Unifying Paths for Updates and Sections in Futhark

Aziz Rmadi

Block 3 - 2026

1 Abstract

Futhark is a purely functional, data-parallel programming language that uses uniqueness types to support efficient functional updates [1]. The primary construct for such updates is the `with` expression, which returns a new value with a designated component replaced. For arrays, these updates may be compiled using memory reuse when uniqueness permits, while record updates are semantically ordinary functional updates. Futhark also supports section syntax for partial projections such as `(.f)` and `(.[0])`.

Prior to this work, array updates, record updates, and their corresponding section forms were handled by separate grammar productions and distinct abstract syntax tree (AST) constructors. This split design prevented uniform support for mixed paths such as `[i].f[j]` in updates and `(.[0].f)` in sections. It also made certain nested updates verbose or impossible to express naturally in the presence of uniqueness and consumption constraints.

This project introduces a unified path-based representation built around a common `UpdateStep` abstraction. Array indexing and slicing, as well as record field access, are represented uniformly as steps in a single path. This representation is used throughout the source frontend for ordinary `with` expressions, let-with shorthand syntax, and sections. The core compiler IR remains unchanged; instead, the implementation is carried out entirely in the frontend through parsing, type checking, normalization, and internalisation.

The implementation preserves Futhark's uniqueness guarantees and handles subtle cases involving consumption and evaluation order, such as updates where index expressions and replacement expressions interact through uniqueness. At the same time, it simplifies the frontend by replacing several parallel syntactic mechanisms with one shared representation. The result is a cleaner and more extensible treatment of paths in Futhark.

2 Introduction

Futhark provides a functional interface to updates through its uniqueness type system. A `with` expression returns a new value in which some component has been replaced. For example:

```
let xs = [1, 2, 3]
let ys = xs with [1] = 9
```

produces a new array `ys` equal to `[1, 9, 3]`. Likewise,

```
let r = {a = 1, b = 2}
let r' = r with a = 3
```

produces a new record `r'` equal to {`a = 3, b = 2`}. These are source-level functional updates, although for arrays the compiler may reuse memory when uniqueness permits.

Although array indexing and record projection are conceptually similar — both describe navigation along a path into a value — they were historically treated separately in the source language and compiler. Arrays used slice-based update syntax, while records used field-based syntax. This split was also reflected in section syntax, where simple index sections and field sections were supported, but mixed forms were not.

The goal of this project was to eliminate this separation by introducing a unified representation of paths. Instead of viewing array updates, record updates, and path sections as unrelated constructs, the compiler now treats them as different uses of the same underlying sequence of update steps. This makes it possible to express mixed nested updates directly, extend let-with shorthand to records and mixed paths, and support mixed sections such as `(.[0].f)` and `(.a[1:3])`.

The implementation was developed across three merged changes to the futhark code base, revolving around adding support for shortened in-place record updates [3], the unification of record and array updates through paths [4], and the unification of sections through the same mechanism [5].

The main features of this work are as follows:

- A unified AST representation of update and projection paths using a common `UpdateStep` datatype.
- A parser refactoring that supports mixed array and record paths in `with` expressions, let-with shorthand, and sections.
- A frontend-only implementation that requires no changes to the core compiler IR.
- Preservation of Futhark's uniqueness and consumption guarantees through careful handling of desugaring, evaluation order, and source-expression reuse during internalisation.

The remainder of this report first motivates the change, then presents the unified path representation, its implementation in the frontend, and its semantic and compiler consequences.

3 Motivation

The compiler previously treated array updates and record updates as distinct syntactic categories. This distinction appeared both in the grammar and in the AST. As a consequence, operations that were conceptually instances of the same idea — navigating a path into a nested value — were implemented through separate mechanisms.

A direct consequence of this split design was that mixed paths were hard, and sometimes impossible, to express directly. For example, the following are three different path updates that are naturally understood as single updates:

- `xs with [0].f = x`
- `r with f[1] = x`
- `xss with [0].a[1] = v`

Under the previous design, such updates had to be expressed using nested update forms. For example, the first two would conceptually have to be written as:

```
xs with [0] = (xs[0] with f = x)
r with f = (r.f with [1] = x)
```

This was not only verbose. It also interacted badly with uniqueness typing. Since indexed array updates consume their source, a nested formulation could trigger consumption errors even when the programmer's intention was a single functional replacement along a path.

This issue was not merely theoretical. Users explicitly requested support for updating arrays stored inside records [2]. Under the previous design, an expression such as:

```
x with a = (x.a with [0] = 1337)
```

could fail with a consumption error. The problem is that the inner array update must be evaluated before the outer record update can be formed, and the array update consumes the source value for uniqueness reasons. Since consumption is tracked at the granularity of whole variables rather than individual record fields, the compiler cannot express that only `x.a` has been consumed while the rest of `x` remains available.

The workaround was to decompose the record manually and reconstruct it:

```
let a = x.a
let b = x.b
in {a = a with [0] = 1337, b = b}
```

This quickly becomes unreadable for larger records. More generally, it showed that the lack of mixed update paths was a real usability problem, not just an internal compiler inconvenience.

The same mismatch appeared in let-with shorthand syntax. A program such as:

```
let var = {a,b}
let var.a = var.a + 1
in var.a + var.b
```

was not accepted, even though the corresponding explicit record update was well-defined:

```
let var = var with a = var.a + 1
```

A related inconsistency existed for sections. Futhark supported simple field sections such as `(.f)` and simple index sections such as `(.[0])`, but not their mixed compositions. Thus, while ordinary expressions could describe a path like `x[0].f`, there was no section syntax corresponding to the projection function `\x -> x[0].f`.

One would think that such syntactic sugar could be desugared away immediately. Here, however, that would not suffice. If a mixed path update were expanded too early into nested array and record updates, the compiler would recreate precisely the consumption-sensitive intermediate form that causes the original problem. The unified syntax therefore had to be represented explicitly through the frontend and type-checking phases.

The central objective of this project was therefore to unify arrays and records under a common path-based model, satisfying several goals:

- mixed paths should be representable directly in the syntax and AST,
- update syntax, let-with shorthand, and sections should reuse the same underlying mechanism,
- the frontend should preserve these constructs until after type checking,
- the design should preserve uniqueness and consumption correctness,
- the implementation should not require changes to the compiler IR.

4 Unified Path Representation

The key design change is that nested update and projection structure is now represented explicitly as a list of path steps. Rather than distinguishing between array updates and record updates at the top level, the compiler stores a path as a sequence whose elements describe how to move from one level of a value to the next.

This path-based view is pretty intuitive. Expressions such as `[0].f[1:3]` are naturally understood as paths, not as unrelated syntactic fragments. By making this explicit in the internal representation, the compiler can support mixed forms uniformly.

The shared representation is captured by the datatype `UpdateStep`. In the source AST, it has two constructors:

```
data UpdateStep f vn
  = UpdateStepSlice (SliceBase f vn)
  | UpdateStepField Name
```

An `UpdateStepSlice` represents array indexing or slicing, while an `UpdateStepField` represents record field access. A complete path is then simply a list of such steps.

This representation is used by the ordinary update expression:

```
Update (ExpBase f vn) [UpdateStep f vn] (ExpBase f vn) ...
```

by let-with shorthand through:

```
LetWith ... [UpdateStep f vn] ...
```

and by sections through:

```
UpdateSection [UpdateStep f vn] ...
```

Because a path is now simply a list of steps, mixed forms arise naturally. For example:

- `[0].f` becomes a slice step followed by a field step,
- `f[1]` becomes a field step followed by a slice step,
- `[0].a[1:3]` becomes slice, field, then slice.

This makes the representation both simpler and more expressive. No dedicated AST constructor is needed for each possible nesting pattern.

4.1 Grammar Refactoring

The parser change mirrors this new representation closely. Before this work, array and record updates were parsed separately. In particular, the parser had distinct productions for indexed array updates and record field updates, as well as separate let-with and section forms for arrays and records.

After the refactoring, update paths are parsed uniformly as lists of `UpdateStep`s. The key parser nonterminals now are:

- `Update`, which parses a complete update path,
- `LetUpdate`, which parses the restricted path syntax used in let-with shorthand,

- `SectionUpdate`, which parses path sections.

All of these produce the same representation: a list of `UpdateStep`s.

This means that expressions such as the following are all handled by the same path machinery:

- `xs with [0].f = x`
- `r with f[1] = x`
- `let xs[1].f = y`
- `(.[0].f)`
- `(.a[1:3])`

Importantly, this refactoring did not introduce significant grammar ambiguities. The change mostly replaced several special cases with one common path-oriented structure.

4.2 From Split Surface Forms to One Frontend Representation

Before this work, the split between arrays and records was visible throughout the frontend. The parser distinguished between array updates and record updates, let-with shorthand for arrays and records, and separate section forms for field projection and indexing. After the refactoring, these constructs are all expressed in terms of the same path datatype.

This is significant because it means the unification is not merely notational. The frontend can now preserve the mixed path as a first-class structured object, which is exactly what is needed for correctness during consumption checking and desugaring.

5 Implementation

A notable aspect of this work is that it required no changes to Futhark's core intermediate representation. The entire implementation lives in the frontend language: parsing, type checking, normalization, monomorphisation, and internalisation.

5.1 Parser and Source AST

The parser is responsible for constructing mixed paths as lists of `UpdateStep`s. These paths are then carried through the source AST and processed uniformly by later frontend phases. In particular, ordinary `with` expressions, let-with shorthand, and sections all rely on the same underlying representation.

Before the change, the frontend had separate constructs for record updates, field-based let-with syntax, and field or index sections. After the change, these

are replaced by a smaller set of path-based constructs centered on `Update`, `LetWith`, and `UpdateSection`.

5.2 Normalization and Desugaring

The full normalization pass rewrites shorthand forms into more explicit source expressions and ensures that non-trivial subexpressions are named through let-bindings. This is also where the implementation carefully preserves the evaluation discipline needed by uniqueness.

For updates, normalization processes the path steps before the replacement value, and the replacement value before the source expression. This ordering is intentional and must match the assumptions made by consumption checking. It is particularly important in cases where path expressions and replacement values interact through uniqueness.

Sections are simpler. Before this work, field sections and index sections were desugared separately. After unification, they are both represented as `UpdateSection` and desugared through the same path-based mechanism. Conceptually, a section such as:

```
(.[0].f)
```

is transformed into a lambda equivalent to:

```
\x -> x[0].f
```

by traversing the path and constructing the corresponding projection body.

5.3 Consumption Checking

The type checker and consumption checker are where the unified representation becomes semantically important. Mixed updates must remain explicit through these phases, because desugaring them too early into nested updates would recreate the consumption problems described earlier.

For both ordinary `Update` expressions and `LetWith`, the consumption checker processes the path steps first, then the replacement value, and only then the source expression. For indexed updates, it also performs overlap checks between the source and the replacement value, reflecting the fact that these updates are the ones that may later be compiled using memory reuse.

5.4 Internalisation

Internalisation is where the source-language path representation is lowered to existing core operations.

Before the unification, internalisation had separate special cases for array updates and record updates. Array updates were lowered through slice-based logic, while record updates were lowered structurally by replacing the relevant

field components. After the change, the general case is handled by a single path-lowering procedure that traverses a list of `UpdateSteps`.

The internaliser still handles trivial one-step cases directly, but the general mixed-path case is lowered by recursively following the path. Field steps decompose the current value structurally, while slice steps compute the indexed subvalue, recursively update it, and then write the updated result back to the original source structure.

This design allows expressions such as:

```
xss with [0].a[1] = v
```

to be treated as one structured update, rather than as a nesting of unrelated record and array updates. The source value is evaluated once, the path is traversed, and the result is lowered to the existing core update operations.

5.5 Monomorphisation of Sections

Section desugaring follows the same pattern. Before this work, projection sections and index sections were desugared by separate functions. After the change, monomorphisation uses a single desugaring function for path sections.

Again, using this model, the same path representation that is used for updates is also used to build the body of the corresponding projection lambda. It also means that section unification required no new semantic machinery beyond the path abstraction already introduced for updates.

6 Semantic Correctness and Evaluation Order

The unification of paths for updates and sections is not only a syntactic change. In Futhark, updates interact directly with uniqueness typing, and this means that evaluation order matters. A correct implementation must therefore do more than parse mixed paths such as `[i].f[j]`: it must ensure that the resulting program respects the language's consumption semantics and does not duplicate expressions whose evaluation is semantically significant.

A `with` expression returns a new value semantically, but indexed array updates are checked and lowered in a way that permits memory reuse when uniqueness allows it. This is why mixed path updates cannot be treated as arbitrary syntactic compositions. The source expression, path expressions, and replacement value may all interact through consumption.

6.1 Ordering of Path, Value, and Source

A crucial implementation detail is that update expressions are not treated as unordered collections of subexpressions. In both normalization and consumption checking, the components of an update are processed in a specific order: path steps first, then the replacement value, and finally the source.

This ordering ensures that any indexing and slicing expressions embedded in the path are processed before a later expression may consume the values they depend on. It also ensures that normalization and consumption checking agree on the structure that is being analyzed.

6.2 Avoiding Use-After-Consume

A representative example is the following:

```
def consume (xs: *[]i64) : i64 = xs[0]

def main (xs: *[]i64) (ys: *[]i64) =
  ys with [xs[1]] = consume xs
```

Here, the index expression `xs[1]` reads from `xs`, while the replacement expression `consume xs` consumes `xs`. If the replacement expression were processed first, the later attempt to determine the update index would become invalid.

The implementation avoids this by processing the path before the replacement value in a way that is consistent across normalization and consumption checking.

6.3 Avoiding Duplication of the Source Expression

A second important example is:

```
def consume 't (xs: *[]t) : []t = xs

def main (xs: *[](i64, i64)) =
  consume xs with [0].0 = 2
```

Here, the source expression of the update is itself consuming. A naïve lowering of the mixed path could duplicate that source expression while traversing the path. The implementation avoids this by ensuring that the source expression is evaluated once and then lowered through the path structure, rather than being recomputed for each step.

This is particularly important in internalisation, where mixed paths are lowered recursively. The path traversal operates on the already-evaluated source representation, not on the original source expression syntax.

6.4 Sections: A Simpler Case

Sections do not pose the same semantic difficulties. Since they are desugared to projection lambdas, they do not involve source consumption or array-update memory reuse. They benefit from the same path abstraction, but without the need for the consumption-sensitive lowering required for updates.

7 Overall Compiler Impact

The compiler impact of this work is primarily a simplification of the frontend. Previously, array updates, record updates, and their corresponding section-like forms were represented and processed separately. After unification, they all reuse the same path abstraction.

This reduces duplication in the parser and source AST, and it also provides a clearer structure for frontend passes such as normalization, monomorphisation, and type checking. At the same time, the work deliberately avoids extending the core compiler IR. The implementation is therefore a net simplification: the surface language becomes more expressive, while the compiler keeps the same core update machinery.

8 Testing and Validation

The implementation was validated through regression tests covering mixed updates, let-with shorthand, and generalized sections. Representative examples include:

- mixed update paths such as `xs with [0].f = x` and `r with f[1] = x`,
- let-with shorthand such as `let xs[1].f = y`,
- consecutive indexing such as `let xss[0][1] = 0`,
- mixed field-and-slice updates such as `let xs.f[1:3] = [y, 0]`,
- mixed path sections such as `(.[0].f)` and `(.a[1:3])`,
- uniqueness-sensitive cases involving consumption in path and source expressions.

These tests confirmed both that the new syntax is accepted and that the frontend handles all relevant and sensitive cases correctly.

9 Experimental Evaluation

Since the introduced constructs are primarily a frontend unification and an extension of existing syntax, their effect is mainly on expressiveness and usability rather than on runtime performance. In particular, programs written using the older forms that existed before these additions can in principle be rewritten to use the new unified path syntax.

A systematic rewriting study was not carried out as part of this project. Doing so would mainly demonstrate that existing update patterns can be expressed in a simpler and more direct way with the new constructs, rather than showing any performance gains. Because the new forms are largely syntactic sugar over

behaviour that was already available, no performance gains are expected solely from rewriting old programs into the new syntax.

For this reason, the efforts in this project focused on correctness through integration testing (accompanying all the code changes delivered), rather than benchmarking rewritten programs.

10 Conclusion

This project unified array and record paths in Futhark through a common `UpdateStep` representation. The result is a more uniform treatment of ordinary updates, let-with shorthand, and section syntax, all of which can now express mixed paths directly.

The main benefit is not merely syntactic convenience, but a better fit between surface syntax and the semantics enforced by Futhark's uniqueness system. Mixed updates that were previously awkward or impossible can now be treated as single structured frontend expressions. At the same time, the implementation requires no changes to the core IR and instead simplifies the frontend by replacing several parallel mechanisms with one shared abstraction.

References

[1] DIKU. The futhark programming language. `https://futhark-lang.org/`, 2026. Accessed: 2026-03-22.

[2] Futhark GitHub Repository. Placeholder: Github issue discussing updates to arrays inside records. `https://github.com/diku-dk/futhark/issues/1160`, 2026.

[3] Aziz Rmadi. Adding support for shortened in-place record update. `https://github.com/diku-dk/futhark/pull/2362`, 2026.

[4] Aziz Rmadi. Unification of records and arrays for updates. `https://github.com/diku-dk/futhark/pull/2369`, 2026.

[5] Aziz Rmadi. Unification of sections. `https://github.com/diku-dk/futhark/pull/2385`, 2026.