Compiling Pattern Matching to In-Place Modifications*

Paul Iannetta
Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP
France
paul.iannetta@ens-lyon.fr

Laure Gonnord
University of Grenoble Alpes, Grenoble INP, LCIS & LIP
France
laure.gonnord@lcis.grenoble-inp.fr

Gabriel Radanne
Inria, EnsL, UCBL, CNRS, LIP
France
gabriel.radanne@inria.fr

Abstract

Algebraic data types and pattern matching are popular tools to build programs manipulating complex datastructures in a safe yet efficient manner. On top of its safety advantages, compilation techniques can turn pattern matching into highly efficient deconstruction code for immutable use cases.

Conversely, high-performance datastructures and languages prefer to leverage (controlled) mutations to maximize time and memory efficiency. Algebraic data types provide a natural framework to efficiently describe *in-place* transformations as rewrite rules. Such representation could take advantage of parallelism opportunities that appear in tree-like structures.

We present early steps towards a new technique to compile pattern matching as parallel in-place modifications of the underlying memory representation. Towards this goal, we combine the usual language approach which is common in pattern-matching compilation with tools from the polyhedral model, which is commonly used in high-performance code generation to output efficient C code. We present our formalism, along with a prototype implementation.

CCS Concepts: • **Software and its engineering → Compilers**; **Source code generation**; • **Theory of computation → Rewrite systems**; **Concurrency**.

Keywords: compilation, ADT, rewriting, code generation

ACM Reference Format:
Paul Iannetta, Laure Gonnord, and Gabriel Radanne. 2021. Compiling Pattern Matching to In-Place Modifications. In *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '21), October 17–18, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3486609.3487204

*This work was partially funded by the French National Agency of Research in the CODAS Project (ANR-17-CE23-0004-01)

GPCE '21, October 17–18, 2021, Chicago, IL, USA

ACM ISBN 978-1-4503-9112-2/21/10.
https://doi.org/10.1145/3486609.3487204

1 Introduction

Algebraic datatypes are used for a wide variety of purposes such as representing context (render trees, ray tracing, abstract syntax trees), or tree-based datastructures (AVL [1], red-black or B-Trees [10]). These structures are widely used in immutable functional languages, but also in high-performance contexts which generally rely on in-place mutations to modify the corresponding term.

Traditional techniques to optimize pattern matching [15] relies on immutability and don't leverage any parallelism. More recent work [9, 12, 13, 18, 20] optimize terms used as container, notably by improving the parallelism and data-locality of traversals. However, these work severely limits the possibility of changing the *structure* of terms by only allowing mutations of values embedded in the structure.

The *memory representation* of terms is also a critical topic in obtaining good performances. This has been tackled both generally by providing optimized layouts for algebraic data-types [20], and on particular cases through cache oblivious algorithm for high performance datastructures [4]. These representations are seldom used in functional programming due to their incompatibility with the high degree of sharing present in immutable datastructures [16]. However, they offer good data-locality which modern processors can take advantage of, notably through cache-oblivious and cache-aware algorithms, and lend themselves better to parallelism.

Many such representations exist (see. [5] for an overview). In the rest of this article, we assume that each subterm in a term is laid out in a set of layers addressable in constant time from the root of the subterm. Layers are considered indivisible and can be copied in a cache-friendly manner. This corresponds, for instance, to the Breadth-First Layout where elements of a term are laid out in an array in their breadth-first ordering. In this context, terms are a subpart of a *support*, which spans the whole underlying array.

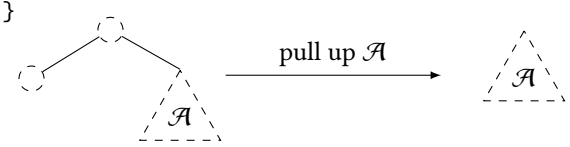
Contributions In this work, we attempt to optimize *structural transformations* on terms of algebraic data-types by exhibiting parallelism. We describe a new approach to compile pattern matching as in-place modifications on flattened terms of algebraic data-types. For this purpose, we define Rew, a core language to rewrite algebraic terms (Section 2). We propose a sequence of algorithms to derive dependencies and then schedule operations on subterms (Section 3). We then emit the appropriate sequential code (Section 4). The approach is implemented as a prototype.

2 From Pattern Matching to Memory Moves

We now describe the Rew core language. Rew allows to describe Algebraic Data Types and simple rewriting rules on them. For conciseness, we omit the type system and dynamic semantic for this simple system (see [11, 15] for exhaustive descriptions of rich pattern languages) and focus on the compilation. We start with our running example before proceeding with the definition of the language.

Example 1 (Simple example of Rew program)**.** *As a running example, we consider the code below which defines the algebraic data type for binary trees containing integers and a transformation* `pull-up` *which pulls the right subtree up, also represented graphically. A* Rew *function is similar to explicitly typed functions using pattern-matching, but defines a* `rewrite`*. Here, expressions only allow constructors and variables.*

```
type tree = Empty | Node (tree,int,tree)
pull_up (t : tree) : tree = rewrite t {
  | Node(a,i,Node(b,j,c)) -> Node(b,j,c)
  | Empty -> Empty
}
```



2.1 The Rew Language

In the rest of this article, we use the following notations. We denote types t, expressions e and patterns p. We use overbars for *syntactic* lists (for instance $\overline{p}$ is a list of patterns) and overarrows for vectors (for instance $\vec{k}$). The syntax of Rew is given in Definition 1 and demonstrated in Example 1.

Types are either built-in or user-defined. We write Scalar the set of built-in scalar types such as integers, floating-points numbers, etc. For simplicity, we only consider non-parametric types, but our setting easily extends to parametric types, as long as the code is monomorphic (or specialized before-hand). A type declaration is composed of several constructors (written Constr $\in$ `Constructors`), each with several parameters. Expressions (resp. patterns) contain variables (resp. bindings) and constructors. A clause is a pair of a pattern and an expression. A program is a list of clauses.

Typing for such a language is a simple subset of typing in much richer languages [11]. As a single restriction, we consider that variables can never be re-defined. In the rest of this article, we assume the existence of an operator $\mathrm{Type}(c, x)$ which gives the type of x in the context of a clause c (i.e., including bindings induced by the pattern part).

Definition 1 (Syntax of the rewrite language Rew)**.**

$$t ::= \overline{\mathrm{Constr}(t_0, \ldots, t_n)} \mid t_0 \in \mathrm{Scalar} \qquad \text{(Types)}$$
$$p ::= x \in \texttt{Vars} \mid \mathrm{Constr}(p_0, \ldots, p_n) \qquad \text{(Patterns)}$$
$$e ::= x \in \texttt{Vars} \mid \mathrm{Constr}(e_0, \ldots, e_n) \qquad \text{(Expressions)}$$
$$c ::= p \to e \qquad \text{(Clauses)}$$

In the rest of this article, we compile each clause independently. As such, we now only consider a single clause $p \to e$, for which we will compute a set of *elementary operations* (copies) that should be performed. We propose an approach in two steps, described in the rest of the Section. We also derive a notion of *dependance relation* that captures a partial ordering for these operations (a read of a term should be performed before its use).

2.2 Characterizing Coarse Grain Memory Moves

The coarse-grain decomposition captures the structural memory moves to be performed: for instance, in the clause `Node(a,i,Node(b,j,c)) -> Node(b,j,c)`, we need to move the subterm corresponding to the variable b to the first field of the constructor `Node`. For this purpose, Definition 2 presents the notions of *location* of a subterm in a term and of *moves* of a subterm from a location to another. A field is composed of an index and a type, written $.i/t$. A subterm location is a (potentially empty) list of fields, or an *external* location (for instance, an argument of the surrounding function). A subterm move is a variable binder annotated with its source and destination locations.

Definition 2 (Subterm locations and movements)**.**

$$\mathrm{f} ::= .i/t \qquad i \in \mathbb{N} \qquad \text{(Field)}$$
$$\ell ::= \overline{\mathrm{f}} \mid \mathrm{External} \qquad \text{(Location)}$$
$$m ::= (\!| x : t \mid \ell \to \ell' |\!) \qquad \text{(Move)}$$

We can compute the moves of a clause $p \to e$ through a traversal, as defined by the Moves function below. We assume the existence of the helper functions `Vars`, which gather all the variables of an expression or a pattern, and $\mathrm{Locs}(a, x)$ which obtain all the positions at which x appears in the pattern or expression a. As additional restriction, a variable only appears once in a pattern (but potentially several times in an expression).

$$\mathrm{Moves}(p \to e, t) = \left\{ (\!| x : t_x \mid \ell_p \to \ell_e |\!) \;\middle|\; \begin{array}{l} x \in \texttt{Vars}(p) \cup \texttt{Vars}(e) \\ t_x = \mathrm{Type}(p \to e, x) \\ \ell_p = \begin{cases} \mathrm{Locs}(p, x) & \text{if } x \in \texttt{Vars}(p) \\ \emptyset & \textit{otherwise} \end{cases} \\ \ell_e \in \begin{cases} \mathrm{Locs}(e, x) & \text{if } x \in \texttt{Vars}(e) \\ \emptyset & \textit{otherwise} \end{cases} \end{array} \right\}$$

Example 2. *On the first clause in Example 1, we obtain the following moves:*

$$(\!| j : int \mid .2/tree.1/int \to .1/int |\!) \qquad (\!| i : int \mid .1/int \to \emptyset |\!)$$
$$(\!| b : tree \mid .2/tree.0/tree \to .0/tree |\!) \qquad (\!| a : tree \mid .0/tree \to \emptyset |\!)$$
$$(\!| c : tree \mid .2/tree.2/tree \to .2/tree |\!)$$

Given a set of movements on subterms, we must decide if some movements should be done before the other. For instance, in Example 2, the moves of b and j must be executed

before the move of c, as this last move will erase the location $.2/tree$, which originally contains b and j. For this purpose, we define the notion of *conflict* between two locations.

Definition 3 (Conflict between locations)**.** We say that ℓ and ℓ' are in *conflict*, written $\ell \bowtie \ell'$, if ℓ is prefix of ℓ' or ℓ' is prefix of ℓ. External locations are never in conflict with anything. If $\ell \bowtie \ell'$, we write $\mathrm{diff}(\ell, \ell')$ the extra suffix.

2.3 Fine Grain Decomposition into Memory Moves

The moves we have shown so far operates on *subterms*, i.e. a given location and all its descendants. This coarse-grained approach causes two issues. First, it induces more conflicts than necessary, making the "happens before" less precise. Indeed, any subterm will trigger a conflict, even if other part of the term could be modified independently. Second, it means moves will easily conflict with themselves, as is the case of the move of c in Example 2. In particular, it is not clear at this stage how we could implement the move of c.

To alleviate these problems, we leverage the memory representation mentioned in Section 1 by decomposing each subtree move into a collection of memory moves on *paths*. Definition 4 gives the notion of *path* π which extends locations with repetitions indexed by an iteration variable k. Path also include wildcards φ which correspond to any field. These wildcards allow separating the representation by *layers*: φ^k is the k^{th} layer of a subterm. Memory moves are moves operating on memory paths. Paths correspond to regular expressions on locations without alternatives and of star height 1. Matching is immediate by treating named repetitions as Kleene stars.

Definition 4 (Paths and memory movements)**.**

$$\begin{aligned} k &\in \texttt{ItVars} && \text{(Iteration variables)}\\ \pi &::= \ell \,.\, \pi \mid \ell^k \,.\, \pi \mid \varphi^k \mid \varepsilon && \text{(Path)}\\ m_\pi &::= (\!| \pi \to \pi' |\!) && \text{(Memory Move)} \end{aligned}$$

The Atomize function aims to decompose subtree movements (where the iteration is implicit) into memory movements with explicit iteration. On the way, it eliminates spurious "self-conflict", i.e., rules whose source and destinations are in conflict and reveal potential parallelism for later phases. We first look at the output of Atomize on an example.

Example 3. *In Example 2, we obtained the subterm moves for the first clause of Example 1. By application of* Atomize*, we obtain the memory movements. For convenience, we sometimes shorten paths of the form $f.f^k$ as f^{k+1}.*

$$\begin{aligned} (\!| .0/tree.\varphi^{k_0} &\to External |\!) && \text{(a)}\\ (\!| .1/int &\to External |\!) && \text{(i)}\\ (\!| .2/tree.0/tree.\varphi^{k_1} &\to .0/tree.\varphi^{k_1} |\!) && \text{(b)}\\ (\!| .2/tree.1/int &\to .1/int |\!) && \text{(j)}\\ (\!| (.2/tree)^{k_2+2} &\to (.2/tree)^{k_2+1} |\!) && \text{(c)}\\ (\!| (.2/tree)^{k_2+2}.0/tree.\varphi^{k_3} &\to (.2/tree)^{k_2+1}.0/tree.\varphi^{k_3} |\!) && \text{(c0)}\\ (\!| (.2/tree)^{k_2+2}.1/int &\to (.2/tree)^{k_2+1}.1/int |\!) && \text{(c1)} \end{aligned}$$

The rules (i) and (j) correspond directly to the subterm moves: Since those terms are scalar, they do not require any iteration. (a) and (b) correspond to moves on a and b. Since source and destination do not conflict, we simply copy each layer separately. $\ell \,.\, \varphi^k$ here denotes the k^{th} layers of the subterm anchored in ℓ and is used to copy a subterm layers by layers. The subterm move for (c), on the other hand, has conflicting source and destination and requires additional care. We decompose it into several memory moves, corresponding to climbing the "stair" of subterms along the direction $.2/tree$. This is schematized on Fig. 1, which represents the memory layout of a term of type tree, along with the new memory movements in bold arrows and the iteration directions in double arrows. The memory movement (c) is on the stair itself, while $(c1)$ and $(c0)$ correspond to all the potential subterms which are not *reached by the prime iteration direction $.2/tree$. Depending of whether such subterms are scalars or terms, we decompose them further into layers.*

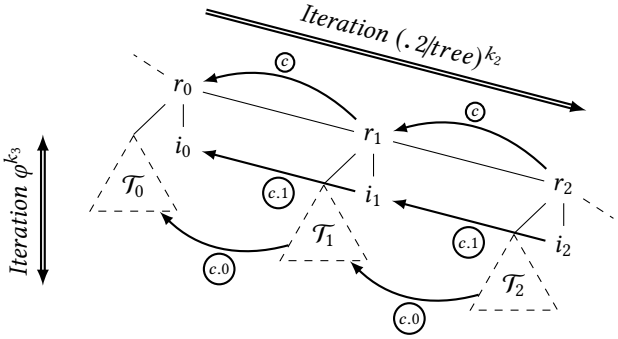


Figure 1. Memory movements (c), $(c0)$, $(c1)$ in Example 1.

Let us now define auxiliary functions used in Atomize.

Definition 5 (Type of a location)**.** The type of a non-external location is $\mathrm{Type}(\ell \,.\, x/T) = T$.

Definition 6 (Complement of a location)**.** Let a location ℓ and a type $t = \overline{\mathrm{Constr}(\overline{t_i})}$. We consider $\mathcal{F}$ the sets of all the fields $(.\, i/t_i)$ present in t. The complement of ℓ in t, written $\mathrm{Compl}(t, \ell)$, is the set of paths in t which are not ℓ. It allows us to inspect all subterms which are not in the prime iteration direction. We define:

$$\begin{aligned} \mathrm{Compl}(t, []) &= \{\}\\ \mathrm{Compl}(t, .\, i/t'.\, \ell) &= (\mathcal{F} - .\, i/t') \cup \{.\, i/t'.\, \ell' \mid \ell' \in \mathrm{Compl}(t', \ell)\} \end{aligned}$$

The complete definition of Atomize is shown in Algorithm 1. The first two cases are simple: identity moves are removed, and scalar moves are kept as-is, as they are non-recursive. The treatment of subterm moves depend if they have a self-conflict. If they do not, we move each layer, which is schematized by the move $(\!| \ell_p \,.\, \varphi^k \to \ell_e \,.\, \varphi^k |\!)$. Note that all such moves are disjoint (neither sources nor destinations can overlap). This hints at the possibility of parallelizing such loop later on. In case of self-conflict, we decompose

this move further by choosing an iteration direction. The conflict gives us a natural choice: since one location is the prefix of the other, we use the extra suffix $\ell = \text{diff}(\ell_p, \ell_e)$ to direct the iteration. All the subterms not present along the iteration direction are then given by $\text{Compl}(\ell)$. We inspect these subterms and create new appropriate moves depending on their types. By construction of Compl, none of these moves overlap (with themselves nor with each other).

Every single move has a *domain*, which is the set of valid values that can be taken by its iteration variables, and computed as a function parametrized by a formal parameter N, that denotes an upper bound on the height of terms embedded in the underlying array support. The domain of a move is directly induced by the admissible lengths of its paths.

Definition 7 (Length of a path)**.** Given a path π, its *admissible length*, written $|\pi|$, is the length of any location that could match the given path. It is the linear form defined as:

$$|\ell\,.\pi| = |\ell| + |\pi| \quad |\varphi^k| = k \quad |\ell^k\,.\pi| = |\ell|.k + |\pi| \quad |\varepsilon| = 0$$

Definition 8 (Domain of a move)**.** We consider a move $m = (\!|\, \pi \to \pi' \,|\!)$. The domain of m is written $\mathcal{D}_m$ and defined

$$\mathcal{D}_m = \left\{ \vec{k} \;\middle|\; (0 \le |\pi|(\vec{k}) \le N) \wedge (0 \le |\pi'|(\vec{k}) \le N) \wedge (\vec{0} \le \vec{k}) \right\}$$

3 Memory Moves Scheduling

Now that we have a fine grain characterisation of memory moves, we need to *schedule* them in order to generate code. We propose a two-steps approach, classical in the HPC community: first, we compute a compact representation for (read/write) dependencies, and from this representation, use optimisation to compute logical dates compatible with these dependencies with the help of an optimisation algorithm.

3.1 Dependencies Computation

A dependency happens if two moves might potentially overlap. Since the language of expression in Rew is pure, the only dependencies are induced directly by the rewriting specification: the left hand side acts as "reads" and the right hand side as "writes" on the specified paths. Usually, there are two types of conflicts: read-write, and write-write. However, in our case, a write-write conflict would mean two moves have the same destination. Since locations are entirely determined by positions in the AST of the right hand side, this is syntactically impossible.

Therefore, we only consider read-write dependencies, i.e., between the source of a move and the destination of another move. This naturally give rises to an ordered "happens-before" relation on moves since the write should be done after the read. Furthermore, not only do we want a relation to indicate potential dependencies, but also *when* this dependency happens, in term of the iteration variables k. We thus annotate this relation with constraints on the ks.

Definition 9 ((R/W) Dependencies between moves)**.** We consider two moves $m = (\!|\, \pi_p \to \pi_e \,|\!)$ and $m' = (\!|\, \pi'_p \to \pi'_e \,|\!)$ The dependencies between (the source of) m and (the destination of) m' is written $Q_{(m,m')}$ and defined as all values of the source and destination iteration vectors for which the memory paths actually intersect (do not have an empty intersection). Given $\mathcal{L}(\pi)$ the set of locations of π, we have:

$$Q_{((\!|\, \pi_p \to \pi_e \,|\!),(\!|\, \pi'_p \to \pi'_e \,|\!))} = \left\{ \begin{pmatrix} \vec{k} \\ \vec{k'} \end{pmatrix} \;\middle|\; \exists \ell \in \mathcal{L}(\pi_p(\vec{k})) \cap \mathcal{L}(\pi'_e(\vec{k'})) \right\}$$

Computing $Q_{(m,m')}$ as a finite description such as a polyhedron is not immediate. Paths are not "usual" regular expressions since the ks are *symbolic*, not concrete integers. In practice, we can obtain an exact representation of $Q_{(m,m')}$ thanks to careful syntactic manipulations on paths.

Lemma 1. *$Q_{(m,m')}$ is a union of polyhedrons and computing its finite representation is decidable.*

Example 4. *(Domain and Dependencies) We now give the domain and dependencies of some memory moves from Example 3 We recall the moves* (b) *and* (c0)*.*

$$(\!|\, .2/tree.0/tree.\varphi^{k_1} \to .0/tree.\varphi^{k_1} \,|\!) \quad \text{(b)}$$
$$(\!|\, (.2/tree)^{k_2+2}.0/tree.\varphi^{k_3} \to (.2/tree)^{k_2+1}.0/tree.\varphi^{k_3} \,|\!) \quad \text{(c0)}$$

The domains are computed from the length of the paths. Given the formal parameter N, we have:

$$\mathcal{D}_b = \{k_1 \mid 0 \le k_1 \wedge k_1 + 2 \le N\}$$
$$\mathcal{D}_{c0} = \{(k_2\ k_3) \mid 0 \le k_2 \wedge 0 \le k_3 \wedge k_2 + k_3 + 3 \le N\}$$

We also compute the following dependencies:

$$Q_{b,c0} = \{(k_1\ k'_2\ k'_3) \mid k'_2 = 0 \wedge k'_3 = k_1\} \qquad Q_{b,b} = \emptyset$$
$$Q_{c0,c0} = \{(k_2\ k_3\ k'_2\ k'_3) \mid k_2 + 1 = k'_2 \wedge k_3 = k'_3\}$$

We remark that the movement (b) *must be done before* (c0)*, as they both access the memory path* .2/tree.0/tree *and all its descendant. We also remark that* (c0) *has a self dependency, which induces an order of iteration along its first direction k_2. The direction k_3 doesn't have such an imposed order, which hints at later parallelism opportunities.*

Algorithm 1 Atomize(m)

$$\text{Atomize}(M) = \cup_{m \in M} \text{Atomize}(m)$$
$$\text{Atomize}((\!|\, x : t \mid \ell \to \ell \,|\!)) = \{\}$$
$$\text{Atomize}((\!|\, x : t \in \text{Scalar} \mid \ell_p \to \ell_e \,|\!)) = \left\{(\!|\, \ell_p \to \ell_e \,|\!)\right\}$$
$$\text{Atomize}((\!|\, x : t \notin \text{Scalar} \mid \ell_p \to \ell_e \,|\!)) \text{ when } \neg(\ell_p \bowtie \ell_e)$$
$$= \left\{(\!|\, \ell_p\,.\varphi^k \to \ell_e\,.\varphi^k \,|\!)\right\} \text{ where } k \text{ fresh}$$
$$\text{Atomize}((\!|\, x : t \notin \text{Scalar} \mid \ell_p \to \ell_e \,|\!)) \text{ when } \ell_p \bowtie \ell_e$$
$$= \left\{(\!|\, \ell_p\,.\ell_d^k \to \ell_e\,.\ell_d^k \,|\!)\right\}$$
$$\cup \left\{(\!|\, \ell_p\,.\ell_d^k\,.\ell \to \ell_e\,.\ell_d^k\,.\ell \,|\!) \;\middle|\; \begin{matrix} \ell \in \text{Compl}(\ell_d) \\ Type(\ell) \in \text{Scalar} \end{matrix}\right\}$$
$$\cup \left\{(\!|\, \ell_p\,.\ell_d^k\,.\ell\,.\varphi^{k_\ell} \to \ell_e\,.\ell_d^k\,.\ell\,.\varphi^{k_\ell} \,|\!) \;\middle|\; \begin{matrix} \ell \in \text{Compl}(\ell_d) \\ Type(\ell) \notin \text{Scalar} \end{matrix}\right\}$$

where ks fresh, $\ell_d = \text{diff}(\ell_p, \ell_e)$

3.2 Scheduling via Constraint Solving

At the end of the previous section, for each clause of our Rew program, we obtained a tuple $(\mathcal{M}, \mathcal{T})$ where:

- each move $m \in \mathcal{M}$ has an application domain $\mathcal{D}_m$.
- each pair of moves carries a "dependency constraint" $\mathcal{Q}_{m,m'}$.

Our objective is now to compute a valid schedule for the set of moves of the initial clause, *i.e.*, an order for the subcomputations. We adapt in this section an approach based on constraint solving used in polyhedral compilation [7, 8, 14] as well as in termination proofs [2, 6].

A schedule is a function that assigns *positive logical dates* to each move computation such that all dependencies are satisfied (a computation that depends on another one is done *strictly* after). This is captured in Definition 10.

Definition 10. Schedule constraints: A *schedule* for the graph $(\mathcal{M}, \mathcal{T})$ is a function $\rho : \mathcal{M} \times \mathbb{Z}^n \to \mathbb{N}^d$, from the graph vertices to $\mathbb{N}^d$, which is positive:

$$\vec{k} \in \mathcal{D}_m \Rightarrow \rho(m, \vec{k}) \geq \vec{0} \text{ (component-wise)} \quad \text{(Positivity)}$$

and whose values *stricly* increase (according to $\preceq_d$, the standard lexicographic order on integer vectors) at each edge $t = (m, m') \in \mathcal{T}$:

$$\mathcal{Q}_{m,m'}(k, k') \Rightarrow \rho(m, \vec{k}) \prec_d \rho(m', \vec{k'}) \quad \text{(Increasing)}$$

It is said *affine* if it is affine in the second parameter (the variables $\vec{k}$). If $d > 0$ the schedule is said to be multi-dimensional of dimension d.

Remark. Schedules can be parallel, indeed there is no constraint forcing two non conflicting moves to happen one before the other. The computation of a valid schedule might thus find equal dates for two different moves.

Searching for one dimensional schedules. First, we relax the increasing constraint (Increasing), for $d = 1$, into:

$$(\vec{k}, \vec{k'}) \in \mathcal{Q}_t \Rightarrow 0 \leq \rho(m', \vec{k'}) - \rho(m, \vec{k}) \leq \epsilon_t \leq 1$$

We now look for affine schedules, that is $\vec{c}, c_0$ such that $\rho(m, \vec{k}) = \vec{c}.\vec{k} + c_0$. Unfortunately, inlining this form leads to quadratic constraints $\vec{k} \in \mathcal{D}_m \Rightarrow \vec{\mathbf{c}}.\vec{\mathbf{k}} + c_0 \geq \vec{0}$. However, we can linearize these constraints using the Farkas lemma [19] (since $\mathcal{D}_m$ and $\mathcal{Q}_{m,m'}$ are polyhedra).

Lemma 2 (Constraints C)**.** *There exists a computable affine set of constraints C computed from $(\mathcal{M}, \mathcal{T})$ that exactly describes the set of admissible schedules.*

Finding a valid schedule consists in solving this set of constraints with an appropriate objective function (Algorithm 2). If a valid schedule exists, all ϵ_t are equal to 1. Otherwise, we have a partial schedule, that we will complete in the next section.

Multidimensional schedules. As all schedules are not of dimension 1, we use a greedy algorithm, described in Algorithm 3, similar to [6, 8, 14] where each component of the schedule ρ is constructed one after the other. At each loop it makes a call to compute1D(C, T). If it succeeds, the number of constraints still to be satisfied have strictly decreased and we can relaunch the procedure on the system without these constraints (Line 9). Otherwise, the procedure ends without concluding. Surprisingly, despite this *greedy* approach, this technique is proven complete (if the dependencies are exact [2]), thus it always gives an affine schedule.

Example 5. *(Schedule) From Example 4 we obtain the following schedules for (b), $(c0)$, $(c1)$:*

$$\rho(b) = (0, k_1) \quad \rho(c0) = (k_2 + 1, k_3) \quad \rho(c1) = (k_2 + 1, 0)$$

As $k_2 \geq 0$, this schedule successfully captures that movement (b) must be done before $(c0)$. Similarly, each $c0(k, k')$ is done before before $c0(k + 1, k')$, as expected.

4 Code Emission

The previous section provides us with a schedule $\rho(m, \vec{k})$ for each move of a given clause, the objective is now to generate a sequence of loop nests that will compute each (set of) moves in the order specified by the schedule, without forgetting any subcomputation. We use an algorithm inspired by previous works on code generation for the polyhedral framework [3, 17] depicted in Algorithm 4. It contains an inner procedure

Algorithm 2 `Compute1D`(C, T) where $T \subseteq \mathcal{T}$

```
1: MaximizeLP(∑_{t∈T} ε_t  on  C)                ▷ LP instance
2: if ∑_{t∈T} ε_t = 0 then
3:     return None                                ▷ No solution
4: else
5:     From the result, compute σ
6:     T_rem ← {t | ε_t = 0}                      ▷ Transitions that are not increasing
7:     return Some (σ, T_rem)
8: end if
```

Algorithm 3 `ComputeSchedule`$(\mathcal{M}, \mathcal{T})$

```
1: C ← ComputeConstraints(M, T)
2: i ← 0; T ← T
3: while T is not empty do
4:     ret ← compute1D(C, T);
5:     if ret = None then
6:         return None                            ▷ No affine schedule.
7:     else if ret = Some (σ, T_rem) then
8:         ρ_i ← σ                                ▷ σ is the i-th component of ρ
9:         T ← T_rem; i ← i + 1
10:    end if
11: end while
12: return Some(ρ)                                ▷ There is a i-dimensional ranking
```

Algorithm 4 `GenerateCodeForClause`$(\mathcal{D}, \rho, d)$)

procedure LoopGen$(i, \mathcal{P})$ ▷ dimension i
  if $i = d$ **then**
    return Moves$(\mathcal{P})$ ▷ Obtain the moves of $\mathcal{P}$
  else
    $\overline{L} \leftarrow \{P|_i \mid P \in \mathcal{P}\}$ ▷ Projection on dimension i
    $\overline{\mathcal{P}'} \leftarrow$ `MergePolyhedra`$(\overline{L})$ ▷ Generate distinct polyhedra with their associated moves.
    return $\left\{\text{LoopGen}(i+1, \mathcal{P}') \mid \mathcal{P}' \in \overline{\mathcal{P}'}\right\}$
      ▷ Decompose along the inner dimensions
  end if
end procedure
$\mathcal{P}_1 \leftarrow \{\ Im(\mathcal{D}_m, \rho_m) \mid m \in \mathcal{M}\ \}$
$r \leftarrow \text{LoopGen}(1, \mathcal{P}_1)$
Generate code from r

LoopGen which iterates recursively over the polyhedra to create a tree of nested loops. At recursive call i, LoopGen generates the sequence of loops corresponding to dimension i of the schedule ρ. At this point, we collect the projection of the polyhedra along dimension i which we partition and merge with the procedure MergePolyhedra. This gives us a list of polyhedra which delimit the inner loops strictly inside i, on which we recursively call LoopGen. When $i = d$, we emit the moves contained in the sub-polyhedra obtained by the recursive partitions. We initialize the set of polyhedrons with the set of images of $\mathcal{D}_m$ by ρ_m.

Example 6. *On the running example, we denote by (i, j) the iteration dimensions. The intial set $\mathcal{P}_1$ contains:*

$$Im(\rho_b, \mathcal{D}_b) = \{i = 0 \text{ and } 0 \le j \le N-2\}$$
$$Im(\rho_{c1}, \mathcal{D}_{c1}) = \{1 \le i \le N-2 \text{ and } j = 0\}$$
$$Im(\rho_{c0}, \mathcal{D}_{c0}) = \{1 \le i \le N-2 \text{ and } 0 \le j \le N-i-2\}.$$

The first recursive call gives $\overline{L}_1 = [P_1, P_2, P_3]$ where $P_1 : \{i = 0\}$, $P_2 = \{1 \le i \le N-2\} = P_3$ (projections on the first dimension i). The partition is then $\overline{\mathcal{P}'}_1 = [(P_1, b), (P_2, \{c0, c1\})]$ (we track the associated moves). The polyhedra P_1 and P_2 encode the outermost iterations. The two other recursive calls generate inner loops inside these "P_1, P_2 loops". From projections on j we obtain two polyhedra $P'_{1,2} = \{j = 0\}$ and $P'_{2,2} = \{0 \le j \le N-i-2\}$, and the final code:

```
for (i = 0 ; i <= 0 ; i += 1) // P1
  for (j = 0 ; j <= N-2 ; j += 1)
    ⟪ . 2/tree . 0/tree.φ^j → . 0/tree.φ^j ⟫  // b
for (i = 1 ; i <= N-2 ; i+= 1)  // P2
  for (j = 0 ; j <= 0 ; j += 1)  // P'1,2
    ⟪ (. 2/tree)^{i+1} . 1/int → (. 2/tree)^i . 1/int ⟫  //c1
  for (j = 0 ; j <= N - i - 2 ; j += 1) // P'2,2
    ⟪ (. 2/tree)^{i+1} . 0/tree.φ^j → (. 2/tree)^i . 0/tree.φ^j ⟫//c0
```

5 Future Work and Conclusion

We have presented the early work towards a framework to optimize in-place pattern matching on algebraic data types. So far, the framework we have presented is limited. We plan to extend it via a number of new features which are essential to improve its applicability.

An end-to-end implementation. We have implemented the first two steps (Sections 2 and 3) as a prototype[1]. The last step is so far done using standard tooling from the HPC community which computes loop nests based on a schedule. We aim to complete and extend this implementation with the features mentioned next.

Richer expression language. A richer expression language is essential to extend the scope of this work. We can in particular sketch the following extensions.

Functions on scalars and terms Our expression language can easily be extended to arbitrary functions on scalars such as arithmetic operations by replacing moves by instructions of the form $\pi \leftarrow f(\pi_0, \ldots, \pi_n)$. Since our framework transforms a function on terms into a set of memory movements. It can easily "inline" function calls by adding a prefix to its source and destination and merging its rules.

Self-recursion Inlining gives a way to handle limited self-recursion. Indeed, we can inline the function itself at the position of the recursive call. More concretely, given a clause of the form `Node(a,b) -> Node(f a,f b)`, it suffices to prefix all the moves by $(.\,0/tree|.\,1/tree)^k$, which spans all the subterms on which f is called. While this is not a fully general recursion scheme, it applies to functions such as `map` or transformations such as constant folding.

Richer pattern language. Guards, i.e., Boolean tests inside patterns, integrates well in our framework by equipping moves with conditions governing their applicability. It also allows compiling the whole pattern matching, including the choice of pattern to apply. The difficulty lies in emitting code properly factorizing the tests to preserve memory locality.

Better code generation. In this article, we only present sequential code generation. We have only scratched the surface of optimizations offered by the polyhedral model. Notably, our setup is ideally designed to produce parallel and vectorized code. We can also leverage the memory layout to improve memory locality when iterating through subterms.

References

[1] G. M. Adel'son-Vel'skii and E. M. Landis. 1962. An algorithm for organization of information. *Dokladi Akademia Nauk SSSR* 146, 2 (April 1962), 263–266.

[2] Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. 2010. Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In *Static Analysis Symposium*. Perpignan, France. https://doi.org/10.1007/978-3-642-15769-1

[1] https://github.com/Drup/adtr/tree/gpce21-camera

[3] C. Bastoul. 2004. Code generation in the polyhedral model is easier than you think. In *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.* 7–16. https://doi.org/10.1109/PACT.2004.1342537

[4] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Bradley C. Kuszmaul. 2005. Concurrent Cache-Oblivious b-Trees. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '05)*. Association for Computing Machinery, New York, NY, USA, 228–237. https://doi.org/10.1145/1073970.1074009

[5] Gerth Stølting Brodal, Rolf Fagerberg, and Riko Jacob. 2002. Cache Oblivious Search Trees via Binary Trees of Small Height. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'02)*. Society for Industrial and Applied Mathematics, USA, 39–48.

[6] Michael A. Colón and Henny B. Sipma. 2002. Practical Methods for Proving Program Termination. In *14th International Conference on Computer Aided Verification (CAV) (Lecture Notes in Computer Science)*, Vol. 2404. Springer Verlag, 442–454.

[7] Paul Feautrier. 1992. Some Efficient Solutions to the Affine Scheduling Problem, I, One-dimensional Time. *International Journal of Parallel Programming* 21, 5 (October 1992), 313–348.

[8] Paul Feautrier. 1992. Some Efficient Solutions to the Affine Scheduling Problem, II, Multi-dimensional Time. *International Journal of Parallel Programming* 21, 6 (December 1992), 389–420.

[9] Michael Goldfarb, Youngjoon Jo, and Milind Kulkarni. 2013. General Transformations for GPU Execution of Tree Traversals. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*. Association for Computing Machinery, New York, NY, USA, Article 10, 12 pages. https://doi.org/10.1145/2503210.2503223

[10] Gaston H. Gonnet and Ricardo Baeza-Yates. 1991. *Handbook of Algorithms and Data Structures in Pascal and C* (2nd ed.). Addison-Wesley Pub (Sd).

[11] Sebastian Graf, Simon Peyton Jones, and Ryan G. Scott. 2020. Lower your guards: a compositional pattern-match coverage checker. *Proc. ACM Program. Lang.* 4, ICFP (2020), 107:1–107:30. https://doi.org/10.1145/3408989

[12] Youngjoon Jo and Milind Kulkarni. 2011. Enhancing Locality for Recursive Traversals of Recursive Structures. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '11)*. Association for Computing Machinery, New York, NY, USA, 463–482. https://doi.org/10.1145/2048066.2048104

[13] Youngjoon Jo and Milind Kulkarni. 2012. Automatically Enhancing Locality for Tree Traversals with Traversal Splicing. In *Proceddings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*. Association for Computing Machinery, New York, NY, USA, 355–374. https://doi.org/10.1145/2384616.2384643

[14] R. M. Karp, R. E. Miller, and S. Winograd. 1967. The Organization of Computations for Uniform Recurrence Equations. *J. ACM* 14, 3 (July 1967), 563–590.

[15] Luc Maranget. 2008. Compiling pattern matching to good decision trees. In *Proceedings of the ACM Workshop on ML, 2008, Victoria, BC, Canada, September 21, 2008*, Eijiro Sumii (Ed.). ACM, 35–46. https://doi.org/10.1145/1411304.1411311

[16] Chris Okasaki. 1999. *Purely functional data structures.* Cambridge University Press.

[17] Fabien Quilleré, Sanjay Rajopadhye, and Doran Wilde. 2000. Generation of Efficient Nested Loops from Polyhedra. *International Journal of Parallel Programming* 28 (10 2000), 469–498. https://doi.org/10.1023/A:1007554627716

[18] Laith Sakka, Kirshanthan Sundararajah, Ryan R. Newton, and Milind Kulkarni. 2019. Sound, Fine-Grained Traversal Fusion for Heterogeneous Trees - Extended Version. *CoRR* abs/1904.07061 (2019). arXiv:1904.07061 http://arxiv.org/abs/1904.07061

[19] A. Schrijver. 1986. *Theory of linear and integer programming.* Wiley, NewYork.

[20] Kirshanthan Sundararajah and Milind Kulkarni. 2019. Composable, Sound Transformations of Nested Recursion and Loops. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 902–917. https://doi.org/10.1145/3314221.3314592