp (StructuralIso.chunkAt 4 3 1) (StructuralIso.laneAt 1)).toStructuralSection
      (SubgroupOp.map (· + 100))) :=
  fun h ⇒ absurd (h sampleBuf) (by native_decide)
```

The proof instantiates the complement-preservation equation at two buffers. `⟨1, 2, 3⟩` and `⟨-1, 2, 3⟩` present the same view `⟨2⟩` to every instruction at the lane selection. Thus every Under at that selection writes the same lane value to both. However, the transformation demands `102` on one and `2` on the other. The write-footprint alone does not admit an operation. The read set must also lie inside the view. Widening the selection re-admits the transformation.

At the two-slot decomposition of slots `0` and `1`, the instruction `fun p ⇒ if p.1 > 0 then (p.1, p.2 + 100) else p` produces it as a plain Under. The data dependence is internal to the view.

An operation's admissible selections are exactly those whose view contains its read set. Content dependence is priced as view width. This is the selection-level

Listing 1.51 The race refuted at the complement level: no independence witness at any complement.

```
example (C: Type):
    IsEmpty (Independent
      (StructuralIso.chunkAt (E := Int) 4 3 1).toStructuralSection
      (StructuralIso.comp (StructuralIso.chunkAt (E := Int) 4 3 1)
        (StructuralIso.laneAt 1)).toStructuralSection
      C) :=
  ⟨fun h ⇒ scan_bump_not_commute (h.under_comm _ _)⟩
```

Listing 1.52 A single-lane write that consults a neighbour admits no Under at the lane selection.

```
def modifyIfPositive (f: Int → Int) (v: Vector Int 3): Vector Int 3 :=
  if v[0] > 0 then v.set 1 (f v[1]) else v

abbrev lane1 : StructuralIso (Vector Int 3) (Vector Int 1) := StructuralIso.laneAt 1

theorem modifyIfPositive_not_under_laneAt :
    ¬ ∃ f, lane1.toStructuralSection.under f = modifyIfPositive (· + 100)
```

analogue of the channel budget of Table 1.3.

As a final example, consider a workgroup sum (Listing 1.53).

Listing 1.53 The workgroup sum as a composite of Structural Unders.

```
def workgroupSumBuf (fold: Monadic M M → M) (w: Nat) (hw: 0 < w):
    Vector M (w * w) → Vector M (w * w) :=
  StructuralUnder
    (StructuralIso.comp (StructuralIso.strided w w) (StructuralIso.slotAt ⟨0, hw⟩)).toStructuralSection
    (fold, GroupOperation.Reduce)
  ∘ StructuralUnder
    (StructuralIso.chunk w w).toStructuralSection
    (SubgroupOp.map (fun v ⇒ groupInstr fold .Reduce w v rfl))

#eval (workgroupSumBuf (sum _).fold 4 (by omega)
        (⟨#[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16], rfl⟩)).toList
-- expect [136, 10, 10, 10, 136, 26, 26, 26, 136, 42, 42, 42, 136, 58, 58, 58]
```

This shows how we can build higher-level operations from subgroup operations. The sum of a 4×4 input array is computed at the subgroup level by composing structural Unders.

The first operation chunks the buffer into four subgroups of four lanes. It applies

a sum reduction within each subgroup. This writes each subgroup's sum to all of its lanes.

The second operation composes striding, another structural isomorphism, with a slot selection. This picks out the first lane of every subgroup. The reduction over this composed view leaves the workgroup sum in the first lane of every subgroup.

1.5 Discussion

1.5.1 Related work

Inverse-based GPU memory mapping. Closest in intent is the line of work on mapping multi-dimensional arrays onto GPU memory by computing index inverses. Janssen and Scholz (2021) is representative prior work.

There the inverse that reassembles addressed memory is constructed by hand for each mapping. Here the reassembly is the automatic backward direction of the Under lens, and correctness is the discharged lens law rather than a hand-checked inverse. The shift is from manual inverses to a structural construction where the preservation property holds by theorem.

Lens-based frames in Isabelle/UTP. Closest in vocabulary is Isabelle/UTP. It models program variables as lenses into an abstract state space. It rebuilds the meta-logic of the laws of programming on a lens algebra (Foster et al. 2016, 2020).

There, independence of two lenses is defined by put-commutation together with two get-stability conditions, which for total lenses reduce to put-commutation alone. Frames are lens sets. The authors observe that lenses modulo equivalence form a partial commutative monoid. This is a separation algebra with independence as its separateness relation.

Their framed substitution applies a state transformer through a lens and re-embeds the result. This is precisely the conjugation this chapter calls structural

Under. It is applied there to variable frames over record-like state.

The difference is the direction of definition. Isabelle/UTP takes put-commutation as the meaning of independence. It builds disjoint state division from it. Here commutation is a theorem (Theorem 1.13). It is derived from a joint constant-complement decomposition. Thus independence has a witness object. A data-race is refuted by showing that no witness exists at any complement (Section 1.4).

Their frames are likewise spelled through put without naming a complement. The constant-complement route reads the preserved frame back as data (Theorem 1.11). Finally, the selections here are structured array decompositions carrying shape certificates. The same lens machinery dispatches certified device operations rather than framing substitutions. The line has since stated lens-based frame rules for cyber-physical verification (Huerta y Munive et al. 2024). An invariant conjoins across a program when a non-modification side condition holds. This is derived compositionally for each program. By contrast, Theorem 1.12 has no side condition to derive. The selection discharges it once for every instruction.

Concurrency in this line is UTP's parallel-by-merge. Its commutativity is conditioned on symmetry of the merge predicate, rather than on disjointness of footprints. A footprint-splitting rule in the sense of Theorem 1.14 is not stated in these works. To our knowledge, the parallel rule has not previously been derived from lens laws. This holds even in its disjoint form. The metatheory of concurrent separation logic derives its rules from partial commutative monoids. In our work, the machinery is constant-complement decompositions.

Verified GPU programming with separation logic. Closest in scope and technique is Kuiper (Martínez et al. 2026). It verifies GPU compute kernels in Pulse. Pulse's logic, PulseCore (Ebner et al. 2025), is a concurrent separation logic embedded in F⋆ in the lineage of Iris (Jung et al. 2018).

Concurrent separation logic in Pulse is based on resource algebras via partial commutative monoids. These have the expressivity to model memory visibility of data movements. They also model correctness of synchronisation barriers and resource ownership in general.

On top of these, Kuiper names a theory of located resources as a necessary component of GPU verification. Resources are indexed by their place in the visibility hierarchy. They are instantiated to verified data abstractions such as views, matrices, and tiling. The selections of Section 1.4 are this chapter's counterpart to those abstractions. The chunk, stride, and slot decompositions provide views and tilings as lawful lenses. These are formulated as array operations that parametrise subgroup widths. The location discipline is carried by the complement type rather than by a resource logic. Kuiper's kernel-launch rule lifts a single thread's specification to a whole-kernel one. This is the same lift that effect-locality (Theorem 1.16) performs from one subgroup chunk to the whole buffer, and that the grid frame (Theorem 1.18) performs for issued operations.

The difference between the two systems is what the lift rests on. In Kuiper, the lift is a per-kernel proof obligation. This is largely automated by SMT solvers. In this work, no per-program obligation arises, so an SMT solver is not needed for the separation proofs. Proofs for data-races use Lean's compiled evaluation (`native_decide`).

The proof methodology here follows the recursive-coalgebra treatment of hylomorphisms (Hinze et al. 2015). The factorisation theorem is a uniqueness-of-hylomorphisms argument. The shape certificate is just an ornamentation of the catamorphic side, but applies also to hylomorphisms. The recursion-side theorems, the factorisation and its locality corollaries, are stated at an arbitrary Lambek pair. The dispatch-side theorems, the frame and parallel rules, are stated at an arbitrary constant-complement decomposition. Operation widths appears only when a concrete selection such as `chunk` is instantiated. Thus, a verified operation ports across subgroup widths without re-proof in the underlying

internals. Kuiper does not surface operation width variability as it compiles to NVIDIA CUDA, which fixes the warp width to a constant. This eliminates their need for operation width parametricity. By contrast, the operations modelled here follow SPIR-V, where the subgroup width is hardware dependent. Width parametricity is thus a necessary abstraction, at both the whole-program and the individual-operation level.

The trade runs in both directions. Kuiper's per-kernel proofs establish full functional correctness and data-race freedom. Its kernels extract to CUDA that runs at cuBLAS-competitive speed, none of which this chapter attempts. Our structural route covers only the spatial separation of lockstep lanes. Barriers, ownership transfer, and atomic memory transactions that Pulse models lie outside the fragment presented here.

Ownership-typed GPU memory safety. Descend (Köpcke et al. 2024) reaches a neighbouring guarantee through the type system. Kernels are written against an ownership discipline with borrow checking extended to the GPU's execution hierarchy, so disjointness of writes is enforced at compile time. That is the same disjointness the complement carries here, but through borrowship rather than by a named frame. This means that the difference is what the types track: Descend tracks who may hold a reference, while our selection tracks what remains invariant.

1.5.2 Limitations and future work

General-purpose GPU programming is about the device-host programming model. This chapter addressed only the device side.

The host layer orchestrates the device from the CPU. This includes enumerating physical devices, opening queue families, allocating buffers, issuing memory transfers, and synchronising through fences and semaphores. These are computational effects or monads in the sense of Moggi, not APL (Moggi 1991). The algebraic vocabulary corresponds to a T-algebra and more generally Eilenberg-Moore categories. Handling host-side orchestration with T-algebras and recur-

sion morphisms would be a logical extension in future work. E.g., combining results between multiple GPUs inherently requires operations on the CPU side, which is necessary to model multi-GPU execution.

The parallel rule (Theorem 1.14), instantiated at sibling subgroups in Section 1.4, is a model of exclusive references. The read-footprint boundary of Section 1.4 is the sequential face of the same exclusivity. A view is a joint read and write capability. Thus an operation reading outside its write slice must widen its view. Two write-disjoint operations whose reads overlap admit no independence witness.

In comparison, concurrent separation logic is often used to prove theorems about shared references via fractional permissions (Boyland 2003). This is imported into separation logic as permission accounting (Bornat et al. 2005). In our approach, a fractional refinement of the joint decomposition would relax the parallel rule's disjointness to the write-disjoint, read-overlapping case. This would make such shared references parallelisable as well.

The refinement is known not to be free. Bornat et al. (2005) record that fractions weaken inductively defined predicates. This is because such predicates enforce structure through disjointness. Disjointness is exactly what fractions relax. A bijective decomposition offers a fraction no home at all.

However, future work could move the fractions into the index. The instrument is an ornament graded by a free abelian group. This is the dimension-typed grading of McBride and Nordvall-Forsberg (2022) with buffer positions in place of base dimensions. With the group structure, splitting and reclaiming a permission is cancellation. This would necessite a sharing-aware commutation theorem.

This work used a paramorphic type to model addressing. A dual, apomorphic scheme is another direction. Instead of descending a flat array towards slices by iterated addressing, it would grow an index structure from a single seed to the full buffer shape before folding the computation onto it. In our total setting, the input shape would serve as the termination boundary of such an unfold. This

turns the ceremony of building a layout on its head: a programmer starts with the operation and considers how the layout can grow to the shape of the input. Constructing the input shape, especially in a dependently typed setting, may also be a clearer proof objective than navigating the hierarchy with a separate performance model in mind.

Another interesting area of future work would be to mechanise the concept of barrier construction. This would push compositions of structural Unders into barriers. Memory barriers hence parallelism could then be derived from the program. The barriers would be inserted whenever the operations start to overlap slices. A pattern of recursion morphisms, called dynamorphism, could be used to possibly automate the construction of the said barriers. As memory is slower than execution, a dynamorphism which minimises the count of barriers would become a scheduling objective. This would enable a sort of structural program optimisation.

The lens laws and the derived theorems for parallelism are type-polymorphic. While the work focused solely on array representations, an avenue of future work could therefore focus on tree structures.

This work also had to rely on extrinsically typed proofs for shape-preservation. This is done by showing that a projection is an F-algebra homomorphism from the operation's algebra to a separate indexing algebra. Having the operations be intrinsically typed within the folds would iron out the wrinkle of resorting to extrinsic types.

1.5.3 Conclusion

This chapter presented what is, to our knowledge, the first dependently typed mechanisation of BQN's structural Under using lens laws. It tied this into extrinsically typed recursion schemes with an application for GPU computation.

Naming the lens complement made memory safety structural. An update cannot write its complement (Theorem 1.11), the frame and parallel rules of disjoint

concurrent separation logic follow as equations, and effect-locality carries the same guarantee at the iterate. The factorisation theorem separates schedule from instruction, and the typed dispatch completes that split as types.

We showed examples of GPU programs made entirely of structural Unders. Such compositions can be reasoned about in Lean 4 through the correspondence layer between lenses and the frame and parallel rules in the concurrent separation logic sense.

The motivation of this approach was to parametrise subgroup widths. No theorem fixes a width, so a verified operation ports across subgroup widths without re-proof. This could allow later developments to use the model for heterogeneous code generation targeting platform-agnostic intermediate representations like SPIR-V.

The lens laws and the parallelism theorems are type-polymorphic; whether they carry beyond arrays, to tree carriers, is left as future work.

Bancilhon, F., and N. Spyratos. 1981. "Update Semantics of Relational Views." *ACM Transactions on Database Systems* 6 (December): 557–75. https://doi.org/10.1145/319628.319634.

Bird, Richard, and Oege de Moor. 1997. *Algebra of Programming.* Prentice-Hall, Inc.

Bornat, Richard, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. 2005. "Permission Accounting in Separation Logic." *ACM SIGPLAN Notices* 40 (January): 259–70. https://doi.org/10.1145/1047659.1040327.

Boyland, John. 2003. "Checking Interference with Fractional Permissions." In *Lecture Notes in Computer Science.* Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-44898-5_4.

Brookes, Stephen. 2007. "A Semantics for Concurrent Separation Logic." *Theoretical Computer Science* 375 (May): 227–70. https://doi.org/10.1016/j.tcs.2006.12.034.

Ebner, Gabriel, Guido Martínez, Aseem Rastogi, et al. 2025. "PulseCore: An Impredicative Concurrent Separation Logic for Dependently Typed Programs." *Proceedings of the ACM on Programming Languages* 9 (June): 1516–39. https://doi.org/10.1145/3729311.

Foster, J. Nathan, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2005. "Combinators for Bi-Directional Tree Transformations: A Linguistic Approach to the View Update Problem." *POPL05: The 32nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages 2005*, January, 233–46. https://doi.org/10.1145/1040305.1040325.

Foster, Simon, James Baxter, Ana Cavalcanti, Jim Woodcock, and Frank Zeyda. 2020. "Unifying Semantic Foundations for Automated Verification Tools in Isabelle/UTP." *Science of Computer Programming* 197 (October): 102510. https://doi.org/10.1016/j.scico.2020.102510.

Foster, Simon, Frank Zeyda, and Jim Woodcock. 2016. "Unifying Heterogeneous State-Spaces with Lenses." In *Lecture Notes in Computer Science*. Springer International Publishing. https://doi.org/10.1007/978-3-319-46750-4_17.

Hinze, Ralf, Nicolas Wu, and Jeremy Gibbons. 2015. "Conjugate Hylomorphisms – or: The Mother of All Structured Recursion Schemes." *POPL '15: The 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January, 527–38. https://doi.org/10.1145/2676726.2676989.

Hoare, C. A. R. 1975. "Recursive Data Structures." *International Journal of Computer &Amp; Information Sciences* 4 (June): 105–32. https://doi.org/10.1007/bf00976239.

Huerta y Munive, Jonathan Julián, Simon Foster, Mario Gleirscher, Georg Struth, Christian Pardillo Laursen, and Thomas Hickman. 2024. "IsaVODEs: Interactive Verification of Cyber-Physical Systems at Scale." *Journal of Automated Reasoning* 68 (December). https://doi.org/10.1007/s10817-024-09709-2.

Iverson, Kenneth. 1972. *Algebra: An Algorithmic Treatment*.

Janssen, Niek, and Sven-Bodo Scholz. 2021. "On Mapping n-Dimensional Data-Parallelism Efficiently into GPU-Thread-Spaces." *IFL '21: 33rd Symposium on Implementation and Application of Functional Languages*, September, 54–66. https://doi.org/10.1145/3544885.3544894.

Jung, Ralf, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. "Iris from the Ground up: A Modular Foundation for Higher-Order Concurrent Separation Logic." *Journal of Functional Programming* 28 (November). https://doi.org/10.1017/s0956796818000151.

Köpcke, Bastian, Sergei Gorlatch, and Michel Steuwer. 2024. "Descend: A Safe GPU Systems Programming Language." *Proceedings of the ACM on Programming Languages* 8 (June): 841–64. https://doi.org/10.1145/3656411.

Kromberg, Morten. 2023. *Structural Vs Mathematical Under*. https://web.archive.org/web/20260305172746/https://www.dyalog.com/blog/2023/01/structural-vs-mathematical-under/.

Lochbaum, Marshall. 2020. *BQN's Development History*. https://web.archive.org/web/20251201044947/https://mlochbaum.github.io/BQN/commentary/history.html.

Martínez, Guido, Bastian Köpcke, Jonáš Fiala, et al. 2026. "Kuiper: Correct and Efficient GPU Programming with Dependent Types and Separation Logic." *Proceedings of the ACM on Programming Languages* 10 (June): 830–54. https://doi.org/10.1145/3808280.

McBride, Conor, and Fredrik Nordvall-Forsberg. 2022. "Type Systems for Programs Respecting Dimensions." In *Series on Advances in Mathematics for Applied Sciences*. WORLD SCIENTIFIC. https://doi.org/10.1142/9789811242380_0020.

Meertens, Lambert. 1992. "Paramorphisms." *Formal Aspects of Computing* 4 (September): 413–24. https://doi.org/10.1007/bf01211391.

Meijer, Erik, Maarten Fokkinga, and Ross Paterson. 1991. "Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire." In *Functional Programming Languages and Computer Architecture*. Springer Berlin Heidelberg. https://doi.org/10.1007/3540543961_7.

Moggi, Eugenio. 1991. "Notions of Computation and Monads." *Information and Computation* 93 (July): 55–92. https://doi.org/10.1016/0890-5401(91)90052-4.

O'Hearn, Peter W. 2007. "Resources, Concurrency, and Local Reasoning." *Theoretical Computer Science* 375 (May): 271–307. https://doi.org/10.1016/j.tcs.2006.12.035.

Reynolds, J. C. 2002. "Separation Logic: A Logic for Shared Mutable Data

Structures." *17th Annual IEEE Symposium on Logic in Computer Science*, 55–74. https://doi.org/10.1109/lics.2002.1029817.

Yang, Hongseok, and Peter O'Hearn. 2002. "A Semantic Basis for Local Reasoning." In *Lecture Notes in Computer Science*. Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-45931-6_28.