Generic Combinators for Monoid Nesting in Futhark

Towards a library for data-parallel one-touch queries in constant space

MARTIN ELSMAN, University of Copenhagen, Denmark (mael@di.ku.dk)

Associative reductions support parallel queries, but structured input often requires several stages of parsing and aggregation. Composing these stages into one reduction is challenging when delimiters and parentheses determine where intermediate results are consumed. We present a Futhark library that equips monoids with input generators and result projections, and combines them into one-touch queries without constructing syntax trees or intermediate sequences. Building on delimiter-based composition, the library provides floating-point parsing, horizontal combinators and bounded parenthesis nesting with fixed-capacity tuple summaries. Rocq developments establish monoid preservation. CPU experiments demonstrate useful parallel scaling, while revealing substantial sensitivity to summary representation, nesting depth and input structure. For maximum-of-group-averages on a laptop GPU, sequential reductions of contiguous input blocks improve execution time by approximately 5–6× over the direct reduction, without changing the monoids. On the configured ThinkPad, for 100 MB inputs, the blocked GPU query is 3.8–4.4× faster than handwritten sequential C and 1.6–2.2× faster than the four-worker multicore query, excluding input loading and GPU transfers. On the same ThinkPad, for one-level parenthesised expressions, blocked GPU evaluation with a restricted-decimal parser is 4.1× faster than sequential C on regular 100 MB input, but only 1.07× faster on irregular input.

Additional Key Words and Phrases: Data-parallelism, Parallel queries, monoids

1 INTRODUCTION

Monoids are a promising mathematical structure for parallel processing: associativity permits an input to be partitioned and reduced without fixing a particular evaluation order. This paper studies how small monoids can be assembled into *one-touch queries*: structured queries that inspect every input element once, construct no syntax tree or intermediate sequence, and are executed as a single data-parallel reduction.

The difficulty is composition. A realistic query may first parse digits into numbers, then aggregate numbers within delimiter-separated groups, and finally aggregate the group results. Parentheses add another form of composition in which a delimited subexpression is evaluated and injected as one value into an outer computation. We present a Futhark library in which monoids expose input generators, which map individual inputs to summaries, and result projections, which map summaries to query results. Generic combinators turn several such monoids into one monoid suitable for Futhark's `reduce` construct.

Our starting point is Kiselyov's *More Fun with Monoids* [11]. Kiselyov demonstrates the delimiter-based monoid composition represented in our library by the `chunk` combinator. He also shows how Horner's rule can be lifted to a monoid for constructing natural numbers from sequences of digits. We realise these constructions in Futhark and generalise the numeric construction to parse signed floating-point values, including decimal fractions. Around this core, we provide further generic operations for horizontal composition, input duplication, observation filtering, and left-biased alternative parsing. We also distinguish heterogeneous one-level nesting, where the monoids inside and outside parentheses may differ, from homogeneous nesting to a statically chosen depth below 50.

Our evaluation covers ARM64 CPU execution and a separate x86-64 CPU/GPU comparison. Besides representation costs, it exposes the importance of reduction granularity. For maximum-of-group-averages, a direct GPU reduction suffers from limited occupancy and substantial synchronisation overhead. Sequentially reducing contiguous blocks of 1024 bytes before combining

their summaries improves GPU performance without changing any monoid component. On the configured ThinkPad, for 100 MB inputs, this blocked query is 3.8–4.4× faster than handwritten sequential C and 1.6–2.2× faster than the corresponding four-worker multicore query. Both GPU and CPU implementations run on that ThinkPad. These are execution-only comparisons, excluding input loading and GPU transfers; smaller inputs can favour multicore execution. Applying the same strategy to one-level parenthesised sums of products with the restricted-decimal parser gives a 4.1× speedup over sequential C on the same configured ThinkPad for regular 100 MB input, but only 1.07× on irregular input. Thus the algebra permits different reduction layouts, but efficient execution still requires matching the layout to the hardware.

The library and its Rocq development are available at https://github.com/diku-dk/monoidal.

The paper makes four contributions:

- A Futhark library of generated and observable monoids, combining delimiter-based composition, numeric parsing and horizontal combinators.
- Paired-delimiter combinators for heterogeneous one-level and bounded homogeneous nesting, with module-inductive tuple storage.
- Mechanised monoid-preservation proofs in Rocq, including the tuple-backed nesting operator via an auxiliary array reference model.
- Structured-query case studies and CPU/GPU experiments demonstrating the effects of representation, parallel scaling and reduction granularity.

Section 2 motivates the design, and Sections 3–7 present the interface and combinators. Section 8 develops their algebraic laws; Sections 9–10 present case studies and evaluation. Sections 11–13 discuss related work, limitations, future work and conclusions.

2 MOTIVATING EXAMPLE: MAXIMUM AVERAGE

Consider a byte sequence containing floating-point numbers separated by commas, with groups separated by vertical bars:

```
12.1,45.3|10,93.2,2|23
```

The query computes the average within each group and returns the maximum group average. It therefore involves several distinct computations: parsing a number, maintaining a sum and count, completing an average at a comma or group boundary, and taking the maximum at a vertical bar.

For this example, `float` parses a sequence of bytes as one number; `sum`, `count`, and `max` provide the named aggregates; `prod` runs two monoids in parallel; and `dup` supplies the same input to both components of a product. The `chunk` combinator evaluates one monoid between delimiter bytes and injects each resulting value into an outer monoid. The `optone` monoid requires a non-empty input and selects its first injected value, and `with_gen` adapts the external byte representation to the input representation expected by the composite monoid.

The following Futhark module builds the query from these operations. Section 3 defines the mathematical interface underlying the code, and Sections 4 and 5 explain the combinators.

```
module max_avgs = {
  open monoidal
  module avg (X:numeric) = {
    open dup { type i = X.t } (prod (sum X) (count { type i = X.t }))
    type o = X.t
    def obs (t:t) : o =
      let (s,c) = obs t
      in if c == 0 then X.i64 0 else s X./ (X.i64 c)
  }
```

```
  module groups = chunk (max f64) (optone f64) { def del_ign (c:u8) = c == '|' }
  module items = chunk (avg f64) groups { def del_ign (c:u8) = c == ',' }
  module M = chunk (float f64) items { def del_ign (_:u8) = false }
  module T = with_gen M {
    type i = u8
    def gen (c:i) : M.i =
      if ('0' <= c && c <= '9') || c == '.' || c == '-'
      then #E c else #Del c
  }
  def red [n] (xs:[n]u8) : f64 =
    reduce T.op T.ne (map T.gen xs) |> T.obs
}
```

The construction is a hierarchy rather than an invocation of one primitive monoid. Nevertheless, the combinators produce one final monoid `T`, so the complete query is executed by one `map` and one `reduce`.

3 GENERATED AND OBSERVABLE MONOIDS

This section introduces an interface notation for describing how monoids are composed. A judgment $\mathcal{M} : I \Rightarrow O$ records only the input and output types exposed by a component, allowing the compatibility of a composite query to be checked without expanding its implementation. The higher-order signatures below consequently describe which monoids the combinators construct; they do not define the summary carriers, neutral elements, operations, generators, or projections of those combinators. Those definitions are given in Sections 4–6.

3.1 Basic notation

Definition 1 (Generated and observable monoid). *A generated and observable monoid is a tuple*

$$\mathcal{M} = (S, \varepsilon, \otimes, I, gen, O, obs),$$

where $(S, \varepsilon, \otimes)$ is a monoid, I is an input-element type, $gen : I \to S$ maps one input element to a summary, O is an output type, and $obs : S \to O$ projects a summary to an observable result.

We call S the *summary carrier*. Its values summarise arbitrary input fragments and can be combined independently of how those fragments were partitioned. The operation $\otimes$ is associative and ε is its two-sided identity. No homomorphism requirement is imposed on *gen* or *obs*.

The components correspond to the Futhark module fields as follows:

mathematics	S	ε	$\otimes$	I	gen	O	obs
Futhark	`t`	`ne`	`op`	`i`	`gen`	`o`	`obs`

After fixing this definition, we write

$$\mathcal{M} : I \Rightarrow O$$

to state that $\mathcal{M}$ has input type I and output type O. The symbol $\Rightarrow$ is an interface annotation, not a function arrow or the monoid operation. For $x_1, \ldots, x_n : I$, execution is defined by

$$run_{\mathcal{M}}(x_1, \ldots, x_n) = obs_{\mathcal{M}}\left(gen_{\mathcal{M}}(x_1) \otimes_{\mathcal{M}} \cdots \otimes_{\mathcal{M}} gen_{\mathcal{M}}(x_n)\right).$$

Consequently, execution forms a size-indexed family of functions

$$run_{\mathcal{M}} : \forall n \in \mathbb{N}.\ I^n \to O,$$

where I^n is the type of input sequences of length n. Equivalently, the size may be made explicit by writing $run^n_{\mathcal{M}} : I^n \to O$. For $n = 0$, the monoidal product in the definition of *run* is $\varepsilon_{\mathcal{M}}$. The

displayed type assumes that generation, combination, and observation are total; the next subsection states our convention for components that may fail. The corresponding Futhark size-polymorphic interface is `val run [n] : [n]i -> o`.

3.2 Implicit failure propagation

Some generated and observable monoids may reject malformed input. In the formal development we implicitly error-complete every summary and observation type:

$$S_\perp = S \uplus \{\perp\}, \qquad O_\perp = O \uplus \{\perp\}, \qquad \perp \otimes x = x \otimes \perp = \perp.$$

An invalid generator, combination, or observation produces $\perp$, which is then propagated strictly. Subsequent interface judgments omit this lifting for readability. Associativity is understood to include failure: whether a query produces $\perp$ must not depend on the reduction tree.

The formal rules continue to use implicit $\perp$ when the choice of concrete error representation is irrelevant. Sections 4 and 6 describe the concrete representations used by the library. Adding an absorbing failure value alone does not make a partial operation associative: rejection must agree under both associations. For an internal representation constrained by structural invariants, the mathematical carrier consists of the consistent representations together with failure. Generation and combination must preserve this carrier. Section 8 establishes these obligations separately from the meaning of observation.

3.3 Base monoids

We first use the notation to summarise the interfaces of the library's base monoids. For a numeric type X, and an arbitrary input type I, these include

$$\begin{array}{rcl@{\qquad}rcl} \mathsf{sum}_X & : & X \Rightarrow X, & \mathsf{mul}_X & : & X \Rightarrow X, \\ \mathsf{min}_X & : & X \Rightarrow X, & \mathsf{max}_X & : & X \Rightarrow X, \\ \mathsf{count}_I & : & I \Rightarrow \mathbb{Z}, & \mathsf{nat}_X & : & \mathsf{Byte} \Rightarrow X, \\ \mathsf{float}_X & : & \mathsf{Byte} \Rightarrow X, & \mathsf{all} & : & \mathsf{Bool} \Rightarrow \mathsf{Bool}, \\ \mathsf{exists} & : & \mathsf{Bool} \Rightarrow \mathsf{Bool}. & & & \end{array}$$

Here $\mathrm{Opt}(X) = X \uplus \{\mathsf{none}\}$ represents an optional value. The total interface of the option-valued floating-point parser is

$$\mathsf{float\_try}_X : \mathsf{Byte} \Rightarrow \mathrm{Opt}(X).$$

The interfaces expose only input and output types; the summary carriers of the numeric parsers, for example, contain additional state used to combine partial numbers.

3.4 Non-inductive combinators

Structured queries involve several components whose interfaces must agree. The following combinators compose a fixed collection of component monoids. We present a combinator as a curried higher-order module signature. Thus, an arrow between parenthesised interface judgments describes module application, whereas $\Rightarrow$ continues to describe the input and output interface of one generated and observable monoid. Ordinary function arguments retain the arrow $\rightarrow$.

Horizontal combinators construct new interfaces without introducing delimiters. Product composition has the signature

$$\mathsf{module\ prod} : (I \Rightarrow O) \rightarrow (J \Rightarrow R) \rightarrow ((I \times J) \Rightarrow (O \times R)).$$

If both components are to receive the same input, duplication converts a monoid over input pairs into a monoid over single inputs:

$$\mathsf{module\ dup} : ((I \times I) \Rightarrow O) \rightarrow (I \Rightarrow O).$$

It changes generation by mapping x to (x, x) and otherwise preserves the summary operation and observation. These two signatures explain the sum-and-count construction used by avg in Section 2.

Input and output adapters have the signatures

$$\text{module with\_gen} : (J \to I) \to (I \Rightarrow O) \to (J \Rightarrow O),$$
$$\text{module with\_obs} : (O \to R) \to (I \Rightarrow O) \to (I \Rightarrow R).$$

As a concrete derivation, let X be a numeric type, let $\iota_X : \mathbb{Z} \to X$ be integer conversion, and let $q_X : X \times \mathbb{Z} \to X$ be the observation function

$$q_X(s, c) = \begin{cases} 0, & c = 0, \\ s/\iota_X(c), & c \neq 0. \end{cases}$$

The average monoid used in Section 2 has the following derivation tree, whose internal nodes instantiate the higher-order signatures:

$$\dfrac{\dfrac{\dfrac{\text{sum}_X : X \Rightarrow X \qquad \text{count}_X : X \Rightarrow \mathbb{Z}}{\text{prod}(\text{sum}_X, \text{count}_X) : (X \times X) \Rightarrow (X \times \mathbb{Z})}}{\text{dup}\big(\text{prod}(\text{sum}_X, \text{count}_X)\big) : X \Rightarrow (X \times \mathbb{Z})} \qquad q_X : X \times \mathbb{Z} \to X}{\text{avg}_X = \text{with\_obs}_{q_X}\big(\text{dup}(\text{prod}(\text{sum}_X, \text{count}_X))\big) : X \Rightarrow X}.$$

The product monoid computes a sum and a count in parallel. Duplication changes its input from a pair (x, x) to a single x by duplicating during generation, and with_obs replaces the pair-valued observation (s, c) with the average $q_X(s, c)$. The underlying summary operation of avg_X remains the product of addition for the sum and addition for the count.

For a predicate $p : O \to \text{Bool}$, filtering preserves the interface:

$$\text{module filter} : (O \to \text{Bool}) \to (I \Rightarrow O) \to (I \Rightarrow O).$$

The library implements filtering at observation time: an observation that does not satisfy p is replaced by the observation of the neutral summary. When a filtered monoid is placed inside chunk, the predicate is therefore applied to each completed chunk rather than to each raw input element.

Alternative composition applies two monoids to the same input and chooses the leftmost successful observation:

$$\text{module or} : (I \Rightarrow \text{Opt}(O)) \to (I \Rightarrow \text{Opt}(O)) \to (I \Rightarrow \text{Opt}(O)).$$

Its summary contains both component summaries and combines them pointwise; the left-biased choice is made by observation. Thus the current combinator does not claim that a failed branch is eliminated during reduction.

3.5 Inductive combinators

An inductive combinator constructs a monoid that may itself be supplied as a component to another application of the same combinator. This permits query structures with an arbitrary statically specified number of aggregation levels. The delimiter-based chunk combinator has this form. Let

$$\mathcal{A} : I \Rightarrow O \qquad \text{and} \qquad \mathcal{B} : \text{Del}_{D \setminus \{d\}}(O) \Rightarrow R,$$

where $D \subseteq \text{Byte}$ is a finite set of delimiter bytes and $\text{Del}_D(X) = X \uplus D$. This index records the delimiters that may still occur at an interface; it does not denote an additional run-time tag. Let $d \in D$ be the delimiter byte for the current composition level—for example, $d =$ `','` when aggregating comma-separated values. The subscript in chunk_d records this byte: occurrences of

d separate values handled at this level, whereas delimiter bytes in $D \setminus \{d\}$ are propagated to the outer monoid. Delimiter-based composition has the signature

$$\mathsf{module\ chunk}_d : (I \Rightarrow O) \to (\mathsf{Del}_{D\setminus\{d\}}(O) \Rightarrow R) \to (\mathsf{Del}_D(I) \Rightarrow R), \qquad d \in D.$$

Thus, $\mathcal{A}$ processes the elements within a chunk, and its observations become inputs to $\mathcal{B}$. The composite itself is again a generated and observable monoid. Each application removes one byte from the set of delimiters still to be handled.

For multiple levels, we use descriptive calligraphic names for components and numeric subscripts only for levels. For example,

$$\begin{aligned}\mathcal{P} &: I_0 \Rightarrow O_0,\\ \mathcal{A}_1 &: \mathsf{Del}_{D\setminus\{d_1\}}(O_0) \Rightarrow O_1,\\ \mathcal{A}_2 &: \mathsf{Del}_{D\setminus\{d_1,d_2\}}(O_1) \Rightarrow O_2.\end{aligned}$$

Here $d_1, d_2 \in D$ are distinct, $\mathcal{P}$ may be a primitive parser, and $\mathcal{A}_1$ and $\mathcal{A}_2$ aggregate at successively outer structural levels. Repeated application of chunk turns the hierarchy into a single monoid from $\mathsf{Del}_D(I_0)$ to O_2. The construction avoids materialising the intermediate sequences of O_0 and O_1 values.

Paired-delimiter nesting comes in two forms. Let $D_1, D_2 \subseteq \mathsf{Byte}$ be disjoint sets recognised as opening and closing delimiters, respectively, and write $\mathsf{Nest}(I, O) = \mathsf{Raw}(I) \uplus \mathsf{Nested}(O)$. The heterogeneous, one-level combinator permits distinct monoids inside and outside parentheses:

$$\mathsf{module\ parens}^{D_1,D_2} : (I \Rightarrow O) \to (\mathsf{Nest}(I, O) \Rightarrow R) \to (\mathsf{Del}_{D_1\cup D_2}(I) \Rightarrow R).$$

A region-storage module $\mathcal{R}$ fixes a capacity of $2n + 1$ component summaries at a static depth $0 \le n < 50$. It is constructed inductively using the module combinators `Parens.Zero` and `Parens.Succ`: starting at depth zero, each successor adds one nesting level. Section 7 defines their interfaces. For repeated homogeneous nesting, the component must accept its own observed values as nested inputs:

$$\mathsf{module\ Parens.Make}^{D_1,D_2}_{\mathcal{R}} : (\mathsf{Nest}(I, O) \Rightarrow O) \to (\mathsf{Del}_{D_1\cup D_2}(I) \Rightarrow O), \qquad 0 \le n < 50.$$

Here $\mathcal{R}$ determines the maximum parenthesis depth n. The Futhark combinator is parameterised separately by the raw type I and value type O; no additional nesting-aware monoid interface is introduced.

Figure 1 summarises the combinator interfaces.

$$\begin{aligned}
\mathsf{module\ prod} &: (I \Rightarrow O) \to (J \Rightarrow R) \to ((I \times J) \Rightarrow (O \times R)),\\
\mathsf{module\ dup} &: ((I \times I) \Rightarrow O) \to (I \Rightarrow O),\\
\mathsf{module\ with_gen} &: (J \to I) \to (I \Rightarrow O) \to (J \Rightarrow O),\\
\mathsf{module\ with_obs} &: (O \to R) \to (I \Rightarrow O) \to (I \Rightarrow R),\\
\mathsf{module\ filter} &: (O \to \mathsf{Bool}) \to (I \Rightarrow O) \to (I \Rightarrow O),\\
\mathsf{module\ or} &: (I \Rightarrow \mathsf{Opt}(O)) \to (I \Rightarrow \mathsf{Opt}(O)) \to (I \Rightarrow \mathsf{Opt}(O)),\\
\mathsf{module\ chunk}_d &: (I \Rightarrow O) \to (\mathsf{Del}_{D\setminus\{d\}}(O) \Rightarrow R) \to (\mathsf{Del}_D(I) \Rightarrow R), \quad d \in D,\\
\mathsf{module\ parens}^{D_1,D_2} &: (I \Rightarrow O) \to (\mathsf{Nest}(I, O) \Rightarrow R) \to (\mathsf{Del}_{D_1\cup D_2}(I) \Rightarrow R),\\
\mathsf{module\ Parens.Make}^{D_1,D_2}_{\mathcal{R}} &: (\mathsf{Nest}(I, O) \Rightarrow O) \to (\mathsf{Del}_{D_1\cup D_2}(I) \Rightarrow O).
\end{aligned}$$

Fig. 1. Module signatures for the non-inductive and inductive combinators.

4 PRIMITIVE MONOIDS AND HORIZONTAL COMBINATORS

This section describes the library components that do not introduce structural delimiters. The primitive monoids provide common folds and numeric parsers; the horizontal combinators change inputs and observations or run several monoids over the same sequence. All the constructions implement the interface of Section 3, so they remain suitable for a final parallel `reduce`.

4.1 Primitive reductions

The modules `sum`, `mul`, `min`, and `max` are instances of one construction in which the summary, input, and output types coincide. Both generation and observation are the identity; only the neutral element and operation vary. For a numeric type X, the implementations are summarised by

	sum_X	mul_X	min_X	max_X
ε	0	1	$highest_X$	$lowest_X$
$\otimes$	+	$\times$	min	max

where `highest` and `lowest` are supplied by Futhark's `numeric` module type. The Boolean modules `all` and `exists` follow the same pattern, using $(\mathsf{true}, \wedge)$ and $(\mathsf{false}, \vee)$, respectively.

Counting differs only in generation. Its carrier and output are integers, its operation is addition, and every input element generates one:

$$S = \mathbb{Z}, \qquad \varepsilon = 0, \qquad x \otimes y = x + y, \qquad gen(\_) = 1, \qquad obs(x) = x.$$

The input type is otherwise unconstrained. Consequently, `count` can be paired with any monoid that consumes the same logical values.

4.2 Numbers as monoidal parses

The `nat` module implements Horner composition for decimal digits. A summary (v, p) represents a digit fragment with value v and positional factor $p = 10^k$, where k is the fragment length. A digit c generates $(\delta(c), 10)$, and adjacent fragments compose as

$$(v_1, p_1) \otimes (v_2, p_2) = (v_1 p_2 + v_2, p_1 p_2).$$

The neutral summary is $(0, 1)$ and observation returns v. This operation is associative because it is precisely concatenation represented by an affine map. The generator assumes that its input byte is a decimal digit; lexical classification is performed by the surrounding query.

Floating-point syntax requires summaries for fragments that may begin or end in the middle of a number. Internally, `float0` uses the raw carrier

$$S_0 = \mathsf{I}(neg, v, p) \mid \mathsf{F}(neg, v, p, f) \mid \mathsf{P} \mid \mathsf{M} \mid \mathsf{NE}.$$

Here I represents a digit fragment without a decimal point, F a fragment containing one, P a point in isolation, M a minus sign in isolation, and NE the neutral summary. The Boolean component records a leading minus sign. As for `nat`, p records the positional factor of an integer fragment; in F, f is 10^{-k} after k fractional digits have been accumulated. Representative cases are

$$\mathsf{I}(s, x, b_1) \otimes \mathsf{I}(\mathsf{false}, y, b_2) = \mathsf{I}(s, xb_2 + y, b_1 b_2),$$
$$\mathsf{F}(s, x, b, f) \otimes \mathsf{I}(\mathsf{false}, y, b_2) = \mathsf{F}(s, x + y(f/b_2), b, f/b_2),$$
$$\mathsf{I}(s, x, b) \otimes \mathsf{P} = \mathsf{F}(s, x, b, 1).$$

The remaining cases handle a point or minus sign at a partition boundary and reject combinations containing, for example, two decimal points or a minus sign away from the beginning. The accepted syntax includes leading and trailing zeroes, numbers such as `.5` and `5.`, and the library's deliberately

permissive form `-.`, which observes as zero. Exponents and leading plus signs are not part of the current grammar.

Parser failure is factored out of this state machine. The internal `failure` module supplies a type constructor `result`, successful and failing injections, and unary and binary binding operations:

```
module type failure = {
  type result 'a
  val ok    'a : a → result a
  val fail  'a : bool → a → result a
  val bind  'a 'b : result a → (a → result b) → result b
  val bind2 'a 'b 'c : (a → b → result c) →
                       result a → result b → result c
}
```

The actual carrier is `Fail.result raw_t`. Thus failure is propagated through combination without duplicating the parsing cases. Instantiating `float0` with `assert_failure` yields `float`, whose invalid cases abort by assertion and whose observation type is X. Instantiation with `option_failure` yields `float_try`, whose observation type is $\mathrm{Opt}(X)$ and whose invalid cases become none. This realises in code the two failure views discussed in Section 3.2: assertions keep ordinary pipelines simple, while explicit options permit failure to participate in composition.

For workloads restricted to short decimal literals, `decimal` accepts an optional minus sign, at least one digit, and an optional fractional part consisting of a point followed by at least one digit. It permits at most 18 digits in total and no exponent or leading plus. Its summary accumulates an integer mantissa and decimal-position information, converting to the output numeric type at observation. Section 10.4 evaluates this specialisation against `float`.

4.3 Adapters and products

The `prod` combinator is the direct product construction. For component summaries S_1 and S_2, it uses carrier $S_1 \times S_2$, combines both coordinates independently, and applies generation and observation componentwise. It therefore runs two reductions in lockstep without constructing an intermediate sequence:

$$(s_1, s_2) \otimes (t_1, t_2) = (s_1 \otimes_1 t_1, s_2 \otimes_2 t_2).$$

The adapter combinators preserve the carrier, neutral element, and operation of their component. Given $g : J \to I$, `with_gen` replaces generation by $gen \circ g$; given $q : O \to R$, `with_obs` replaces observation by $q \circ obs$. Since neither adapter changes summary combination, both inherit associativity from the component monoid.

Duplication is the input-adapter specialisation $g(x) = (x, x)$. Hence `dup` turns the product of `sum` and `count` into a monoid that accepts a single numeric value and produces its sum and cardinality. The `avg` module in Section 2 then changes only observation, mapping (s, c) to s/c when $c \neq 0$ and to zero otherwise. No averages are computed while partial summaries are combined; reduction maintains only the pair (s, c).

Filtering is the corresponding observation adapter. For a predicate $p : O \to \mathrm{Bool}$, it observes a summary s as

$$obs_{\mathrm{filter}^p(\mathcal{M})}(s) = \begin{cases} obs_{\mathcal{M}}(s), & p(obs_{\mathcal{M}}(s)), \\ obs_{\mathcal{M}}(\varepsilon), & \text{otherwise.} \end{cases}$$

Thus `filter` does not discard raw input elements. It accepts or replaces the completed observation, which is why placing it inside `chunk` filters whole parsed chunks.

4.4 Non-empty, optional, and alternative results

The helper `one` has carrier $\mathrm{Opt}(X)$ and uses left-biased choice, with `none` as its neutral element. Generation wraps an input in `some`, while observation asserts that the result is present. Despite its name, the current implementation does not reject a second input: it retains the first. The derived `optone` module additionally adapts inputs of type `del X`; an element is unwrapped, whereas observing a delimiter at this stage triggers an assertion. These modules are used at the outer boundary of examples where a non-empty singleton result is expected.

The `opt` combinator gives a component optional input and output. Its carrier is $\mathrm{Opt}(S)$ and its operation is again left-biased choice. A present input generates a present component summary, an absent input generates `none`, and observation maps the component observation over a present summary. It therefore selects the first present contribution; it does not combine several present component summaries.

Finally, `or` evaluates two option-valued monoids over the same input. Its carrier is their product and all generation and combination work is performed componentwise. Observation first inspects the left result, returns it when present, and otherwise inspects the right result. Consequently, `or` provides left-biased alternative parsing, but it does not short circuit during generation or reduction: both alternatives are maintained until the final observation. This design is possible precisely because `float_try` exposes failure as data; an assertion-based parser cannot serve as a recoverable branch of `or`.

5 DELIMITER-BASED VERTICAL COMPOSITION

The `chunk` combinator is our Futhark realisation of Kiselyov's vertical composition of observable monoids [11]. We give only the representation needed to connect his construction to our interface; Kiselyov gives the detailed derivation, laws, and further examples.

Consider an inner monoid $\mathcal{A} : I \Rightarrow O$, an outer monoid $\mathcal{B} : \mathrm{Del}_{D\setminus\{d\}}(O) \Rightarrow R$, and the adapter

$$lift(s) = gen_{\mathcal{B}}(\mathsf{E}(obs_{\mathcal{A}}(s))).$$

As in Section 3.5, the delimiter d is consumed at this level, while other members of D are passed to $\mathcal{B}$. The Futhark type `del I` implements the erased representation of $\mathrm{Del}_D(I)$, using `#E` for an element and `#Del` for a byte. The supplied `del_ign` predicate recognises d.

The composite summary has two forms:

$$S = \mathsf{A}(S_{\mathcal{A}}) \;\mid\; \mathsf{C}(S_{\mathcal{A}}, S_{\mathcal{B}}, S_{\mathcal{A}}).$$

A summary $\mathsf{A}(s)$ contains no delimiter propagated to the outer level. A summary $\mathsf{C}(l, c, r)$ retains the unfinished inner summaries at its two boundaries and an outer summary for the completed material between them. This constant-size boundary representation is what permits independently reduced partitions to be joined without materialising their chunks.

The only non-componentwise case is the combination of two summaries that both contain propagated delimiters:

$$\begin{aligned}&\mathsf{C}(l_x, c_x, r_x) \otimes \mathsf{C}(l_y, c_y, r_y)\\ &\quad = \mathsf{C}\big(l_x, c_x \otimes_{\mathcal{B}} lift(r_x \otimes_{\mathcal{A}} l_y) \otimes_{\mathcal{B}} c_y, r_y\big).\end{aligned}$$

The adjacent boundary fragments r_x and l_y are first joined in $\mathcal{A}$; their observation is then generated as one input to $\mathcal{B}$. The other three combinations merely extend the appropriate boundary summary. Observation similarly lifts the remaining boundary: it maps $\mathsf{A}(s)$ to $obs_{\mathcal{B}}(lift(s))$ and $\mathsf{C}(l, c, r)$ to the observation of $lift(l) \otimes_{\mathcal{B}} c \otimes_{\mathcal{B}} lift(r)$.

Repeated applications consume one delimiter at a time and expose another generated-and-observable monoid. In the maximum-average example of Section 2, the innermost application

converts byte fragments to floating-point values, the next consumes commas while maintaining a group average, and the outer application consumes vertical bars while maintaining the maximum. The result is nevertheless one monoid and hence one final parallel reduction. Apart from the set index on Del_D, which makes delimiter propagation explicit, this construction follows Kiselyov's `chunk` monoid.

6 PAIRED-DELIMITER NESTING

Parentheses differ from the separators of Section 5: they do not merely end one aggregate and begin another, but change how the enclosed input is interpreted. Moreover, an opening parenthesis and its closing parenthesis may occur in different input partitions. A parallel summary must therefore retain enough boundary information to match them when independently reduced partitions are combined.

6.1 Injecting a nested result

Both nesting combinators use the input type

$$\mathsf{Nest}(I, O) = \mathsf{Raw}(I) \uplus \mathsf{Nested}(O).$$

A raw input continues the current parse, whereas a nested input is the observed result of a completed parenthesised region and must be treated as one logical value. The `nestable` adapter equips any monoid $\mathcal{M} : I \Rightarrow O$ with this interface. Its summary carrier is

$$\mathsf{NE} \mid \mathsf{M}(S_{\mathcal{M}}) \mid \mathsf{V}(O).$$

Raw inputs generate $\mathsf{M}(gen_{\mathcal{M}}(x))$, nested values generate $\mathsf{V}(v)$, and observation either observes the stored component summary or returns the stored value directly. Two M summaries combine using $\mathcal{M}$; a V summary cannot be combined directly with another non-neutral summary. In the expression examples below, `chunk` ensures that raw numeric fragments and injected nested values reach `nestable` as separate logical numbers.

For the remainder of this section, let D_1 and D_2 be the disjoint sets of opening and closing delimiter bytes introduced in Section 3.5. The concrete examples use $D_1 = \{\texttt{'('}\}$ and $D_2 = \{\texttt{')'}\}$.

6.2 Heterogeneous one-level nesting

The combinator $\mathsf{parens}^{D_1,D_2}$ takes an inner monoid $\mathcal{M}_1 : I \Rightarrow O$ and an outer monoid $\mathcal{M}_2 : \mathsf{Nest}(I, O) \Rightarrow R$. Define the operation that closes an inner summary by

$$close(s) = gen_{\mathcal{M}_2}(\mathsf{Nested}(obs_{\mathcal{M}_1}(s))).$$

For a fragment with relative prefix depths $h_0, \ldots, h_k$, define $lo = \min_j h_j$, $hi = \max_j h_j$, and $net = h_k$. The compact summary uses five forms, whose tags determine these depth measures:

$$\begin{aligned}
\mathsf{A}(s_i, s_o) &: \text{No parenthesis boundary,}\\
\mathsf{Closed}(s_o) &: \text{A completed parenthesised region,}\\
\mathsf{L}(s_o, s_i) &: \text{An unmatched opening boundary,}\\
\mathsf{R}(s_i, s_o) &: \text{An unmatched closing boundary,}\\
\mathsf{B}(s_l, s_o, s_r) &: \text{An unmatched closing followed by an opening boundary.}
\end{aligned}$$

The subscripts i and o distinguish summaries of $\mathcal{M}_1$ and $\mathcal{M}_2$. An A summary contains both interpretations because a fragment without a parenthesis may later be combined either inside or outside a pair. In L, the prefix is outside and the suffix is inside; in R, the prefix is inside and the suffix is outside. The B form contains an inner prefix, an outer middle, and an inner suffix. The neutral summary is $\mathsf{A}(\varepsilon_1, \varepsilon_2)$. The respective depth triples for A, Closed, L, R, and B are $(0, 0, 0)$, $(0, 1, 0)$, $(0, 1, 1)$, $(-1, 0, -1)$, and $(-1, 0, 0)$. They need not be stored or reconstructed during combination:

the constructor pair determines whether the concatenation is viable. The carrier adds a separate Failure constructor to these five forms, rather than an outer option tag. To characterise the rejected pairs mathematically, depth metadata composes as

$$\begin{aligned} lo_{ab} &= \min(lo_a, net_a + lo_b), \\ hi_{ab} &= \max(hi_a, net_a + hi_b), \\ net_{ab} &= net_a + net_b. \end{aligned}$$

The result is Failure when either argument is invalid or $hi_{ab} - lo_{ab} > 1$; invalidity is absorbing. Crucially, matching boundaries produces Closed, retaining depth history without an unused inner summary. Thus `()` has depth $(0, 1, 0)$ and cannot subsequently be mistaken for a fragment that never entered a nested region.

Generation establishes this invariant locally:

$$\begin{aligned} \mathsf{Raw}(x) &\mapsto \mathsf{A}(gen_1(x), gen_2(\mathsf{Raw}(x))), \\ d_1 \in D_1 &\mapsto \mathsf{L}(\varepsilon_2, \varepsilon_1), \\ d_2 \in D_2 &\mapsto \mathsf{R}(\varepsilon_1, \varepsilon_2). \end{aligned}$$

Table 1 gives the result form for every pair of summary forms. A dash denotes a form pair whose concatenation necessarily has depth span greater than one. The combination table rejects precisely these pairs without computing a separate depth check; the other form combinations preserve the boundary form.

Table 1. Result forms for the one-level parenthesis operation.

⊗	A	Closed	L	R	B
A	A	Closed	L	R	B
Closed	Closed	Closed	L	–	–
L	L	–	–	Closed	L
R	R	R	B	–	–
B	B	–	–	R	B

One central case is $\mathsf{L} \otimes \mathsf{R}$, where an opening boundary in the left partition is matched by a closing boundary in the right partition:

$$\begin{aligned} &\mathsf{L}(x_o, x_i) \otimes \mathsf{R}(y_i, y_o) \\ &\quad = \mathsf{Closed}\big(x_o \otimes_2 close(x_i \otimes_1 y_i) \otimes_2 y_o\big). \end{aligned}$$

The two inner boundary summaries are combined, observed, and generated as one nested input to the outer monoid. The $\mathsf{B} \otimes \mathsf{B}$ case similarly closes the right unmatched opening of the left fragment against the left unmatched closing of the right fragment, while retaining the two outer boundaries. The remaining defined cases extend a boundary or carry an unmatched pair across the partition join. Observation succeeds only for $\mathsf{A}(-, s_o)$ or $\mathsf{Closed}(s_o)$ and returns $obs_{\mathcal{M}_2}(s_o)$; another form denotes globally unbalanced parentheses and triggers an assertion.

This construction deliberately permits $\mathcal{M}_1$ and $\mathcal{M}_2$ to be different. That generality is useful when a single parenthesised layer has a specialised interpretation. Its finite five-form representation is small, but the depth-span check also explains why simply applying the combinator again is not a representation of arbitrary nesting.

When the same computation applies inside and outside parentheses, `parens_hom` specialises depth-one nesting to a single component $\mathcal{M} : \mathsf{Nest}(I, O) \Rightarrow O$. Each raw region then needs only

one component summary. It retains the constructor-based structural cases and absorbing failure, observing a completed inner region and regenerating its value as a nested operand. Its summary has no separately stored depth counters.

6.3 Homogeneous nesting to depth n

The bounded combinator `Parens.Make` supports repeated nesting to a depth $0 \le n < 50$, fixed by its region-storage module, by requiring one component

$$\mathcal{M} : \mathsf{Nest}(I, O) \Rightarrow O$$

at every level. After parentheses that match within a fragment have been closed, every fragment has the normal form

$$r_0\ )\ r_1\ \cdots\ )\ r_c\ (\ r_{c+1}\ \cdots\ (\ r_{c+o},$$

where c is the number of unmatched closing parentheses, o is the number of unmatched opening parentheses, and each r_j is a summary of $\mathcal{M}$. The implementation stores $2n+1$ component regions in a fixed-capacity tuple, together with three depth measures and a validity flag. Section 7 describes the storage interface. The Rocq model represents invalidity using an outer option instead. The counts are derived as $c = -lo$ and $o = net - lo$ rather than stored separately. For a fragment whose prefix depths are $h_0, \ldots, h_k$, these measures denote

$$lo = \min_j h_j, \quad hi = \max_j h_j, \quad net = h_k.$$

A viable fragment may contain as many as n unmatched closings followed by n unmatched openings, hence the $2n+1$ region slots. The region-storage module fixes n, so the summary structure does not grow with input length.

When summaries a and b are combined, their depth metadata compose as

$$\begin{aligned} lo_{ab} &= \min(lo_a, net_a + lo_b), \\ hi_{ab} &= \max(hi_a, net_a + hi_b), \\ net_{ab} &= net_a + net_b. \end{aligned}$$

The result is marked invalid if either operand is invalid or $hi_{ab} - lo_{ab} > n$. Otherwise, the first region of b is appended to the last region of a. The operation then replays the unmatched closings of b, matching them against unmatched openings of a where possible, and finally appends the unmatched openings of b. Whenever a pair is matched, its inner region is observed and regenerated as $\mathsf{Nested}(v)$ in the surrounding region. Thus replay implements the same close-and-inject step as the one-level combinator.

Final observation additionally requires

$$lo \ge 0 \wedge hi \le n \wedge net = 0 \wedge c = 0 \wedge o = 0.$$

For a non-failing summary, these conditions express balanced input whose absolute nesting depth remains between zero and n. Thus n is the maximum number of simultaneously open parentheses: an unparenthesised expression has depth zero, (x) has depth one, and $((x))$ has depth two. A violation is reported by assertion in the current implementation, corresponding to the implicit failure described in Section 3.2.

6.4 Parenthesised sum of products

The case study begins with the unparenthesised expression monoid `Expr`, built by composing `float`, multiplication, and addition with `chunk`. The outer component accepts either the raw input expected by `Expr` or an already evaluated floating-point value. At its numeric layer it uses `nestable (float f64)`, so a nested result is treated as one factor rather than reparsed as bytes. A small

`with_gen` adapter routes arithmetic delimiters to `chunk` and raw or nested numeric input to this numeric layer.

The one-level instantiation uses `parens Expr Outer`. For example,

$$\texttt{2*(3+4)+5} \longmapsto 19, \quad \texttt{(12+34)*(56+78)} \longmapsto 6164.$$

Here the observation of `3+4` is injected as a single factor into the outer product. The bounded variants reuse `Outer` at every level. For depth two, the library instantiates `Parens.Make` with raw type `Expr.i`, value type `f64`, component `Outer`, a region module built by two `Parens.Succ` applications to `Parens.Zero`, and the predicates recognising `(` and `)`. This instance evaluates, for example, `2*(3+(4*5))` = 46 and `12*(34+56*(2+3))+78` = 3846. Adding a third successor handles a third level without changing the component monoid.

The library retains both combinators because neither subsumes the other's useful representation. The one-level version admits different inner and outer monoids and has a compact five-form summary. The bounded homogeneous version supports several levels, but requires a component closed under nested observations and uses $2n + 1$ summary slots. Tests exercise every two-way split of representative expressions; the depth-two suite also compares both associations at every three-way split. These tests provide evidence that results do not depend on reduction partitioning, while the algebraic statement and its validity domain are deferred to Section 8.

7 MODULE-INDUCTIVE TUPLE STORAGE: THE PARENS MODULE

The `Parens` module implements bounded homogeneous nesting using module-inductive, fixed-capacity tuples. It accesses the regions through head replacement, shifts and reversal. The specialised one-level `parens` and `parens_hom` combinators remain available; no additional nesting-aware monoid interface is needed.

7.1 Region modules and the public interface

A region module stores elements of type `elem`. Its carrier `t` has capacity $2n + 1$, where `depth` supplies the static bound n. The operation `fill` initialises all slots; `top` and `last` project the first and last slots; `replace` changes the first slot; `push` prepends an element and discards the last slot; `pop` removes the first slot and appends a padding element; and `reverse` reverses the slots. These operations have the following Futhark interface.

```
module type parens_regions = {
  type elem
  type t
  val depth : i64
  val fill : elem → t
  val top : t → elem
  val last : t → elem
  val replace : t → elem → t
  val push : t → elem → t
  val pop : t → elem → t
  val reverse : t → t
}
```

`Parens.Zero E` uses the single-slot carrier `E.t` and depth zero. Given a region module R with element type S, `Parens.Succ R` has carrier $S \times R.\texttt{t} \times S$ and depth $R.\texttt{depth} + 1$. Writing its value as

(a, r, z), its shifts and reversal are defined by

$$\begin{aligned} push(a, r, z)\ x &= (x, R.push\ r\ a, R.last\ r), \\ pop(a, r, z)\ p &= (R.top\ r, R.pop\ r\ z, p), \\ reverse(a, r, z) &= (z, R.reverse\ r, a). \end{aligned}$$

Filling and head replacement follow the tuple structure. Thus each successor adds two slots without introducing integer indexing. Futhark specialises the module applications and eliminates the tuple structure; nested tuples do not introduce pointer-linked runtime objects.

The constructor `Parens.Make X M R D` takes raw and value types `X`, a monoid `M` with input `nest X.raw X.value` and output `X.value`, region storage `R` with element type `M.t`, and delimiter predicates `D.left` and `D.right`. Its result has input `del X.raw` and output `X.value`. For example, given an expression monoid `M` accepting raw delimiter-tagged bytes and nested floating-point values, depth two is constructed as follows.

```
module R = {
  local open monoidal.Parens
  open Succ (Succ (Zero {type t = M.t}))
}
module P = monoidal.Parens.Make
  {type raw = del u8 type value = f64} M R
  {def left (c:u8) = c == '('
   def right (c:u8) = c == ')'}
```

Further successors construct depths three and five in the same way. `Make` obtains the depth from `R`; the user does not also supply a numeric bound. Custom region modules must satisfy the fixed-capacity sequence laws above, not merely implement the types in the signature.

7.2 Combination and compact metadata

Let $k = c + o$ be the active-region index from Section 6. Instead of storing $(r_0, \ldots, r_k)$ in forward order, the tuple stores

$$(r_k, r_{k-1}, \ldots, r_0, \underbrace{\varepsilon_{\mathcal{M}}, \ldots, \varepsilon_{\mathcal{M}}}_{2n-k}).$$

The head is therefore always the active region. Appending a region updates the head using the component operation. Opening a parenthesis pushes a neutral region. Closing a matched parenthesis observes the head, pops it, and combines the regenerated nested value with the new head. An unmatched closing parenthesis pushes a new neutral region instead.

To combine two fragments, `Make` first checks their combined depth metadata using the equations of Section 6. It reverses the right-hand tuple and removes its leading padding to expose r_0. The remaining regions are consumed as a queue while replaying unmatched closings and then unmatched openings. The final summary receives the combined *lo*, *hi* and *net*. Tuple projections replace dynamic slot selection; the replay counts and loops remain dynamic. This representation change does not by itself guarantee less work or lower execution time.

The implementation supports $0 \le n < 50$. It stores a validity flag and three signed-byte depth fields alongside the tuple; counts and replay arithmetic also use signed bytes. For consistent operands, both depth ranges contain zero and their own net depth. After shifting the second range by the first net depth, the ranges overlap, so the combined span is at most $2n$. Its extrema and net depth lie in $[-2n, 2n]$, so this metadata arithmetic fits in signed bytes even for a rejected combination. The active count is bounded by $2n \le 98$ for consistent summaries; replay counts are used only after the combined depth check succeeds. No such active-count bound is claimed for

rejected combined metadata. The three stored counters consequently require three bytes of payload rather than 24 bytes for three `i64` fields; this is not a claim about padding in the complete generated representation. The replay invariants in the next section justify the intermediate bounds and ensure that a push discards only neutral padding. No special depth-one or depth-two region constructor is required. GPU performance of this bounded-nesting representation remains unmeasured.

8 ALGEBRAIC MONOID LAWS AND CORRECTNESS

The Rocq development in the library's `proof` directory checks the algebraic laws of the summary operations. It comprises 12,232 lines across 27 Rocq source files, including comments and blank lines. The array-based bounded-nesting development accounts for 7,659 lines; the region modules, tuple model, correspondence proof and auxiliary direct lemmas add 2,813 lines. Its central question is whether a combinator constructs a monoid when its arguments are monoids. This is separate from proving that observation implements a specified sequential query. We describe the checked results and their scope below.

8.1 Interface and failure model

The record `MonoidalOps`, defined in `Monoid.v`, contains the summary carrier, neutral element, operation, generator, and observation. It deliberately contains no proofs. The separate predicate `IsMonoid` states associativity and both identity laws over every element of that carrier. This separation mirrors Futhark: implementing the module interface supplies operations but does not establish their laws.

Observation has type $S \to \mathrm{Opt}(O)$ in Rocq. We use None for failure and Some(v) for success, making the implicit $\bot$ of Section 3.2 explicit. When generation or combination can fail, the summary carrier also has absorbing failure, represented by an option or a dedicated failure constructor. Successful absence must remain distinct from failure: for `opt`, observation can return Some(None); for `optone`, the neutral summary is Some(None), whereas outer None denotes failure. No homomorphism or totality law is imposed on a component's observation.

8.2 Primitive and compositional results

The files `arith.v`, `all.v`, `Exists.v`, and `count.v` establish the primitive monoid laws. The arithmetic instances use exact integers for sum, multiplication, and counting; extrema use integers extended with explicit least and greatest elements. The counting development additionally relates reduction to input length. These are mathematical arithmetic models, not proofs about fixed-width overflow or IEEE floating-point operations.

For `nat`, `nat.v` abstracts over the arithmetic identities needed by Horner composition. Associativity expands to

$$((x, b) \otimes (y, c)) \otimes (z, d) = (xcd + yd + z, bcd) = (x, b) \otimes ((y, c) \otimes (z, d)).$$

The exact-integer instance discharges those identities. The corrected `float` case table is modelled with exact canonical rationals in `float.v`. Its invariant requires nonzero decimal scales; generation and combination preserve this invariant. The theorem `float_is_monoid` packages the consistent optional summaries as a monoid. It does not assert bitwise reduction-tree independence for IEEE arithmetic, nor does it constitute a complete lexical correctness proof.

Separate files prove preservation of `IsMonoid` for `prod`, `dup`, `with_gen`, `with_obs`, `filter`, and `nestable`. Products combine componentwise; adapters leave the summary algebra unchanged. Filtering also preserves the laws when observing the neutral element fails. The first-present operation of `opt` is associative independently of any payload monoid laws. The `optone` development additionally models delimiter rejection as absorbing failure.

The theorem `chunk_is_monoid` in `chunk.v` checks the delimiter-based construction of Section 5, including failure during inner observation. Its proof analyses the summary cases directly; it needs only the two component monoid laws. Thus the mechanised result complements Kiselyov's presentation [11]. These preservation theorems can be composed along a query's construction, subject to the carrier and arithmetic assumptions of each instance. We do not claim mechanised coverage of every library combinator; in particular, `or` and the restricted `decimal` parser used in the evaluation have no separate Rocq developments at present.

8.3 One-level nesting

The compact representation of Section 6 is modelled directly in `Parens.v`. The type `ParensForm` has constructors `Failure`, `A`, `Closed`, `L`, `R`, and `B` and is itself the carrier. The operation `parens_op` uses the combination table directly. The proof-only function `form_depth` reconstructs depth metadata from the tag; `parens_op_guarded_equivalent` proves that an explicit depth-span check would not change the result.

Theorem 1 (Mechanised one-level monoid law). *If $\mathcal{M}_1$ and $\mathcal{M}_2$ satisfy* `IsMonoid`*, then the Rocq construction* `parens` $\mathcal{M}_1$ $\mathcal{M}_2$ *satisfies* `IsMonoid`*.*

The checked theorem is `parens_is_monoid`. Its associativity proof analyses the constructor triples, using component associativity for surviving regions and explicit cases for failed observations. Both identities follow from the component identities. All forms, including failure, belong to the carrier: no reachability invariant, canonical input sequence, or assumed normalisation theorem is needed. The related `parens_del` interface adds configurable delimiter predicates without changing the summary algebra.

The representation matters to this result. An earlier representation kept an unused inner field after closing a region. For an atom followed by `()`, two associations could retain different inner fields while producing equal observations. The `Closed` constructor removes that field, making equality of complete summaries possible. The development includes the former counterexample as a passing equality regression.

In `parens_hom.v`, homogeneous depth-one nesting is proved to be a monoid whenever its single component is. The proof uses the direct constructor-based operation, including its failure case.

8.4 Array reference model for bounded nesting

The file `Parens_n.v` is retained as an auxiliary array reference model for the proof of `Parens.Make`, not as a public Futhark combinator. `PullArray` represents an array by a size and an indexing function. For bound n, consistency requires $2n + 1$ region slots,

$$lo \le 0 \le hi, \qquad lo \le net \le hi, \qquad hi - lo \le n,$$

a valid active-region index $c + o < 2n + 1$, and neutral padding beyond that index, where $c = -lo$ and $o = net - lo$. The lifted consistency predicate also admits failure. These are structural conditions, assigning no parsing meaning to the regions.

The checked array development establishes consistency preservation, associativity and two-sided identity on this consistent optional carrier, assuming the component is a monoid. Boundary-cancellation lemmas equate the two replay orders; separate arguments establish propagation of exceeded depth bounds and failed inner observations. The associativity theorem does not assume total observation or an interpretation by canonical input sequences.

8.5 Tuple-backed bounded nesting

The files `ParensRegions.v`, `ParensTuple.v` and `ParensTupleProof.v` cover the module of Section 7. The first defines a proof-only sequence interpretation $view_{\mathcal{R}}$ of a region module's carrier. The predicate `Laws` requires this interpretation to be injective, of length $2n+1$, and to commute with filling, projections, replacement, push, pop and reversal. The lemmas `Zero_laws` and `Succ_laws` establish these laws inductively for the public constructors. Thus arbitrary user-defined storage requires a proof of the same laws, whereas a chain of successors has them by construction.

Write S_{tuple} for the direct model's record of depth metadata and a region tuple; failure is represented by an outer option. Its consistency predicate has the same depth inequalities as the array model, but requires neutral padding after the occupied reversed prefix. Let $k = c + o$ and let q be the tuple in a consistent summary s. Define $\rho(s)$ to keep the depth metadata and have an array of size $2n + 1$ with entries

$$\rho(s)[i] = \begin{cases} view_{\mathcal{R}}(q)[k-i], & 0 \le i \le k, \\ \varepsilon_{\mathcal{M}}, & i > k. \end{cases}$$

Extend ρ to options by mapping failure to failure. Neutral padding and injectivity of $view_{\mathcal{R}}$ make ρ injective on consistent optional summaries. The correspondence file checks consistency with `encode_consistent` and injectivity with `encode_opt_injective`.

The main representation argument proves that head updates, opening and closing steps, queue initialisation, and both replay loops agree under ρ. An intermediate state is kept within a fixed depth envelope $[L, H]$ satisfying $H - L \le n$. Closing requires its next net depth to be at least L; opening requires its next net depth to be at most H. The envelope lemmas preserve consistency and the bounds needed by tuple shifts. The final steps relate the depth guards and installation of combined metadata. In particular, `encode_op` establishes

$$\rho(a \otimes_{\text{tuple}} b) = \rho(a) \otimes_{\text{array}} \rho(b)$$

for consistent operands and $n < 50$. This is a proved correspondence of the direct operators, not a definition replacing tuple replay by array replay. No canonical input sequence or parsing-correctness assumption is used.

By this equality, the array associativity theorem, and injectivity of ρ, the two associations of the tuple operation are equal. The checked results are `op_associative`, `op_preserves_consistency`, both neutral laws, and `gen_consistent`. To express the result with the same `IsMonoid` predicate as the primitive constructions, define the carrier to be

$$\widehat{S} = \{s : \mathsf{Opt}(S_{\text{tuple}}) \mid \mathsf{ConsistentOpt}(s)\}.$$

The packaged `TupleProof.Make` uses the direct operation, generator and observation, with consistency proofs attached to summaries.

Theorem 2 (Mechanised tuple-nesting monoid law). *Let $\mathcal{R}$ satisfy the region laws at depth $0 \le n < 50$, with element type $S_{\mathcal{M}}$. If $\mathcal{M}$ satisfies* `IsMonoid`*, then* `TupleProof.Make`*, instantiated with $\mathcal{M}$, $\mathcal{R}$ and the delimiter predicates, satisfies* `IsMonoid` *on $\widehat{S}$.*

The theorem is `Make_is_monoid`. Failure belongs to $\widehat{S}$, and component observation need not be total or a homomorphism. The abstract Futhark result signature hides the internal tuple summary; the subtype is proof-only and does not add runtime checks or fields.

8.6 Consequences and verification boundary

For a monoid carrier, let $\mathrm{fold}_T(w)$ denote the product of the generated summaries of w along a binary tree T preserving input order. Associativity implies

$$\mathrm{fold}_{T_1}(w) = \mathrm{fold}_{T_2}(w),$$

and equal summaries have equal observations, including failure. For a carrier restricted by an invariant, this consequence additionally requires generated summaries to satisfy the invariant and combination to preserve it. Reduction-tree independence is thus a consequence of the algebraic laws, not their definition.

The developments compile and pass Rocq's kernel checker without admitted lemmas. The array bounded-nesting proofs and the transferred tuple associativity proof use functional extensionality for equality of function-backed arrays. The packaged `TupleProof.Make_is_monoid` additionally uses proof irrelevance to equate consistency proofs attached to equal summaries. These are the only axioms reported by its assumption audit; there are no admitted obligations or custom axioms. The compact one-level theorem requires neither axiom.

This verification does not establish equivalence between the Futhark source and its Rocq model, compiler correctness, or agreement of nesting observation with a separate sequential stack evaluator. Those are distinct correctness obligations. Futhark regression tests complement the proofs by comparing complete summaries across reduction splits, including malformed inputs and depth-bound rejection. Such tests check the implementation but do not bridge these verification boundaries by themselves.

9 CASE STUDIES

The examples in this section illustrate how a small collection of combinators produces queries with different aggregation and parsing structures. Each example maps the input bytes to the delimiter representation required by its innermost module, generates one summary per byte, performs one parallel reduction, and observes the final summary. Thus, the examples differ in their module composition rather than in their reduction driver.

9.1 Grouped aggregates

The maximum-sum query consumes a sequence of floating-point numbers in which commas separate items and vertical bars separate groups. For example,

$$\texttt{34,234,23,23|23,2|22,3,2|122} \quad \longmapsto \quad 314.$$

Starting at the outermost level, the query takes the maximum over completed groups, sums the items within each group, and parses the characters of each item as a floating-point number. Its module composition is

$$\begin{aligned} \mathcal{G}_{sum} &= \mathsf{chunk}_{|}(\mathsf{max}_{\mathbb{R}}, \mathsf{optone}_{\mathbb{R}}), \\ \mathcal{I}_{sum} &= \mathsf{chunk}_{,}(\mathsf{sum}_{\mathbb{R}}, \mathcal{G}_{sum}), \\ \mathcal{Q}_{sum} &= \mathsf{chunk}(\mathsf{float}_{\mathbb{R}}, \mathcal{I}_{sum}). \end{aligned}$$

Here the innermost application has no delimiter of its own: it propagates the comma and vertical-bar bytes to the two enclosing applications. The `optone` adapter supplies the singleton interface expected when a completed group is inserted into the maximum monoid.

Changing only the item aggregate gives the maximum-average and maximum-cardinality queries. The average module from Section 2 uses `dup`, `prod`, `sum`, and `count` to maintain the pair consisting of an item sum and an item count. Substituting it for the inner sum yields

$$\mathcal{I}_{avg} = \mathsf{chunk}_{,}(\mathsf{avg}_{\mathbb{R}}, \mathcal{G}_{avg}), \qquad \mathcal{G}_{avg} = \mathsf{chunk}_{|}(\mathsf{max}_{\mathbb{R}}, \mathsf{optone}_{\mathbb{R}}).$$

For instance, the four groups in `3,5,4|10,12,11|100,200|7,8,9` have averages 4, 11, 150, and 8, so the result is 150. Replacing the item aggregate by `count` and the numeric parser by `nat` instead returns the largest group cardinality. On `3,5,4|10,12,11|3,2,100,200|7,8,9`, the result is 4.

These three queries have identical delimiter structure. Their differences are confined to independently reusable modules, which is the central benefit of separating generation and observation from the summary monoid.

9.2 Filtering before aggregation

The query `sum_greaterthan10` demonstrates that selection need not be built into either parsing or aggregation. It first constructs

$$\mathcal{F} = \mathsf{filter}(\mathsf{nat}_{\mathbb{Z}}, \lambda x.x > 10)$$

and then uses $\mathcal{F}$ as the item parser of a comma-delimited sum. An item that fails the predicate observes as the neutral input for the outer sum; an accepted item observes as its parsed integer. Consequently,

$$\texttt{3,5,4,12,10,11,3,2,100,200,7,8,9} \quad \longmapsto \quad 12 + 11 + 100 + 200 = 323.$$

The parallel reduction still processes all bytes once. Filtering changes the observation of each completed item rather than materialising an intermediate sequence of accepted integers.

9.3 Arithmetic expressions

The sum-of-products query treats addition and multiplication signs as two delimiter levels. Multiplication is consumed by the inner `chunk`, and addition by the outer one:

$$\begin{aligned}\mathcal{A} &= \mathsf{chunk}_{+}(\mathsf{sum}_{\mathbb{R}}, \mathsf{optone}_{\mathbb{R}}),\\ \mathcal{P} &= \mathsf{chunk}_{*}(\mathsf{mul}_{\mathbb{R}}, \mathcal{A}),\\ \mathcal{E} &= \mathsf{chunk}(\mathsf{float}_{\mathbb{R}}, \mathcal{P}).\end{aligned}$$

The order of composition encodes multiplication precedence without an explicit syntax tree. For example, `34*234*23*23+23*2+22*3*2+122` evaluates to 4,209,024.

Parentheses require the nesting constructions of Section 6. For one level, the inner monoid is $\mathcal{E}$ and the outer monoid consumes either raw expression input or a completed value injected by `Nested`. The heterogeneous `parens` combinator then evaluates expressions such as

$$\texttt{100+25*(30+4)*2+7} \quad \longmapsto \quad 1807.$$

The bounded homogeneous construction uses the same nestable expression monoid at every level. Using `Parens.Make` with two successor region modules admits

$$\texttt{12*(34+56*(2+3))+78} \quad \longmapsto \quad 3846,$$

whereas the expression `12*(34+(56*(2+3)))+78` reaches depth three and is rejected at that bound. Instantiating the same combinator at $n = 3$ accepts it and returns 3846. This boundary case distinguishes maximum parenthesis depth from the number of pairs occurring in an expression.

Table 2. Combinators used by the case studies.

Query	Primitive	Horizontal	Structural
Maximum sum	`float`, `sum`, `max`	`optone`	chunk^3
Maximum average	`float`, `sum`, `count`, `max`	`dup`, `prod`, `with_obs`	chunk^3
Maximum cardinality	`nat`, `count`, `max`	`optone`	chunk^3
Filtered sum	`nat`, `sum`	`filter`, `optone`	chunk^2
Sum of products	`float`, `mul`, `sum`	`optone`	chunk^3
Parenthesised expression	As above	`nestable`, `with_gen`	`parens` or `Parens.Make`

Table 2 separates primitive reductions, horizontal adaptation, and delimiter-based structure. The superscript on chunk counts nested applications, including the innermost numeric parser. Across all examples, this composition is resolved statically by Futhark's module system; execution remains a map followed by one reduction and one observation.

10 IMPLEMENTATION AND EVALUATION

We evaluate the maximum-of-group-averages query introduced earlier, comparing generic monoid composition with three reference implementations, and then evaluate parenthesised sums of products against handwritten sequential C. These experiments examine composition overhead, paired-delimiter nesting and multicore scaling. A separate ThinkPad experiment evaluates GPU execution of maximum-of-group-averages and one-level parenthesised sums of products using blocked sequential inner reductions.

10.1 Implementations and inputs

All implementations consume comma-separated numbers in groups separated by |, compute each group's average, and return the maximum average. The *library* implementation composes the numeric parser, delimiter chunking, sum, count, product and observation combinators. The *specialised* Futhark implementation first discovers group boundaries in parallel, then parses and accumulates each group sequentially, processing different groups in parallel, before reducing their averages. It does not materialise individual parsed numbers. The *staged* Futhark implementation discovers token boundaries, parses numbers in parallel and uses a segmented scan of sums and counts to obtain group averages. Its intermediate arrays are explicit in source; their physical materialisation depends on compiler optimisation. Finally, *sequential C* parses and aggregates the input in one pass with constant auxiliary space. All four implementations use double-precision arithmetic.

The repository's input generator produces groups of 4–49 positive integers between 1 and 1299. We generated inputs of approximately 0.11, 1.11, 11 and 100 MB, where MB denotes 10^6 bytes. The largest contains 100,004,359 bytes, 910,000 groups and 24,110,226 numbers. To isolate the effect of group layout, we also replaced every group delimiter except the first at or after the byte midpoint with a comma. The resulting input has two approximately 50 MB groups, containing 12,055,124 and 12,055,102 numbers. It preserves all numeric tokens, their order and the total byte count, but changes the query result. This deliberately low-group-count case tests the specialised implementation's limited within-group parallelism.

10.2 Measurement method

Unless stated otherwise, measurements were collected on an Apple M2 Max desktop with 12 logical CPUs and 32 GiB of memory, using Futhark 0.26.4 and Apple Clang 21.0.0. Both the generated

Table 3. Many small groups, 100 MB: median milliseconds (sample standard deviation).

Implementation	1 worker	2 workers	4 workers	8 workers
Sequential C	178.71 (0.78)	—	—	—
Library	343.64 (1.10)	180.07 (1.59)	93.42 (0.37)	48.18 (0.37)
Specialised	325.05 (1.07)	175.92 (1.21)	93.59 (0.41)	49.05 (0.46)
Staged	669.19 (1.60)	373.06 (9.66)	202.55 (0.48)	116.56 (0.86)

Table 4. Two large groups, 100 MB: median milliseconds (sample standard deviation).

Implementation	1 worker	2 workers	4 workers	8 workers
Sequential C	167.63 (0.81)	—	—	—
Library	333.98 (0.45)	174.04 (1.58)	89.67 (0.70)	46.76 (0.69)
Specialised	317.88 (1.17)	172.83 (1.31)	130.89 (0.75)	111.67 (1.03)
Staged	670.27 (3.85)	362.76 (1.44)	199.92 (0.80)	115.33 (1.82)

Futhark executables and the C references were native ARM64 binaries running on macOS. Futhark's multicore benchmark runner used 1, 2, 4 and 8 workers, with `-runs=20`, default convergence and no tuning-file loading. The C reference was compiled with `-O3 -ffp-contract=off`, without link-time optimisation or fast-math, and used one untimed warm-up followed by 20 timed executions. Compilation, file loading and result output are excluded. All configurations were checked against an independent exact-rational reference answer, using relative tolerance 10^{-10} and absolute tolerance 10^{-9}; all checks passed. Input hashes, compiler metadata and raw timing samples are retained with the benchmark artefacts.

Tables 3 and 4 report median execution times, followed in parentheses by sample standard deviations in milliseconds. For samples $t_1, \ldots, t_k$, the latter is $s = \sqrt{\sum_i (t_i - \bar{t})^2/(k-1)}$, where $\bar{t}$ is their arithmetic mean. Each table entry uses 20 samples. These deviations describe within-process run-to-run dispersion, not confidence intervals or variability across independent campaigns.

10.3 Results and limitations

With many small groups, the library and specialised implementation have similar eight-worker times. The library achieves a 7.13× speedup over its one-worker execution and input throughput of 2.08 GB/s. Sequential C is faster than the one-worker library, but the eight-worker library is 3.71× faster than sequential C.

With two large groups, the library retains 7.14× scaling and reaches 2.14 GB/s at eight workers, outperforming the specialised implementation by 2.39×. The latter has only two independent group-parsing tasks, although its boundary-discovery pass can still use additional workers. Specialised remains slightly faster at one and two workers. This result demonstrates a benefit of within-group parallel reduction over this particular group-parallel baseline, not over every possible specialised parser: a block-parallel specialised implementation could also exploit such parallelism.

These grouped-average results are preliminary desktop measurements. Workers were not pinned, execution order was fixed, C ran after Futhark, and desktop activity was not isolated. Warm-up and convergence policies differ between the runners; the two input layouts were measured in separate campaigns. Independent repeated campaigns and a sweep of group counts are needed for

Table 5. Depth-one specialisations, 100 MB: median milliseconds (sample standard deviation). C denotes Futhark's sequential C backend.

Comparison	Variant	C	8 workers
Nesting	`parens`, `float`	691.29 (2.38)	120.87 (1.14)
	`parens_hom`, `float`	594.47 (2.44)	117.39 (0.73)
Numbers	`parens_hom`, `float`	575.25 (2.45)	120.42 (1.77)
	`parens_hom`, `decimal`	587.90 (2.56)	99.73 (2.25)

stronger conclusions for this query. The following experiment includes fractional inputs and one-level nesting; malformed-input handling, peak memory use and GPU execution of bounded-depth nesting remain for future evaluation.

10.4 Parenthesised sums of products

We next evaluate the arithmetic-expression composition from Section 6 with parenthesis depth exactly one. Inputs are sums of blocks $x * (y * z + u * v)$, where each operand is independently generated from $\{1/8, 2/8, \ldots, 7/8\}$ and written as a decimal literal. The generator uses seed 2026 and constructs an exact-rational reference answer alongside the expression. Target sizes are 0.1, 1, 10, 100 and 200 MB; each input exceeds its target by 31 bytes to finish a complete block. The 200 MB input contains 6,250,001 blocks. A handwritten sequential C reference evaluates the expression in one pass, respecting precedence, without materialising tokens or a syntax tree. Its evaluation stack is bounded by the supported parenthesis depth. Both implementations return a double-precision result.

Useful specialisations. We measure two specialisations introduced above. First, `parens_hom` is the homogeneous, depth-one variant of `parens`: it takes a single component accepting raw input or an already evaluated nested value. A raw region therefore needs only one component summary, rather than separate inner and outer summaries. Closing a region observes that summary and injects its value as a nested operand. This variant uses a fixed constructor-based representation. Its constructors encode the structural cases, including failure, without separate depth counts or an additional outer option tag.

Second, `decimal` is a restricted alternative to `float`. It accepts an optional minus sign, at least one digit, and optionally a point followed by at least one digit, with at most 18 digits in total, including leading zeros. Exponents and a leading plus sign are not accepted. Its summary uses an unsigned 64-bit mantissa, a digit count, a fractional digit count and a sign, together with failure and incomplete-sign cases. Composition accumulates the mantissa using integer arithmetic; observation converts and scales the result to floating point. Powers of ten are supplied by an explicit case function. This restriction suffices for the benchmark but is not a general replacement for `float`, nor a claim of generally correctly rounded conversion. Failures propagate internally; invalid final observation asserts, while `decimal_try` returns an option. The measured composition retains ordinary `chunk` and wraps the numeric parser with `nestable` to accept evaluated subexpressions.

Table 5 reports separate controlled comparisons on the same 100 MB input family. With `float`, replacing `parens` by `parens_hom` reduces sequential-backend time by 14.0% and eight-worker time by 2.9%. Within `parens_hom`, replacing `float` by `decimal` reduces eight-worker time by 17.2%, but increases sequential-backend time by 2.2%. These are end-to-end expression measurements, not isolated parser timings. The two comparisons were run in different campaigns and should not be read as one cumulative timing series.

Table 6. Depth-one expressions with `decimal` and `parens_hom`: median milliseconds (sample standard deviation). MC denotes the multicore backend; the reference is handwritten sequential C.

MB	Reference C	Futhark C	MC 1	MC 2	MC 4	MC 8
0.1	0.313 (0.027)	1.214 (0.233)	1.214 (0.179)	0.425 (0.025)	0.236 (0.010)	0.123 (0.014)
1	1.105 (0.045)	6.064 (0.117)	6.067 (0.097)	3.721 (0.082)	1.935 (0.036)	0.991 (0.212)
10	11.089 (0.101)	60.632 (0.385)	60.780 (0.631)	36.883 (0.285)	19.090 (0.130)	9.671 (0.088)
100	110.538 (0.738)	604.499 (2.421)	614.375 (5.690)	370.151 (2.046)	188.015 (0.421)	96.001 (0.645)
200	221.619 (1.703)	1228.695 (3.601)	1237.896 (3.997)	743.419 (1.882)	377.736 (10.744)	193.269 (3.282)

Table 7. Irregular depth-one expressions, 100 MB, using `decimal` and `parens_hom`: median milliseconds (sample standard deviation), 20 samples per configuration.

Configuration	Time
Handwritten sequential C	128.50 (0.33)
Futhark sequential C	574.06 (1.27)
Multicore, 1 worker	573.69 (4.14)
Multicore, 2 workers	344.20 (2.43)
Multicore, 4 workers	179.20 (0.89)
Multicore, 8 workers	92.25 (0.71)

Protocol and comparison with C.. These runs use the same native ARM64 platform and compiler versions as the grouped-average experiment. The executables were compiled with `-O3 -arch arm64 -ffp-contract=off`, without fast-math or link-time optimisation. Each configuration performs an untimed warm-up and 20 timed executions; generation, compilation, input loading and output are excluded. All results pass the rational oracle with absolute and relative tolerances of 10^{-9}. Input hashes, source and compiler metadata, generated C and raw samples are retained in the benchmark artefacts.

Table 6 compares `decimal` within `parens_hom` against handwritten C. All sizes and both implementations were remeasured in this campaign. At 200 MB, the eight-worker library is 1.15× faster than handwritten sequential C, attaining 1.03 GB/s, but its sequential C backend is 5.54× slower. Thus parallelism compensates for substantial sequential composition overhead; the experiment does not show an advantage over a specialised parallel expression evaluator.

Sensitivity to expression structure. To examine a less regular workload, we generated a second input of 100,000,169 bytes, retaining depth one and the same uniform distribution of positive dyadic literals. Unlike the fixed block pattern, its top-level products contain one to six factors, each chosen to be a literal or a parenthesised expression. Parenthesised sums contain between one and 32 terms, each with one to six literal factors. The generator uses a seeded random stream rather than repeating a bank of expressions. The resulting input contains 410,759 top-level terms, 575,385 parenthesis pairs and 16,474,900 literals. Thus the comparison varies not only regularity but also the relative frequencies of arithmetic and parenthesis operations: the regular 100 MB input contains 3,125,001 parenthesis pairs.

Table 7 uses freshly built native ARM64 executables and the same compiler settings, warm-up, 20-sample protocol and oracle tolerances as the preceding comparison. All checks passed. The eight-worker library is 1.39× faster than handwritten C, compared with 1.15× on the regular 100 MB input, and scales by 6.22× from one worker. Relative to the earlier regular-input campaign,

handwritten C takes 16.2% longer, whereas the eight-worker library takes 3.9% less time. These observations suggest that relative performance is sensitive to the expression workload even at a fixed byte length and nesting depth.

There are two important qualifications. First, these are separate desktop campaigns, not a contemporaneous paired comparison; the percentages do not isolate a causal effect of input structure. Second, changing expression shape changes the work performed per byte. Fewer parenthesis pairs reduce the frequency of closing and injecting nested results, while changes in operator sequences may affect dispatch and branch prediction. These are possible explanations, not conclusions established by profiling. In particular, the result does not imply that irregular expressions are intrinsically easier for monoidal evaluation. A controlled sweep of parenthesis density and product and sum lengths, with interleaved repeated measurements, is needed to separate these effects. Byte throughput alone does not fully characterise the cost of an expression workload.

Scope and unsuccessful alternatives. The regular-input configurations execute serially with alternating comparison order, without core pinning or desktop isolation. The table reports within-process dispersion, not variation across independent campaigns. The irregular-input campaign uses fixed configuration order, with the same lack of core pinning and desktop isolation. Both input families contain short positive dyadic literals and a fixed nesting depth; they do not establish performance for arbitrary decimal syntax or deeper nesting. Further representation changes did not consistently help. In particular, a specialised sum-of-products summary, retaining boundary products and the sum of completed interior terms, reduced arithmetic chunking but was 2.7–3.8% slower inside `parens_hom` at eight workers on approximately 100 MB of varied depth-one expressions, across two timing rounds and short and 18-digit padded literal profiles. We therefore retain ordinary `chunk` in the reported implementation. A smaller or more specialised summary alone is not sufficient evidence of improved execution time.

10.5 Tuple-backed nesting at depths two, three and five

We evaluate the public `Parens.Make` construction of Section 7 at bounds $n \in \{2, 3, 5\}$. Each tuple is built by n applications of `Succ` to `Zero`. All instances use the same `decimal`, `nestable`, multiplication, sum and ordinary `chunk` components, with compact signed-byte metadata and replay arithmetic.

Inputs and protocol. The regular generator recursively constructs blocks $B_0 = x * y$ and $B_d = x * (B_{d-1} + y * z)$ for $d > 0$, drawing each literal independently from the same seven positive dyadic values as the depth-one experiment. Independent blocks are joined by addition until the target size is reached. The 100 MB inputs contain 100,000,003, 100,000,007 and 100,000,095 bytes, respectively, and have exact maximum depths two, three and five. Their block counts are 1,923,077, 1,388,889 and 892,858. The seed is 2026. Each parser's bound equals its input's maximum depth; this is not a sweep of over-provisioned bounds on one fixed input.

The experiment uses the same Apple M2 Max and native ARM64 compiler settings as above, comparing handwritten sequential C, Futhark's sequential C backend, and its multicore backend with 1, 2, 4 and 8 workers. All compilation and generation finish before timing. Each configuration has two rounds of five timed executions, each preceded by an untimed warm-up. The second round reverses depth and configuration order. Idle sleep is inhibited, but there is no core pinning or desktop isolation. Input loading and result output are excluded. The benchmark artefacts retain sources, hashes, generated C, raw samples and oracle checks against the exact-rational answers, using absolute and relative tolerances of 10^{-9}.

Table 8 confirms increasing tuple cost with depth. At eight workers, 100 MB takes 720, 967 and 1802 ms for depths two, three and five. Relative to the one-worker multicore executable, the

Table 8. Tuple-backed nesting at 100 MB: median milliseconds (sample standard deviation). Two five-sample rounds are pooled (10 samples per cell); the reference is handwritten sequential C.

n	Reference C	Futhark C	MC 1	MC 2	MC 4	MC 8
2	109.0 (0.8)	2139.9 (24.3)	5488.7 (72.5)	2789.7 (20.1)	1406.8 (3.6)	719.7 (2.5)
3	110.6 (1.9)	4944.4 (75.7)	7350.9 (101.7)	3756.3 (24.1)	1890.9 (3.6)	967.5 (3.5)
5	114.8 (2.9)	9329.2 (81.8)	13845.2 (132.5)	6997.7 (29.9)	3534.4 (23.0)	1801.9 (15.4)

speedups are 7.63×, 7.60× and 7.68×. However, the corresponding eight-worker times are still 6.60×, 8.74× and 15.70× those of handwritten sequential C. Thus, despite near-linear multicore scaling, bounded generic nesting remains slower than this specialised sequential evaluator on these deeper expressions. All 36 large-input configuration checks passed. The smaller sample count and desktop execution conditions limit the precision of these estimates; the complete samples from both rounds are included, without selecting the faster round.

Depth affects both representation size and workload structure in this experiment. The tuple capacities are five, seven and eleven component summaries; deeper inputs also exercise different replay lengths. Moreover, the 100 MB inputs contain approximately 3.85, 4.17 and 4.46 million parenthesis pairs, respectively, while their numbers of literals decrease slightly. The sweep therefore does not isolate the cost of adding a tuple level: it changes the bound and the expression family together. A separate experiment holding input fixed while increasing only the bound would measure the cost of over-provisioning. Multicore scaling is measured against the one-worker multicore executable, since Futhark's sequential C backend can generate different code even when both execute on one core.

Compact-metadata trade-off. An earlier controlled depth-two experiment compared the same inductive tuple representation with three metadata choices: `i64`, stored bytes widened to `i64` for arithmetic, and bytes retained throughout the arithmetic. On a 100 MB regular input, sequential C medians were 2206.40, 2231.96 and 2089.21 ms, respectively, with sample standard deviations 1.44, 2.79 and 3.39 ms. At eight workers the corresponding values were 662.27 (3.11), 808.52 (0.70) and 702.79 (20.23) ms. Narrow arithmetic recovered most of the widening overhead, but was still 6.1% slower than `i64` at eight workers; it was 5.3% faster on the sequential backend. These paired groups were repeated after noisy original measurements and are retained with those original samples. They belong to a separate campaign and are not combined with the depth-sweep samples. The compact representation was selected for its smaller counter payload, not because these CPU measurements establish a uniform speedup. No GPU performance claim follows from the space saving.

10.6 GPU execution with blocked reductions

We separately evaluate maximum-of-group-averages and one-level parenthesised sums of products on a ThinkPad running Ubuntu 26.04, with an Intel Core i7-1360P and an NVIDIA RTX A500 Laptop GPU with 4 GiB of device memory. The processor has four performance cores and eight efficiency cores. We report only one-, two- and four-worker multicore configurations on this machine. Workers were not pinned, so these counts denote software workers, not measurements restricted to the performance cores. These x86-64 CPU and CUDA measurements are distinct from the preceding ARM64 experiments. We use Futhark 0.25.32, CUDA Toolkit 12.8 and NVIDIA driver 595.91.07, with the laptop connected to AC power and the balanced power profile selected.

The maximum-of-group-averages inputs contain signed decimal literals with two fractional digits, generated from integer hundredths between −999999 and 999999. We use approximately

1, 16 and 100 MB inputs, with either 4–12 numbers per group (short groups) or 512–1024 (long groups), and a deterministic seed of 2026. These workloads differ from the integer inputs and two-group input above. Both the `float` and `decimal` variants use double-precision aggregation. An independent one-pass sequential C parser and aggregator is compiled with `-O3 -march=native -ffp-contract=off`, without link-time optimisation or fast-math.

Blocking without changing the monoid. The direct GPU reduction was substantially slower than multicore execution. Profiling its float variant on the short-group 100 MB input showed 32.4% achieved occupancy, an average of 7.6 active lanes per 32-lane warp, and substantial block-barrier waiting, while DRAM throughput was below 0.3% of peak. Register use was high (123 registers per thread), but offline assembly reported no spills. These observations motivate amortising cross-thread combination over sequential local work rather than changing the monoidal interface.

Let g be the generator of the composed monoid M. We partition the byte sequence into contiguous blocks of at most B bytes, reduce each block sequentially, and reduce the resulting summaries in their original order. Only the reduction expression changes; the generator, operator, neutral element and observation are unchanged. In Futhark, the relevant expression is:

```
let n = length xs
let summaries = map (\b →
  let start = b * block_size
  let len = i64.min block_size (n - start)
  in #[sequential]
     reduce M.op M.ne (map g xs[start:start+len]))
  (iota ((n + block_size - 1) / block_size))
in M.obs (reduce M.op M.ne summaries)
```

Contiguous blocks preserve the order required by the non-commutative summary operation; they need not end at delimiters. The algebraic monoid laws justify regrouping, subject to the floating-point qualification in Section 8. Inspection of generated CUDA confirms a sequential character loop inside the bulk reduction kernel: the compiler fuses local folds with the outer reduction rather than materialising the entire array of block summaries.

An exploratory sweep of $B \in \{256, 1024, 4096, 10000\}$ found similar performance for 256–4096 bytes and worse performance at 10000 bytes. We select $B = 1024$ for the following comparison. In that separate sweep, blocking reduced 100 MB execution times by approximately 5–6× relative to freshly measured direct reductions. This is more effective than merely changing the CUDA thread-block size, which gave only a modest, workload-dependent improvement. The multicore implementations below retain their original, unblocked reduction.

Method and results. We reran C, multicore and blocked GPU configurations in two rounds with reversed configuration order, checking AC connection before each timed configuration. C has 20 measured executions; Futhark has at least 20, with additional samples for short runs. Warm-up is excluded and the Futhark convergence phase and tuning-file loading are disabled. All reported results passed oracle checks. Compilation, file loading and GPU input transfer are excluded: the GPU timings describe execution with input already resident on the device, not end-to-end latency. Tables report medians with sample standard deviations in parentheses, in milliseconds.

On the configured ThinkPad, for 100 MB, blocked GPU float is 3.79× and 4.35× faster than sequential C on short and long groups, respectively, and 2.17× and 1.60× faster than four-worker float. At 16 MB it also outperforms four-worker float; at 1 MB multicore is faster. Decimal does not improve these blocked GPU medians. The short-group CPU measurements have substantial variability. Affinity, clocks and thermals were not controlled, and the block size was selected on these workloads; the results are not evidence of a universal optimum or of end-to-end GPU speedups.

Groups	Parser	1 worker	2 workers	4 workers	GPU
Short	Float	253.85 (1.13)	138.71 (16.30)	98.95 (15.24)	45.51 (0.36)
Short	Decimal	322.72 (1.48)	192.63 (15.76)	131.84 (19.14)	45.92 (0.10)
Long	Float	231.63 (2.07)	116.47 (2.53)	57.65 (1.44)	35.97 (0.23)
Long	Decimal	302.36 (0.95)	150.74 (2.28)	75.47 (4.35)	38.49 (0.13)

Table 9. ThinkPad maximum-of-group-averages, 100 MB. GPU reductions use 1024-byte sequential inner folds. Sequential C takes 172.67 (1.47) ms for short groups and 156.52 (0.45) ms for long groups. CPU worker counts are unpinned.

MB	Groups	Sequential C	Float, 4 workers	Float, GPU
1	Short	1.98 (0.88)	1.15 (0.29)	1.81 (0.09)
1	Long	1.84 (0.56)	0.97 (0.16)	1.40 (0.07)
16	Short	28.65 (0.64)	16.71 (2.68)	7.61 (0.15)
16	Long	24.97 (0.58)	9.25 (3.66)	6.00 (0.09)

Table 10. Smaller ThinkPad inputs with the same blocked GPU configuration. Decimal GPU times are 1.78 (0.03), 1.47 (0.02), 7.67 (0.13) and 6.37 (0.14) ms, respectively.

Nevertheless, the experiment shows that the same monoidal composition can benefit substantially from changing the reduction layout without changing its component interfaces.

One-level parenthesised expressions. We also block the reduction of the `parens_hom` sum-of-products query, using either `float_nested` or `decimal_nested` with double-precision aggregation. The library combinators are unchanged. Two inputs of approximately 100 MB, generated with seed 2026, have maximum parenthesis depth one. The regular input sums motifs of the form `x*(y*z+u*v)`; the irregular input varies the numbers of terms, factors and parenthesised factors. Literals are dyadic decimals from 0.125 to 0.875, supporting an exact-rational oracle.

We sweep the same four byte-block sizes and select $B = 4096$, which gives the lowest measured median for both parsers on both inputs. At 1024 bytes, regular float and decimal take 35.63 and 30.49 ms, compared with 35.56 and 29.84 ms at 4096 bytes; irregular input benefits more, falling from 207.62 and 192.03 ms to 166.26 and 148.40 ms. Increasing the block size to 10000 bytes worsens all four medians. Generated-code inspection again finds a sequential character loop fused into the bulk reduction kernel. This experiment does not isolate the effects of amortised cross-thread combination and compiler specialisation of local updates.

Table 11 compares freshly measured direct and blocked GPU reductions with the unchanged multicore implementations and sequential C reference. The hardware, execution-only timing convention and AC checks are as above. Each configuration has at least 20 samples across two rounds with reversed configuration order; GPU sweeps precede the fresh CPU runs, rather than being interleaved with them. All 24 small-expression checks, 16 large-input variant/block-size checks and timed outputs pass comparison with the oracle at relative and absolute tolerances of 10^{-9}.

Blocking accelerates direct GPU float and decimal by 24.0× and 31.7× on regular input, but only 5.0× and 6.4× on irregular input. On the configured ThinkPad, blocked decimal is 4.08× faster than C on regular input; on irregular input it is only 1.07× faster, while blocked float remains slower than C. Thus input structure strongly affects the benefit even at the same input size and maximum depth. Decimal helps both blocked workloads, unlike the grouped-average experiment,

Input	Parser	Direct GPU	Blocked GPU	4 workers
Regular	Float	852.78 (1.32)	35.56 (0.25)	282.63 (13.61)
Regular	Decimal	946.57 (0.41)	29.84 (0.17)	359.82 (8.07)
Irregular	Float	829.18 (0.97)	166.26 (1.25)	279.68 (15.31)
Irregular	Decimal	955.97 (0.85)	148.40 (1.25)	390.55 (7.87)

Table 11. ThinkPad `parens_hom`, 100 MB, median execution time in ms (sample standard deviation). Blocked GPU uses 4096-byte sequential folds. Sequential C takes 121.81 (2.22) ms on regular input and 158.48 (3.27) ms on irregular input. Four-worker CPU reductions are unblocked and unpinned; they are faster than the measured one- and two-worker configurations in all four cases.

but does not remove this sensitivity. These workload-selected block sizes and execution-only measurements establish neither a universal optimum nor end-to-end speedups. The experiment concerns `parens_hom`, not the deeper, module-inductive `Parens.Make` representation.

11 RELATED WORK

List homomorphisms and parallel reductions. List homomorphisms provide the classical foundation for computing a function by mapping input fragments and combining their summaries with an associative operation. Gibbons's third homomorphism theorem characterises when compatible leftwards and rightwards computations admit such a representation [8]. Cole develops list homomorphisms as a practical method for parallel programming and includes a language-recognition example with nested parallelism [3]. Geser and Gorlatch instead address the systematic derivation of a homomorphic representation from sequential leftwards and rightwards programs [7]. Our starting point is not the discovery of a homomorphism for one closed program, but the construction of homomorphisms from separately reusable generated and observable monoids.

Kiselyov introduces observable monoids and their vertical `chunk` composition, and demonstrates both Horner composition and nested grouping over raw data [11]. The present work implements that design in Futhark, extends numeric parsing to floating-point values, adds the horizontal combinators of Section 4, and introduces paired delimiter combinators for bounded nesting.

Monoids in query processing. Fegaras and Maier's monoid calculus uses monoid comprehensions as an intermediate language for object queries, supporting aggregation, nested collections, and query transformation [6]. Fegaras later uses monoid homomorphisms and comprehensions as the algebraic basis of MRQL, a distributed query processing and optimisation system [5]. Lin similarly presents monoids as a design principle for local aggregation in MapReduce [13]. These systems operate at the level of collections and distributed query plans. Our combinators instead build a single reduction directly over the encoded input, with parsing and aggregation represented in the same constant-size summary.

Parallel parsing and nested words. Parallel operator-precedence parsing exploits local parsability so that independently processed input segments can be joined at their boundaries. Barenghi et al. develop this approach into a parser generator and evaluate it on JSON and Lua [2]. Li and Taura relax deterministic operator-precedence parsing when the semantic operation connecting parallel elements is associative [12]. These approaches construct syntax, whereas our restricted parsers compute an application-specific observation without materialising a syntax tree.

The separation of input bytes into opening, closing, and ordinary symbols is also reminiscent of visibly pushdown languages, where the input symbol determines the stack action [1]. The relationship is structural rather than an expressiveness claim: `Parens.Make` supports a statically

bounded depth and computes monoidal observations, while visibly pushdown automata recognise an unbounded class of nested words.

Futhark as an implementation setting. Futhark supplies the parallel `map` and `reduce` operators to which the library targets [9]. Of particular importance here, its higher-order module system is eliminated by static interpretation, so module-level abstraction need not introduce run-time dispatch [4]. Source-level higher-order functions are eliminated separately by Futhark's type-directed defunctionalisation, which introduces no run-time branching [10]. For example, the predicate supplied to `filter` is higher-order behaviour from the library user's perspective. Static interpretation resolves the enclosing module application, and defunctionalisation removes any remaining functional representation, leaving direct predicate code in the specialised reduction. Together, these mechanisms allow the library to express combinators as higher-order modules and functions while exposing one first-order summary type and operation to the final data-parallel reduction.

12 LIMITATIONS AND FUTURE WORK

The library supports heterogeneous nesting at one level and homogeneous nesting up to a static depth bound. Its fixed-size summaries avoid storage that grows with input length, but their size and combination cost still grow with the chosen bound. The Rocq proofs establish algebraic monoid laws; they do not establish compiler correctness or equivalence to a separate sequential parser.

The GPU experiments illustrate the importance of reduction layout and input structure, but do not establish performance portability across nested queries. The behaviour of deeper module-inductive nesting under GPU memory, register and execution constraints remains open.

Further work includes specialising replay operations and extending queries to richer tokenisation and multiple paired-delimiter classes. Dynamically bounded nesting would require revisiting the fixed-size summary design.

A longer-term target is one-touch queries over structured data such as

```
[
  { "id": 5, "score": 8 },
  { "id": 5, "score": 3 },
  { "id": 3, "score": 9 }
]
```

For example, a query could return the average of the scores whose `id` field equals 5. A challenge is that commas separate both object members and array elements, so their interpretation depends on the surrounding structure.

13 CONCLUSIONS

Generated and observable monoids provide a compositional interface for combining parsing and aggregation into a single data-parallel reduction. The Futhark library extends delimiter-based composition with numeric parsing, horizontal combinators and paired nesting, while the Rocq development establishes the algebraic laws required for reassociation.

The experiments show both the promise and the cost of this approach. Specialised representations and multicore execution can make one-touch queries effective, but deeper nesting and irregular input expose substantial overheads, and generic nesting remains slower than the handwritten sequential reference in the reported deeper-expression experiments.

The GPU grouped-average experiment shows that reduction layout is equally important. Sequential local folds over contiguous blocks amortise cross-thread summary combination and improve

direct GPU execution by approximately 5–6×, while retaining the same combinators. On the configured ThinkPad, for 100 MB inputs, the blocked query outperforms handwritten sequential C by 3.8–4.4× and four-worker multicore execution by 1.6–2.2×, excluding input loading and GPU transfers. This benefit does not extend uniformly to small inputs. For one-level parenthesised sums of products, blocking with the restricted-decimal parser outperforms sequential C on the same ThinkPad by 4.1× on regular 100 MB input, but only 1.07× on irregular input. Deeper module-inductive nesting remains unevaluated on the GPU. Compositionality and verified associativity therefore provide a foundation for adapting parallel execution, not a guarantee of competitive performance for every layout and workload.

ACKNOWLEDGEMENT

The development was assisted using GPT-6 Astra.

REFERENCES

[1] Rajeev Alur and P. Madhusudan. 2004. Visibly Pushdown Languages. In *Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing (STOC '04)*. Association for Computing Machinery, New York, NY, USA, 202–211. https://doi.org/10.1145/1007352.1007390

[2] Alessandro Barenghi, Stefano Crespi Reghizzi, Dino Mandrioli, Federica Panella, and Matteo Pradella. 2015. Parallel Parsing Made Practical. *Science of Computer Programming* 112, 3 (2015), 195–226. https://doi.org/10.1016/j.scico.2015.09.002

[3] Murray Cole. 1995. Parallel Programming with List Homomorphisms. *Parallel Processing Letters* 5, 2 (1995), 191–203. https://doi.org/10.1142/S0129626495000175

[4] Martin Elsman, Troels Henriksen, Danil Annenkov, and Cosmin E. Oancea. 2018. Static Interpretation of Higher-Order Modules in Futhark: Functional GPU Programming in the Large. *Proceedings of the ACM on Programming Languages* 2, ICFP, Article 97 (2018), 30 pages. https://doi.org/10.1145/3236792

[5] Leonidas Fegaras. 2017. An Algebra for Distributed Big Data Analytics. *Journal of Functional Programming* 27 (2017), e27. https://doi.org/10.1017/S0956796817000193

[6] Leonidas Fegaras and David Maier. 1995. Towards an effective calculus for object query languages. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data* (San Jose, California, USA) *(SIGMOD '95)*. Association for Computing Machinery, New York, NY, USA, 47–58. https://doi.org/10.1145/223784.223789

[7] Alfons Geser and Sergei Gorlatch. 1999. Parallelizing Functional Programs by Generalization. *Journal of Functional Programming* 9, 6 (1999), 649–673. https://doi.org/10.1017/S0956796899003536

[8] Jeremy Gibbons. 1996. The Third Homomorphism Theorem. *Journal of Functional Programming* 6, 4 (1996), 657–665. http://www.cs.ox.ac.uk/people/jeremy.gibbons/publications/thirdht.ps.gz Earlier version appeared in C. B. Jay, editor, *Computing: The Australian Theory Seminar*, Sydney, December 1994, p. 62–69.

[9] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: purely functional GPU-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 556–571. https://doi.org/10.1145/3062341.3062354

[10] Anders Kiel Hovgaard, Troels Henriksen, and Martin Elsman. 2019. High-Performance Defunctionalisation in Futhark. In *Trends in Functional Programming (Lecture Notes in Computer Science, Vol. 11457)*, Michał Pałka and Magnus Myreen (Eds.). Springer, Cham, 136–156. https://doi.org/10.1007/978-3-030-18506-0_7

[11] Oleg Kiselyov. 2026. More Fun with Monoids: Declarative Pearl. In *Functional and Logic Programming: 18th International Symposium, FLOPS 2026, Tsukuba, Japan, May 26–28, 2026, Proceedings* (Akita, Japan). Springer-Verlag, Berlin, Heidelberg, 156–172. https://doi.org/10.1007/978-981-92-0184-6_7

[12] Le Li and Kenjiro Taura. 2023. Associative Operator Precedence Parsing: A Method to Increase Data Parsing Parallelism. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region (HPCAsia '23)*. Association for Computing Machinery, New York, NY, USA, 75–87. https://doi.org/10.1145/3578178.3578233

[13] Jimmy Lin. 2013. Monoidify! Monoids as a Design Principle for Efficient MapReduce Algorithms. arXiv:1304.7544 [cs.DC] https://arxiv.org/abs/1304.7544