Compiling Trees : Combining Data Layouts and the Polyhedral Model

Paul Iannetta

Numéro National de Thèse: 2022LYSEN013

THÈSE DE DOCTORAT DE L'UNIVERSITÉ DE LYON

opéréé par

l'École Normale Supérieure de Lyon

École Doctorale N° 512

École Doctorale en Informatique et Mathématiques de Lyon

Discipline : Informatique

Soutenue publiquement le 02/05/2022, par:

Paul IANNETTA

Compiling Trees: Combining Data Layouts and the Polyhedral Model

Compilation des arbres : Représentation mémoire et modèle polyédrique

---

Devant le jury composé de:

CHARLES Henri-Pierre, Directeur de Recherche, CEA Grenoble	Rapporteur.
CLAUSS Philippe, Professeur des Universités, Université de Strasbourg	Rapporteur.
COLLANGE Caroline, Chargée de Recherche, Inria Rennes	Examinatrice.
KELLER Gabriele, Professeure, Utrecht University	Examinatrice.
GONNORD Laure, Professeure des Universités, Grenoble INP	Directrice de thèse.
RADANNE Gabriel, Chargé de Recherche, Inria Lyon	Co-encadrant.
MOREL Lionel, Maître de Conférences, Insa de Lyon	Co-encadrant.

To my parents,

ABSTRACT

The polyhedral model is a algebraic-based framework which enables efficient code optimization for computer-intensive programs which has been a prolific area of research since its inception. However, its strong assumptions about the shape of the control flow and memory accesses makes its application quite narrow in practice, even if many efforts have been done to extend it, mostly in the direction of more complex control-flow.

In this thesis, we propose to explore two research directions in order to deal with arbitrary control flow and memory pattern accesses, and target other data structures than arrays. Our first contribution is a semantic-based rephrasing of the framework that partially answers the first question and could sustain more in-depth research on the extension of the scope of the model to encompass "almost polyhedral programs". This contribution highlights the static program properties used by the polyhedral algorithms.

Our second contribution deals with algebraic data types, such as trees, which are, if not as common as arrays in scientific programs, ubiquitous in many algorithms. We propose a compilation scheme for algebraic data types laid out in memory according to a fixed layout. Memory movements are characterized "in the polyhedral model style" which enables optimized C code generation.

All in all, this thesis contributes to the on-going research of the polyhedral model on two points: by giving another point of view of polyhedral programs and by exploring how algebraic data types could reuse ideas from the framework.

RÉSUMÉ

Le modèle polyédrique est un framework algébrique qui permet une optimisation efficace des programmes de calculs intensifs. Ce domaine de recherche a été un domaine de recherche prolifique depuis sa création. Cependant, ses hypothèses fortes sur la forme du flot de contrôle et des accès mémoires rendent son application assez limitée en pratique, même si de nombreux efforts ont été faits pour l'étendre, principalement en autorisant des programmes avec des flots de contrôle plus complexes.

Dans cette thèse, nous proposons d'explorer deux directions de recherche afin de traiter un flot de contrôle et des accès mémoires de forme arbitraire, et de cibler d'autres structures de données que les tableaux. Notre première contribution est une reformulation sémantique du modèle polyédrique qui répond partiellement à la première question et qui pourrait servir de base mathématique pour des résultats plus fins sur les extensions possibles de la portée du modèle. Cette contribution met en évidence les propriétés statiques des programmes utilisées par les algorithmes polyédriques.

Notre deuxième contribution traite des types de données algébriques, tels que les arbres, qui, s'ils ne sont pas aussi courants que les tableaux dans les programmes scientifiques, sont omniprésents dans de nombreux algorithmes. Nous proposons un schéma de compilation pour ces types de données, en les représentant en mémoire selon une disposition fixe. Les mouvements en mémoire sont caractérisés dans le style du modèle polyédrique, ce qui permet une génération de code C optimisée.

En résumé, cette thèse contribue à la recherche en cours sur le modèle polyédrique sur deux points : en donnant un autre point de vue sur les programmes polyédriques et en explorant comment les types de données algébriques pourraient réutiliser les idées du cadre.

REMERCIEMENTS

Je tiens, en premier lieu, à remercier ma directrice de thèse : Laure Gonnord pour m'avoir fait confiance tout au long de ces quatre années (si l'on inclut mon stage de master). Je tiens aussi à remercier mes co-encadrants : Lionel Morel et Gabriel Radanne. Les remarques de Lionel étaient toujours avisées et, ont plus d'une fois contribués à simplifier ou éclairer mes raisonnements. Gabriel est arrivé vers le milieu de ma thèse. Son dynamisme et sa bonne humeur permanente ont su redonner un élan à ma motivation. Il a aussi apporté une nouvelle vision, qui a par la suite permis de généraliser mes premiers résultats sur les arbres aux types algébriques.

Merci à Philippe Clauss, et Henri Pierre Charles qui ont accepté d'être mes rapporteurs. Merci aussi à Caroline Collange et Gabriele Keller d'avoir accepté de faire partie de mon jury. Ainsi qu'à Benoît Meister avec qui j'ai eu l'occasion d'avoir des discussions très intéressantes à la conférence IMPACT.

Je tiens aussi à remercier toute l'équipe CASH (Compilation and Analysis, Software and Hardware), en particulier Matthieu Moy dont j'ai eu la chance de superviser une partie de ses TDs à Lyon 1. Sans oublier Ludovic Henriot, Christophe Alias et Yannick Zakowski avec qui j'ai aussi eu l'occasion d'avoir des discussions très enrichissantes. Je n'oublie pas non plus mes co-bureaux Alexis, Julien et Amaury sans qui cette thèse aurait très certainement été beaucoup plus ennuyeuse.

Enfin, je voudrais remercier Simon & Blandine avec qui j'ai passé des moments inoubliables ; Anaïs qui a toujours répondu présente quand je voulais parler de japonais ; l'association Espace-Lyon Japon et tous ses membres. Et pour finir je tiens à remercier mes parents, mon frère et ma sœur sans qui je n'aurais probablement jamais fait cette thèse.

CONTENTS

LIST OF FIGURES

INTRODUCTION

Context

In recent years, we have seen the emergence of new trends and new challenges to meet the ever-increasing demand for computing capacity. This demand comes from several actors. On the one hand, the scientific community that needs to run simulations of increasingly complex systems. On the other hand, private actors have demanding applications such as data analysis through machine learning or huge computations for graphical rendering.

These needs have resulted, in an upsurge of specialized boards, custom boards (ASICs) or reconfigurable boards (FPGAs) and in the progressive incorporation of frameworks created to drastically optimize the most common computational kernels into compilers.

From a language point of view, this thesis is mainly interested in structural optimizations made by optimizing compilers. Historically, structural memory optimizations have been performed on programs by means of *program transformations*, initially manually performed and now more amenable to automation. In our opinion, the more mature framework to deal with intensive computations and loop transformation is the so-called *polyhedral model*. This framework proposes to encode both computations and transformations in the same algebraic elements, namely, polyhedra, and thus enables clean and elegant definition of algorithms and tools.

The polyhedral model is now quite standard in the High Performance Computing (HPC) community, and is has notably demonstrated its applicability for data intensive regular computations named *kernels* such that matrix multiplication and *stencils*, which serve as basis on a large part of machine learning or scientific applications. It also have been implemented in production compilers such as gcc or LLVM.

Despite this success, the polyhedral model is not yet a complete solution for non regular kernels and, more importantly, for other data structures than arrays such that sparse matrices (heavily used in machine learning applications) or trees (present in automatic language processing). In this thesis, we propose contributions in this direction.

Objective

This thesis has mainly two objectives, both closely related to extensions of the polyhedral model inside compilers.

Historically, the polyhedral model's theory and algorithms have been defined on a subset of programs that were *syntactly* restricted. Although very common to define the frame of applicability of the framework, this representation of programs causes difficulties when it comes to properly defining extensions. An ideal polyhedral model would follow a *semantic* definition to better characterise its limits. Our first objective is to define such foundations.

The polyhedral model has shown its ability to express program optimizations for intensive computations on arrays. Although some existing work try to handle other complex data structures like trees, for which they also propose code optimization, few work exist to adapt or reuse ideas of the polyhedral model framework. Our second objective is to define a "treeshaped"-model for intensive tree computations that fit, potentially loosely, the polyhedral model.

Outline of the thesis

Instructions and Dependences Chapter 1 serves as an introduction and presents key concepts for analyzing and transforming programs, such as the distinction between instructions and instances. We present *inter-instruction* dependences that should be preserved by program transformations. Historically, propositions that conservatively compute these dependences were first proposed (Bernstein Conditions, the GCD test and the Banerjee's test). A bit later, two frameworks, namely the Omega Project and the Polyhedral Model, were developed onto a restricted class of programs for which they can exactly compute all dependences. We restrict our study to the Polyhedral model, present its input language and identify some of its limitations.

Revisiting the Polyhedral Model In Chapter 2, we focus on the polyhedral model and propose a new representation for its input language which address the limitations which we underlined in Chapter 1. We then use our input language to define dependences and revisit the polyhedral model's dependence analysis. We conclude this chapter by briefly presenting two other key parts of the classical polyhedral model workflow: instruction scheduling and code generation.

Complex Data Structures Chapter 3 begins by reviewing the literature related to how the polyhedral model deals with complex data structures (*i.e.*, data structures which are not directly arrays) such as sparse matrices. We then present algebraic data types with a focus on tree-shaped terms. Finally, we briefly review the different usage of trees and how they have been optimized in these different contexts.

Trees and their layouts Chapter 4 presents the general concept of memory layout for trees, that was historically mostly developed and used in the *cache-aware algorithms* community. We analyze representations in which trees are linearized and define layout functions for mapping tree positions to array indices. We will later use these mapping to transform programs on trees. We also analyze memory layouts characteristics: how the tree structure is kept despite the "flatness" of arrays, and to what extent layouts support repeated insertions and deletions.

Tarbres: AVL Trees as Arrays In Chapter 5, we build upon the basis laid out in Chapter 4 and tackle the optimizations of self-balancing trees, with AVL trees as illustration. We propose to use the breadth first layout to represent AVL trees (Tarbres) and build a library upon this representation. The key operation, namely, rotation, is decomposed into low-level operations that can be optimized and parallelized by reusing ideas from the polyhedral model. An experimental study was performed, that show the benefit of the optimizations on Tarbres over the layout induced by the traditional collection of pointers.

In-place transformations on ADTs Chapter 6 is the direct continuity of the previous chapter and proposes to address its shortcomings. We concluded Chapter 5 on only partially satisfying experimental results, because our manual transformations have limitations when it comes to perform more complex optimizations such that pipelining. We go one step further towards automatic code generation with the definition of the compilation process of a domain specific language (DSL) on trees. This DSL enables us to express in-place structural transformations such as rotations through pattern-matching rules for which we propose a complete "polyhedral-like" compilation process.

CHAPTER 1

INSTRUCTIONS AND DEPENDENCES

Let us begin with a very simple example Figure 1.1, the product of two polynomials. In this example, the only hard constraint is that the initialization at *s0* must be done first, or at least, each cell should be set to 0 before any other writes are performed. The computations in the second loop can be, theoretically, executed in any order and the work can be divided into many cores with no influence on the final result. In order for a computer to be able to figure out such information we will need to define the concept of *inter-instruction dependence* in Section 1.1.

```
for (int i = 0 ; i < m + n ; ++i)
c[i] = 0; /* s0 */

for (int i = 0 ; i < m ; ++i)
for (int j = 0 ; j < n ; ++j)
  c[i+j] += a[i] * b[j]; /* s1 */
```

Figure 1.1: Multiplication of two polynomials $C = AB$ with $A \in \mathbb{K}_m[X]$ and $B \in \mathbb{K}_n[X]$.

As the above example suggests, a huge portion of scientific computations relies on linear algebra which main objects: vectors and matrices, are often found in compute-intensive kernels as multidimensional arrays. Operations on matrices always involve loops, which is the core reason why loops have been under the focus of the optimizing-compilation community from the very beginning.

1.1 Definitions: Instructions and Instances

Definition 1. A *computer instruction* is the textual representation of a command that a computer can execute.

Definition 2. An *instance of a computer instruction* is the actual command that is executed on the computer. In particular, even though the same instruction may be executed multiple times, each time a different instance is executed.

Example 1. In C, `i = i + 1`, is an instruction which increases the value stored in variable i by one. Each time it will be executed the result will be different. Assuming that the variable `i` is initialized to 0, the first instance of the instruction will store 1 into `i`, the second will store 2 and so on.

Example 2. In C, `a[i] = b[i - 1] + 1` is an instruction which takes the value in the `(i - 1)`th cell of the array increments `b` by one and store it in the `i`th cell of array `a`. Each time it will be executed, the result will be dependent on the current value of variables `i`, `a` and `b`.

Example 3. In x86 assembly, `lodsb`[1] is an instruction which loads the content of registers `ds:si` into register `eax`. Each time `lodsb` is executed, depending on the value in registers `ds:si`, the value of register `eax` will be set accordingly.

In summary, an instruction is the textual representation of a command, and the instance of an instruction is the actual command executed. In summary, the instance of an instruction is an instruction and the point in time when it will be executed.

Definition 3. A *dependence* exists between two instructions u and v if those are trying to access, that is either read or write, or both, the same memory cell at different times. The order of the reads and writes determines the category of the dependence.

- A *flow dependence* (or read after write (raw)) occurs when there is a write later followed by a read.
- An *output dependence* (or write after write (waw)) occurs when there is a write later followed by a write.
- An *anti dependence* (or write after read (war)) occurs when there is read later followed by a write.
- An *input dependence* (or read after read (rar)) occurs when there is read later followed by a read.

1.2 Approximate Dependences

As the computation of loop-carried dependences is not decidable in the general case, there has always been a trade-off between soundness and completeness. Many early

[1] Intel 64 and IA-32 Architectures Software Developer's Manual. Vol 2A p.3-568

approaches such as Bernstein conditions [Ber66] or the Banerjee test [Ban04], and direction vectors, distance vectors, dependence vectors [Pad11] relied on methods which yield an approximation of the real dependences. However, those approximations might not be sufficient to be prove that a code transformation is valid.

In the case where we want to swap instructions, we first need to prove that this swap won't have any effect on the result of the program. A first idea would be to look at operations which are completely unrelated, that is, two operations which are accessing and/or writing different memory cells. This is the idea behind *Bernstein conditions*.

Bernstein Conditions. The *Bernstein conditions* [Ber66] are a sufficient condition for two instructions to be independent. Let u an v be two instructions and let $W(u)$, $W(v)$ and $R(u)$, $R(v)$ the sets of memory cells which are written (respectively read) by u and v. Then, those two instructions are said to meet the *Bernstein conditions* if:

$$W(u) \cap W(v) = R(u) \cap W(v) = W(u) \cap R(v) = \emptyset.$$

Example 4.
- Let u be $a = b + c$ and $d = e + f$. Those assignments are independent and meet the Bernstein conditions.
- Let u be $a = b + c$ and $d = a + e$. Those assignments are not independent and do not meet the Bernstein conditions.
- Let u be $a = a + 1$ and $a = a + 2$. Those assignments are independent but do not meet the Bernstein conditions.

Remark. On single assignment languages, and especially on the SSA form, if we suppose that u is executed before v, $W(u) \cap W(v)$ is always empty and $R(v) \cap W(u)$ is also empty because it is not possible to read a variable before it has been assigned a value. In this context the *Bernstein conditions* are simplified into: $R(u) \cap W(v) = \emptyset$.

Remark. On many languages, especially language used by intermediate representation, an instruction writes at most one location at a time. That is $W(u)$ and $W(v)$ are singletons.

Bernstein conditions can be applied if and only if we can figure out the set of addresses which are read and written by the instructions at end. This is not so much a problem when dealing with scalar values, however it turns out to be non-trivial when dealing with array accesses within loops like in Figure 1.2. The *Banerjee test* [Ban04] address this concern.

Let consider two instructions u and v. Both of them use values and/or modify values which are stored are in an array t. Let $t[f(\mathbf{i_u})]$ and $t[g(\mathbf{i_v})]$ be two memory accesses, the first one is performed in u while the second one is performed in v. $\mathbf{i_u}$ and $\mathbf{i_v}$, the iteration vectors (that is the value of $\mathbf{i}$ at the time the instance of the instruction is executed) associated with the instructions u and v.

The GCD test. If both f and g as defined in the previous paragraph are both affine with respect to iteration vectors there exist a_0, $\mathbf{a} = \{a_k \mid 1 \leq k \leq n_{\mathbf{i_u}}\}$, b_0 and $\mathbf{b} = \{b_k \mid 1 \leq k \leq n_{\mathbf{i_v}}\}$ where $n_{\mathbf{i_u}}$ and $n_{\mathbf{i_v}}$ are the loop depth at instruction u and v respectively, such that, $f(\mathbf{i_u}) = a_0 + \mathbf{a} \cdot \mathbf{i_u}$ and $g(\mathbf{i_v}) = a_0 + \mathbf{a} \cdot \mathbf{i_v}$. There is a dependence between u and v if $f(\mathbf{i_u}) = g(\mathbf{i_v})$. There is a dependence at depth ℓ if the ℓ first coordinates of $\mathbf{u_i}$ and $\mathbf{v_i}$ coincides. The *GCD test* tells us that there is a dependence at depth $\ell \leq \lfloor n_{\mathbf{i_u}}, n_{\mathbf{i_v}} \rfloor$ if

$$\left(\bigwedge_{k=1}^{\ell} a_k - b_k \right) \bigwedge_{k=\ell}^{n_{\mathbf{i_u}}} a_k \bigwedge_{k=\ell}^{n_{\mathbf{i_v}}} b_k \text{ divides } b_0 - a_0.$$

then, there is a dependence between u and v at level ℓ.

Example 5. If we apply the GCD test to the example of Figure 1.2a we see that: $(3 - 8) \wedge (11 - 3) = -5 \wedge 8 = 1$ divides 3. Therefore, there might be a dependence between *s0* and *s1* at depth 2. The existence of this dependence depends on the bound of the loop but the GCD test does not take them into account.

Example 6. If we apply the GCD test to the example of Figure 1.2b we see that: $(3 - 7) \wedge (11 - 5) = -4 \wedge 6 = 2$ does not divide 3, therefore we know for sure that there are no dependences between *s0* and *s1* at depth 2. Thus we can, for example, swap the two loops.

```
for (int i = ... ; i < ... ; i++)
  for (int j = ... ; j < ... ; j++)
    a[3*i + 11*j - 3] += ... /*s0*/

for (int i = ... ; i < ... ; i++)
  for (int j = ... ; j < ... ; j++)
    a[8*i + 3*j] += ... /*s1*/
```

(a) GCD test yields true ($i = 8n + 1$ and $j = 5n + 1$ for $n \in \mathbb{Z}$)

```
for (int i = ... ; i < ... ; i++)
  for (int j = ... ; j < ... ; j++)
    a[3*i + 11*j] += ... /*s0*/

for (int i = ... ; i < ... ; i++)
  for (int j = ... ; j < ... ; j++)
    a[7*i + 5*j - 3] += ... /*s1*/
```

(b) GCD test yields false (no solutions)

Figure 1.2: GCD test

The Banerjee test. The GCD test provides a quick way to decide whether a dependence may exist. However, even though the GCD test gives the existence of a solution it does not guarantee the solution found can happen. An improvement is provided by the *Banerjee conditions* which takes into account loop bounds when they can be obtained. However, the *Banerjee conditions* check the existence of a solution over the reals that is why it is always done in conjunction with the GCD test. Since the Banerjee test can be performed by a couple of additions and the verification of a condition it allows the elimination of false dependences without incurring any additional cost over performing only the GCD test.

1.3 Exact Computation of Dependences

A second generation of dependence-computation emerged in the end of the eighties with the raise of polyhedral techniques which are now at the root of what is called the polyhedral model. This time, rather than targeting general programs the approach focus on a subclass of programs called polyhedral programs (which will be addressed in the next section) on which the exact computations of the dependences is decidable.

Up until now, all the algorithms used to test dependences are not exact. This is because exact tests are expensive, and it was thought to be prohibitive to run algorithms that were not polynomial in a compiler. However, around the early nineties two frameworks computing exact dependences for a sub-class of programs emerged.

The Omega Project. This project [Pug91], developed by Bill Pugh and his students, aims at providing tools to manipulate sets of affine constraints over integer variables. Its core relies on Presburger arithmetics and provide not only yes/no answer but symbolic answers. It can remove redundant constraints, project on existentially quantified variables, simplify formulas and more. It is also a tool that can be used to analyze the dependences in a program (as long as those can be expressed with Presburger constraints) and generate code that will take those dependences into account. Although very expressive (in particular the use of Presburger formulas instead of (piecewise) quasi-affine selection trees [Fea91]), this framework did not convince the HPC community which showed more interest into the polyhedral model.

As of now, this project is mostly defunct but its contributions and expressivity has been merged into the Integer Set Library (isl) [Ver10] which implements everything and more from the omega projects, and which is now at the core of all the implementation of the polyhedral model, inside production compilers, Graphite [TCE+10] for gcc or Polly [GZA+11]

The Polyhedral Model. This framework aims at harnessing the inner parallelism present in polyhedral programs by describing programs as geometric objects. This is done in three steps *(i)* compute the dependences, *(ii)* compute a scheduling function (at this point optimizations such as tiling are performed), *(iii)* generate the final source code.

The array dataflow analysis [Fea91] and the Omega test [Pug91] proposed exact dependence analysis for loops with affine controls and array accesses. Both of these work rely on the ability to characterize the three conditions that define dependences as affine functions of syntax elements in the source program—loop iterators. The semantics of the target language are abstracted away and are assumed to provide the required properties.

The original exact dependence analyses were later extended to expand the scope of

the analysis, including `while` loops, non-affine `if` guards, and non-affine array accesses [BCF97, PW96, BPCB10, SV13]. In Fuzzy Array Dataflow Analysis [BCF97], the non-affine conditions are expressed as predicates encoded with additional parameters, whereas the extension to the Omega test [PW96] express them with uninterpreted function symbols. Exact analysis is possible in some cases, but these extensions require runtime checks or over-approximations in general.

The polyhedral model's success has led to two major integrations in state-of-the-art compilers. Graphite [TCE+10] has been to our knowledge the first polyhedral optimization framework usable for non-specialists (the algorithms are performted on the GIMPLE representation of gcc). More recently, Polly [GGL12] has been integrated to the LLVM's main branch [2]. It performs polyhedral optimizations to LLVM-IR, which is a low-level IR without high-level information such as loop iterators. The *semantic polyhedral regions* in a program are identified by searching for a *single induction variable* (with affine lower bounds and upper bounds) for each loop. Combined with additional analyses and transformations in LLVM, Polly can recognize program regions that are syntactically far from the canonical polyhedral loops in the original high-level specification.

Alphabets [RGK11] is an equational language that can be viewed as an intermediate representation for polyhedral compilers. The dependences[3] are expressed as affine functions of the domain indices, including the unbounded domain corresponding to `while` loops.

Apollo [SRC15] is a framework for runtime optimization that detects (affine) regularity in program behavior and applies polyhedral optimizations, speculating that the regularity persists. Runtime analysis enables code regions that cannot be determined to be polyhedral at compile-time to be found and optimized.

1.3.1 Polyhedral Programs

Before defining *polyhedral programs* we must clarify the concepts of *iteration space* and *data space* of a loop. A loop is some code that will produce a cycle in the control flow graph and can be achieved through different ways (for, while, gotos).

The *iteration space* of indexed loops is defined as the locus formed by the points reached by the indexes of the loop.

The *data space* of loops is defined as the locus formed by the coordinates of the arrays which are accessed within the loop.

In this end, a *polyhedral program* is a loop program whose iteration space and data space are polyhedra (even though most of the time we are more concerned by polytopes

[2] Polly moved to an LLVM infrastructure around April 2011 and was integrated as LLVM project in February 2012 (cf. polly.llvm.org)

[3] Alphabets can express such computations as data-dependent dependences like other work, but this is not the default/intended use in Alphabets.

(bounded polyhedra)) and there is an affine function between the iteration and space and the data space.

At first, those constraints were directly enforced on the source code of the programs (for loops with constant step), which is discussed in the next subsection (Section 1.3.2).

1.3.2 The input language of the polyhedral model and its limitations

The polyhedral model will be discussed in more details in Chapter 2 and this section will only present the input language on which the polyhedral model is based and its limitations.

The input language to the array data-flow analysis is made of affine loop nests, that is nested for loops where the loop conditions are affine expressions of the iteration variables. Moreover, the only variable which may appear in the statement enclosed by loops are either scalar or array variables and all array accesses must be affine functions of iteration variables. The product of polynomials is an example of a polyhedral program Figure 1.1.

The input language can be partly described by this BNF schema with infinitely-many rules:

$$\begin{aligned} \mathtt{s}^i ::= &\quad \mathtt{for}\,(\mathtt{int}\ \mathtt{x}_i = \mathtt{y}_i\ ;\ f_i(\mathtt{x}_0,\ \ldots,\ \mathtt{x}_i) < 0\ ;\ \mathtt{x}_i += \mathtt{z}_i)\ \{\ \mathtt{s}^{i+1}\ \} \quad | \quad \mathtt{t}^i \\ \mathtt{t}^i ::= &\quad \mathtt{t}^i; \mathtt{t}^i \quad | \quad \mathtt{I}^i \end{aligned}$$

where $\mathtt{y} = \{\mathtt{y}_0, \mathtt{y}_1, \cdots,\}$ is the initialization vector, and $\mathtt{z} = \{\mathtt{z}_0, \mathtt{z}_1, \cdots,\}$ is the constant increment vector, and statements $\mathtt{I}^i$ are of the form:

$$\mathtt{a}[g_i(\mathtt{x}_0, \cdots, \mathtt{x}_i)] = \ell(\mathtt{b}[h_i(\mathtt{x}_0, \cdots, \mathtt{x}_i)], \mathtt{c}[k_i(\mathtt{x}_0, \cdots, \mathtt{x}_i)])$$

where a, b and c are arrays which may be the same array, g_i, h_i, k_i are affine functions with i parameters with a return value within the bounds of the array they address, and ℓ is an arithmetic operation.

We can now compute the inter-instruction dependences of programs constructed from the above rules.

For example, let $\mathtt{I}^1$ and $\mathtt{I}^2$ be two operations defined as follows:

$$\begin{aligned} \mathtt{I}^1 :&\quad \mathtt{a}^1[g_i^1(\mathtt{x}_0, \cdots, \mathtt{x}_i)] = \ell^1(\mathtt{b}^1[h_i^1(\mathtt{x}_0, \cdots, \mathtt{x}_i)], \mathtt{c}^1[k_i^1(\mathtt{x}_0, \cdots, \mathtt{x}_i)]) \\ \mathtt{I}^2 :&\quad \mathtt{a}^2[g_{i'}^2(\mathtt{x}_0, \cdots, \mathtt{x}'_i)] = \ell^2(\mathtt{b}^2[h_{i'}^2(\mathtt{x}_0, \cdots, \mathtt{x}_{i'})], \mathtt{c}^2[k_{i'}^2(\mathtt{x}_0, \cdots, \mathtt{x}_{i'})]) \end{aligned}$$

$\mathtt{I}^1$ has to be executed before operations $\mathtt{I}^2$ if:

- $\mathtt{I}^1$ is executed before $\mathtt{I}^2$
- $\mathtt{b}^1[h_i^1(\mathtt{x}_0, \cdots, \mathtt{x}_i)]$ or $\mathtt{c}^1[k_i^1(\mathtt{x}_0, \cdots, \mathtt{x}_i)]$ is the same cell that $\mathtt{a}^2[g_{i'}^2(\mathtt{x}_0, \cdots, \mathtt{x}'_i)]$
- All accesses are within bounds.

Determining whether $\mathtt{I}^1$ is executed before $\mathtt{I}^2$ inherently relies on the textual form of the program in this formalism. Indeed, each instruction is identified by the current iteration vector of the surrounding loops, hence for instructions which are in the same loop we need to keep around syntactic information to know which instruction comes before.

The BNF formalization presented above has the disadvantage of only capturing a small part of the set of polyhedral programs: the programs which are perfect loop nests, with no more than one loop at each level. This issue arise from the fact that even though it is easy to give an intuitive definition of what polyhedral programs are in plain English it is very hard to formalize it. Which also incurs difficult when we want to equip polyhedral programs with a mathematical structure that can serve as a basis to prove statements on polyhedral programs.

The other problem of this source language is that the relation between the different objects such as loops, statements, operations, iteration vectors is loosely defined and relies on implicit semantic rules of the input language which impedes any mathematical reasoning on the language. That is why in the next chapter we will present our formalism for the input language based on flowchart programs, and the rest of the polyhedral framework will be presented on top of our formalism.

This formalism will characterize polyhedral programs as a subclass of flowchart programs and provide a setting were extensions to the polyhedral model, such as arbitrary loops, which can now be integrated and it removes the constant need to refer to the textual source code for instruction which are at the same level in a loop.

1.4 Conclusion

This chapter exposed the core issues that have to be addressed to exploit the inherent parallelism of programs. The main issue is to figure out dependences between instructions so that they can be swapped and reorganized to provide improved performance by harnessing the characteristics of the underlying hardware.

Computing those dependences is in general not decidable that is why compilers rely mostly on algorithms which gives an approximation of those dependences. However, it should be noted that in a certain case those dependences can be exactly computed and that is where the polyhedral model shines.

In the previous section, we have briefly covered some limitations of the polyhedral model, especially the fact that the input language is hard to describe with mathematical tools which makes abstract reasoning on polyhedral programs hard. Those limitations are not directly about the power of the polyhedral model but only about the fact that this makes it difficult to extend it to a broader class of programs which shares most of the properties of traditional programs.

The second limitation, that we have not yet touched upon, is that this framework only targets programs which deals with arrays.

Summary

+ The polyhedral model is a good fit for regular HPC kernels.
+ An efficient way to auto-optimize polyhedral programs through a sequence of exact algorithms.
− **Weakness 1** Only applicable to well-formed programs (Polyhedral control)
− **Weakness 2** Only applicable to arrays (Direct Access)

Chapter 2 tries to address the first weakness by giving a reformulation of its semantic fundations. Chapter 3, Chapter 4, Chapter 5, Chapter 6, address the second issue.

CHAPTER 2

REVISITING THE POLYHEDRAL MODEL

Research Questions

? How to express the polyhedral model in a more semantic way?

? How to adapt the classical algorithm to general control flow?

The previous chapter concluded by a brief presentation of polyhedral programs and their syntax. In this chapter, we present how those programs can be represented as flowchart programs in order not to rely on syntax anymore. Those flowchart programs serve as the basis for the computation of dependences. All along, we use the product of matrices as our running example (cf. Figure 2.6)

```
for (int i = 0 ; i < N ; ++i)
  for (int j = 0 ; j < N ; ++j)
     M[i][j] = 0;
    for (int k = 0 ; k < N ; ++k)
       M[i][j] += A[i][k]*B[k][j];
```

Figure 2.1: Standard matrix multiplication

2.1 Flowchart Programs and Programs Fragments

This section presents a classical model of programs: flowchart programs which we augment with watched variables. We also present how to modularly compose them.

Flowchart programs with watched variables Let $\mathcal{A}$ be an alphabet, and $\texttt{Var} = \mathcal{A}+$ the set of non-empty words over that alphabet. A flowchart program $\mathcal{F}$ is a tuple ($\mathcal{K}$, $\mathcal{M}$, $\texttt{init}$, $\texttt{end}$, $\mathcal{T}$, $\mathcal{G}$, $\mathcal{S}$, $\texttt{Var}$, $\texttt{addr}$, $\texttt{w}$).

- The set $\mathcal{K}$ of control points : a control point is either a gard or a statement;
- The memory $\mathcal{M} = \langle\{\widehat{m_{i,j}} \mid i \in \mathbb{N}, j \in \mathbb{N}\}\rangle$;
- The initial $\texttt{init} \in \mathcal{K}$ statement;
- The terminal $\texttt{end} \in \mathcal{K}$ statement;
- The set $\mathcal{G}$ of guards;
- The set $\mathcal{S}$ of statements;
- The set $\mathcal{T}$ of transitions;
- The set $\texttt{Var} = \mathcal{A}+$ of variables;
- A function $\texttt{addr} : \texttt{Var} \mapsto \mathbb{N}$ from variable names to address locations.
- A function $\texttt{w}_\texttt{i} : \mathcal{K} \mapsto \mathcal{P}(\texttt{Var})$, a function which tells which control point introduces which watched variables.
- A function $\texttt{w}_\texttt{o} : \mathcal{K} \mapsto \mathcal{P}(\texttt{Var})$, a function which tells which control point removes which watched variables.

Initial and Final control point Those two special control points can both appear only once in a flowchart program and the initial control point (`init`) begins all flowchart programs while the final control point (`end`) ends all flowchart programs.

The memory model The memory is represented by unitary vectors indexed by $\mathbb{N}^2$. For example, $3\widehat{m_{0,0}} + 5\widehat{m_{0,1}}$ means that the cell $(0, 0)$ holds the value 3 and that the cell $(0, 1)$ holds the value 5. The reason why the unitary vectors are indexed by $\mathbb{N}^2$ and not $\mathbb{N}$ is that it makes it easier to handles arrays. Each variable is associated to an identifier $\texttt{i}$ and if this variable is an array and not a scalar then the $\texttt{j}^{\text{th}}$ value is the coefficient before $\widehat{m_{i,j}}$.

Guards The set $\mathcal{G} \subseteq (\mathcal{M} \mapsto \{\texttt{true}, \texttt{false}\})$ of guards is made of functions which take a snapshot of the whole memory and output a boolean value.

Statements The set of $\mathcal{S}$ statements is made of functions s which can only update one cell of the whole memory.

$$s(\{m_{i,j} \mid (i,j) \in \mathbb{N}^2\}) = \sum_{i,j \neq i_0,j_0} m_{i,j}\widehat{m_{i,j}} + f(m_{i_0,j_0})\widehat{m_{i_0,j_0}}$$

where $f \in \mathcal{M} \mapsto \mathbb{N}$ is a function which computes an integer value from a snapshot of the memory and update the cell (i_0, j_0).

Transitions Due to the fact that control points can be either a guard or a statement, a *transition* can be either of the form (k, g, k') or (k, s, k') where (k, k') are, respectively, the source and target control points; g is a guard, and s a statement.

An example such as the matrix product expressed as a flowchart program can be seen in Figure 2.2.

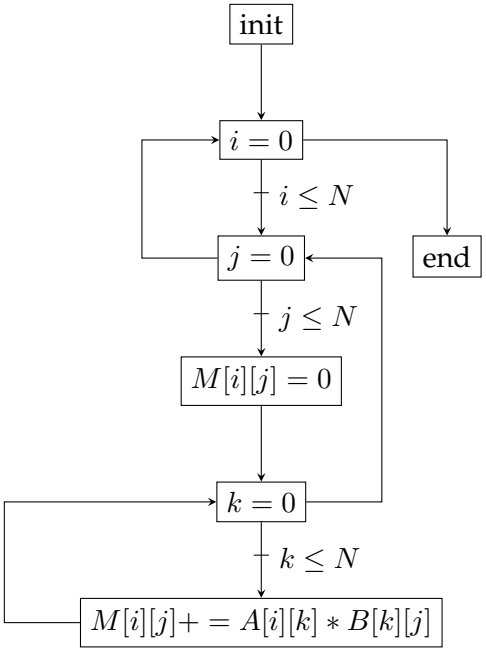


Figure 2.2: Flowchart version of the matrix multiplication (increment statements omitted)

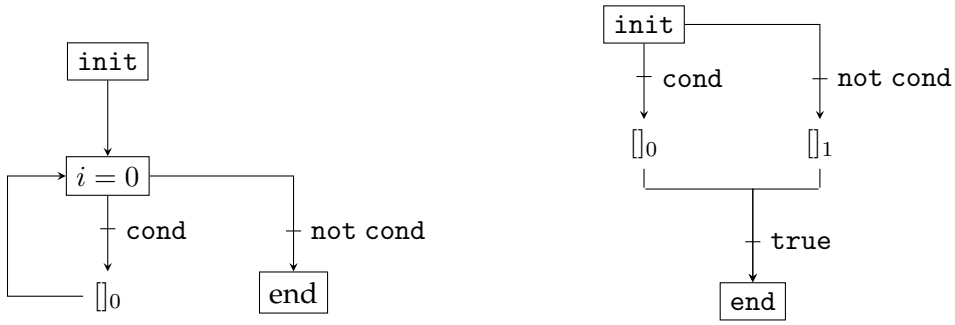
Watched variables Watched variables are special variables which can be introduced and removed by control points and which is active within the control points which introduces it and the one which removes it.

Example 7. In the case of the matrix product Figure 2.2 the statements `i = 0`, `j = 0` and `k = 0` respectively introduce the watched variables `i`, `j`, `k`. Let us call $\mathcal{K}$ the set of control points of the program in Figure 2.2, and let us respectively call k_1, k_2 and k_3 the statements `i = 0`, `j = 0` and `k = 0`, and k_4, k_5, k_6 the respective guards which controls the exit of the first, second and third loops. Then the function $\mathtt{w}_i : \mathcal{K} \mapsto \mathcal{P}(\mathtt{Var})$ and $\mathtt{w}_i : \mathcal{K} \mapsto \mathcal{P}(\mathtt{Var})$ are defined by

$$w_i(k) = \begin{cases} \{\mathtt{i}\} & \text{if } k = k_1 \\ \{\mathtt{j}\} & \text{if } k = k_2 \\ \{\mathtt{k}\} & \text{if } k = k_3 \\ \{\} & \text{otherwise} \end{cases} \quad \text{and} \quad w_o(k) = \begin{cases} \{\mathtt{i}\} & \text{if } k = k_4 \\ \{\mathtt{j}\} & \text{if } k = k_5 \\ \{\mathtt{k}\} & \text{if } k = k_6 \\ \{\} & \text{otherwise} \end{cases}.$$

Flowchart program with holes Flowchart programs with holes are regular flowchart programs where the set of control points $\mathcal{K}$ has been augmented with a set of holes $\{[]_i \mid i \in \mathbb{N}\}$

Example 8. Figure 2.3 presents the fragments for the `while` and `if` control structures, we can see that the while fragment not only depends on its hole but also on the name of the iteration variable and the condition, in the same way, the if fragment depends on the condition.



(a) while fragment (b) if fragment

Figure 2.3: Program Fragments

In this section we have introduced a model which encodes programs into a special kind of graph, which are called flowchart programs. We also introduced how they can be used as bricks to build bigger flowcharts from smaller ones by using flowchart programs with holes. Those flowcharts are further enhanced by watched variables which replace the iteration variables that are traditionally introduced by `for` loops in the polyhedral model. Encoding programs into flowchart allows us to not rely anymore on the source program and give a mathematical foundation which gives a real explicit structure to programs rather than relying on the implicit structure given by the textual representation.

2.2 Polyhedral Programs as Flowchart Programs

As we already saw in the first chapter, the polyhedral model has a long tradition of syntactically-reduced input programs. The language that is used as a support in the seminal papers Feautrier's array dataflow analysis [Fea91] and Barthou's fuzzy array dataflow analysis consists of while loops with an explicit iteration variable (the loop may guarded a predicate in Barthou's work [Bar98]), and conditionals.

This language can be described with the following grammar where `aexpr` are arithmetic expressions (`aexprs` is a list of `aexpr` separated by commas), `bexpr` are boolean expressions, and `id` are identifiers. Variables can appear freely in expressions.

$$
\begin{array}{lcl}
\langle \texttt{do} \rangle & ::= & \text{'do'}\ \langle \texttt{id} \rangle = \langle \texttt{aexpr} \rangle\ \text{'while'}\ (\ \langle \texttt{bexpr} \rangle\ ) : \langle \texttt{stmts} \rangle\ ;\ \langle \texttt{od} \rangle \\
\langle \texttt{if} \rangle & ::= & \text{'if'}\ (\ \langle \texttt{bexpr} \rangle\ ) \text{: 'then'}\ \langle \texttt{stmts} \rangle\ \text{'else'}\ \langle \texttt{stmts} \rangle\ \text{'fi'} \\
\langle \texttt{set} \rangle & ::= & \langle \texttt{id} \rangle [\langle \texttt{aexprs} \rangle] = \langle \texttt{aexpr} \rangle \mid \langle \texttt{id} \rangle = \langle \texttt{aexpr} \rangle \\
\langle \texttt{stmt} \rangle & ::= & \langle \texttt{do} \rangle \mid \langle \texttt{if} \rangle \mid \langle \texttt{set} \rangle \\
\langle \texttt{stmts} \rangle & ::= & \langle \texttt{stmt} \rangle \mid \langle \texttt{stmt} \rangle\ ;\ \langle \texttt{stmt} \rangle \mid \epsilon
\end{array}
$$

Programs described with this grammar can modularly be transformed into flowcharts program, using holes for each compound statement, as described below. The program fragments for the ⟨do⟩ and ⟨if⟩ both introduce watched variables.

Do blocks The statement where ⟨id⟩ get initialized introduces `id` as a watched variables and the guard which gets out the loop removes it.

If blocks In this case, we have to keep apart the `then` and `else` branch, this is done by introducing a watched variable which can take two values: 0 or 1 depending on whether we are in the `then` or `else` branch.

Statement blocks This is a program fragment with an init and final control point with exactly one statement in between.

Once we have those three basic blocks its possible to capture the same programs as the polyhedral model augmented with fuzzy array dataflow analysis [CBF95,BCF97].

Example 9. Matrix multiplication (cf. Figure 2.6) can be easily rewritten with the above grammar:

```
01:  do i = 0 while i <= N:
02:     do j = 0 while j <= N:
03:       C[i,j] = 0;
04:       do k while k <= N:
05:         C[i,j] = C[i,j] + A[i,k]*B[k,j];
06:         k = k + 1;
07:       od;
08:       j = j + 1;
09:     od;
10:     i = i + 1;
11:  od;
```

2.3 Trace Semantics on Flowchart Programs

Now that we have the model of program, we present our notion of traces (more precisely, traces over statements) and a notion of companion sequences (here specialized for polyhedral programs). The role of those companion sequences is to give a number to each statement instances that will appear in a trace, and can be later used to refer to statement instances when computing inter-statement dependences.

Definition 4 (state)**.** A *state* σ is a tuple: (control point, memory snapshot, watched variables). The set of states is $\Sigma = \mathcal{K} \times \mathcal{M} \times \mathcal{W}$, where $\mathcal{W} = \mathfrak{P}(\texttt{Var})$.

Definition 5 (trace)**.** A *trace* Σ is a sequence of pairs of the form (state, statement) $\langle \sigma_0, c_0 \rangle \rightarrow \langle \sigma_1, c_1 \rangle \rightarrow \dots$. An *initial trace* is a trace which begins from the empty state.

Traces of statements Flowchart programs have two kinds of transitions: guards and statements. Hereafter, we are interested in the sequences of statements, that we will call traces over statements. The reason behind the removal of guards in the traces is that only statements modify the environment. Formally this means that we have a relation $\sim_{as}$ such that two adjacent control points only separated by a sequence of guards are equivalent.

Let $\mathcal{P} = (\mathcal{K}, \mathcal{M}, \texttt{init}, \mathcal{T}, \mathcal{G}, \mathcal{S}, \texttt{Var}, \texttt{addr})$ be a flowchart program and $\Sigma/\sim_{as} = (\mathcal{K}/\sim_{as}, \mathcal{M}, \mathcal{W})$ be the set of reduced states.

Deterministic traces A *deterministic trace* $\tau = \{(\sigma_1 = (k_1, \mathcal{M}_1, \mathcal{W}_1) \xrightarrow{s} \sigma_2 = (k_2, \mathcal{M}_2, \mathcal{W}_2)) \mid (k_1, s, k_2) \in \mathcal{T}\}$ is a relation over $(\Sigma/\sim_{as}) \times \mathcal{S} \times (\Sigma/\sim_{as})$ such that $\forall \sigma_1, \sigma_2, s, \sigma_1 \xrightarrow{s} \sigma_2 \in \tau, \sigma_1 \xrightarrow{s} \sigma_2' \in \tau \implies \sigma_2 = \sigma_2'$.

Companion sequences Each trace τ has a companion sequence $\{\rho_{\tau i} \in Vect(\mathcal{W}_i) | i \in \mathbb{N}\}$ which associates to each state τ_i of the trace an expression which value noted $\psi_{\tau i}$ (That is the evalution of the expression $\rho_{\tau i}$ within context τ_i). Those companion sequences satisfy the following property: $\tau_i \rightarrow \tau_j \Rightarrow \psi_{\tau i} < \psi_{\tau j}$. There is a *flow dependence* between two operations $o_1 = (\sigma_1, s)$ and $o_2 = (\sigma_2, s')$ with respect to a trace τ if the statement s' in state σ_2 tries to read a variable written by s in state σ_1 and $\psi_{\tau 1} < \psi_{\tau 2}$.

Traditional polyhedral programs have only one trace due to the fact that there is no conditional control. However fuzzy polyhedral programs might have more than one trace.

Companion sequence for polyhedral programs In this instance ρ is defined as follows ($\rho|_{-i}$ projection on the ith components counting from the last non-zero components). For simplicity, we define it by induction on the grammar:

- `init` : $\rho(\texttt{init}) = (0)$.
- $\rho, \sigma \xrightarrow{s} \rho', \sigma'$
  - $s = \langle \texttt{set} \rangle$, $\rho' = (\rho|_0, \cdots, \rho|_{-2}, \rho|_{-1} + 1)$
  - $s = \texttt{do}\ \langle \texttt{id} \rangle = \langle \texttt{aexpr} \rangle\ [[\cdots]]$, $\rho' = (\rho|_0, \cdots, \rho|_{-1}, \langle \texttt{id} \rangle, 0)$
  - $s = \texttt{od}$, $\rho' = (\rho|_0, \cdots, \rho|_{-2})$
  - $s = \texttt{then}$, $\rho' = \rho' = (\rho|_0, \cdots, \rho_{-1}, 0, 0)$.
  - $s = \texttt{else}$, $\rho' = \rho' = (\rho|_0, \cdots, \rho_{-1}, 1, 0)$.
  - $s = \texttt{endif}$, $\rho' = (\rho|_0, \cdots, \rho|_{-2})$

We can see the introduction of watched variables by ⟨do⟩ and ⟨if⟩ and how they are dropped at the end of the blocks.

2.4 Dependencies for General Programs

Now that we have defined flowchart programs and traces we are now ready to tackle the general definition of inter-instruction dependences. All along this section $\tau = \{\tau_i \mid i \in \mathbb{N}\}$ will be a trace of flowchart program $\mathcal{P}$. In the absence of random of events, the companion sequences is enough to identify each state of a trace, and the companion sequence of τ will be noted $\rho = \{\rho_i \mid i \in \mathbb{N}\}$.

Definition 6 (read and write set)**.** Let c be a control point the program P, then the locations which are written (respectively read) by this control point are noted $\texttt{write}(c@\rho)$ (respectively $\texttt{read}(c@\rho)$). Guards can't write variables, therefore only the read set will be non-empty and the write set is always at most a singleton due to the definition of statements in flowchart programs.

Example 10. Let us consider the statement s defined as $a[i] := a[i-1] + a[i] + 1, t)$, and $\rho' \in \rho$ a fixed instant which corresponds to a τ' such that the statement at τ' is t. Then $\texttt{write}(t@\rho') = \{\ a[i]\ \}$ and $\texttt{read}(t@\rho') = \{\ a[i-1], a[i]\ \}$ where i takes the current value of the watched variable monitoring the variable i.

Definition 7 (Last write)**.** Let ρ' be a fixed instant and ℓ a location, the last write at that location is the fixed instant ρ'' which wrote to that location.

Example 11. In the case of the matrix multiplication, let's consider the statement $\texttt{s}_2$: $\texttt{M[i][j]} += \texttt{A[i][k]} * \texttt{B[k][j]}$ and let i, j, k integers such that $\texttt{s}_2(i, j, k)$ is a valid instance of the statement. For all value of k, the instance write the memory cell $\texttt{M[i][j]}$, the k^{th} instance depends on all the instances $\{s_2(i, j, k') | k' \leq k-1\}$, however it *directly* depends only on $s_2(i, j, k-1)$ which is the last to occur.

Definition 8 (Direct Data Dependencies)**.** An instance of a statement s directly depends on another statement s' if the variable read by s are lastly written by s'. Since a s can read more than one variable, there can be more than one statement s' which is a direct dependence of the statement s. The *most recent dependence* is the instance that is the closer to s of the direct dependences.

Definition 9 (Data Dependencies)**.** An instance of a statement s depends on another statement s' if the variable read by s is written by s' and that s' comes before s.

Example 12 (Dependencies of an operation)**.** Consider the sequence of instances s_0 to s_5 depicted in Figure 2.4. The sequentiality is represented with dashed arrows. The direct dependences between these operations are represented with plain arrows. s_5, directly depends on s_1 and s_3, both being represented with simply dashed circles. The dotted circles denote indirect data dependences of s_5. In Figure 2.5, the most recent direct dependence of s_5 is s_3, noted with double dashed red circle. Even we take the

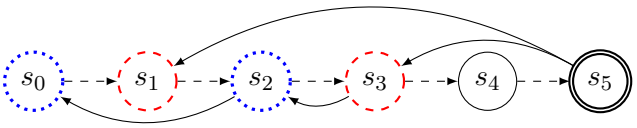


Figure 2.4: Direct (dashed) and indirect (dotted, obtained by transitive closure) data dependences of operation o_5.

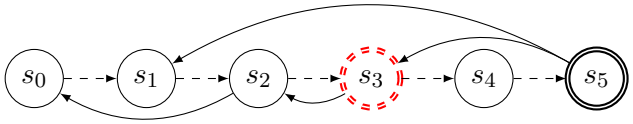


Figure 2.5: Most Recent Direct Data Dependency of s_5.

transitive closure here we can notice that we have definitively lost s_1 and the states which can be reached from s_1.

At that point we have semantically redefined the notion of dependences of the polyhedral model. Now we can build on these definitions and revisit the first key algorithm of the polyhedral model framework, namely, array dataflow analysis.

2.5 Array Data-flow Analysis on Flowchart Programs

The objective of the array dataflow analysis proposed initially in [Fea91] is to compute *all dependences between statements of a given program*. In the polyhedral community's jargon, *dataflow analysis* is thus used as a synonym to *dependence analysis*. In the rest of the thesis, and especially in this chapter, we will use indifferently both terms.

This section rephrases the initial paper [Fea91] in our semantic settings. It is worth to point out that this analysis is exact for loops with static affine control. The "fuzzy" extension [BCF97, Bar98] could also been rephrased in the same way.

When computing dependences, the objective is to compute all the statement-instances which should happen before a specific statement-instance.

Let P be a flowchart program, and s_1 and s_2 be two statements and τ a trace of execution, $Q_{s_1,s_2}(\rho) = \{\rho' < \rho \mid \texttt{write}(s_2@\rho') \subseteq \texttt{read}(s_1@\rho)\}$ be the set of statements on which the statement s_1 at time ρ depends on. Then, the statements on which s_1 depends is $S = \{s_2 \mid Q_{s_1,s_2}(\rho) \neq \emptyset\}$, the statement-instance which (s_1, ρ) depends on is $\cup_{s_2 \in S} \cup_{\rho' \in Q_{s_1,s_2}(\rho)} (s_2, \rho')$, and the most recent dependence is the lexicographic maximum of this last set.

In the case of affine programs, that is, programs where statements are of the form

$$M[f(w)] = M_1[g(w)] \star M_2[h(w)],$$

where M, M_1, and M_2 are matrices, f, g and h are affine functions, w is the vector of watched variables, and $\star$ a binary operator, we can characterize the sets of the above paragraph more precisely.

Let s_1 and s_2 be defined by,

$$\begin{array}{lccc} s_1\text{:} & M[f(w_{s_1})] & = & \ldots \\ s_2\text{:} & \ldots & = & M_1[g(w_{s_2})] \star M_2[h(w_{s_2})] \end{array}$$

Prop 1. *An instance of s_2 depends on an instance of s_1 if and only if the three following conditions are met:*

- $\mathbf{C}_1$ *The instance of s_1 happens before the instance s_2;*
- $\mathbf{C}_2$ *The instance of s_1 wrote a cell that the instance of s_2 reads;*
- $\mathbf{C}_3$ *Both instance of s_1 and s_2 are valid (an instance is valid if there exists a state in which the instance is executed).*

Since, we assume that there is no aliasing, a dependence between s_1 and s_2 can only arise if M_1 or M_2 is the matrix M. If M_1 and M_2 are both M, there is dependence if and only if $g(w_{s_2}) = h(w_{s_2}) = f(w_{s_1})$. If only M_1 is M, there is a dependence if and only if $g(w_{s_2}) = f(w_{s_1})$, and vice-versa if only M_2 is M. Moreover, all those equations are affine.

Example 13 (Computations of dependences for the matrix product, shown in Figure 2.6)**.** This program is made of two statements: $s_1 : M[i,j] = 0$ and $s_2 : M[i,j] += A[i,k] * B[k,j]$, which both write into M. In order to compute the dependences we need to compute Q_{s_1,s_1}, Q_{s_1,s_2} and Q_{s_2,s_2}.

Let us start by computing Q_{s_1,s_1}. We can see that s_1 does not need to read any variable. Hence, Q_{s_1,s_1} is empty. The same is true for Q_{s_1,s_2}.

Now, let us compute Q_{s_2,s_1}. At s_2 both the variables i, j and k are watched, and at s_1 only i and j are watched. Let $\langle i_2, j_2 \rangle$ be the watched vector of statement s_1 and $\langle i_1, j_1, k_1 \rangle$ the watched vector of statement s_2.

$$Q_{s_1,s_2}(i_1, j_1, k_1) = \{(i_2, j_2) \mid i_1 = i_2 \wedge j_1 = j_2\}$$

Lastly, let us compute Q_{s_2,s_2}. Let $\langle i_2, j_2, k_2 \rangle$ and $\langle i'_2, j'_2, k'_2 \rangle$ be the watched vectors of statement s_2 at two distinct instants.

$$Q_{s_2,s_2}(i_1, j_1, k_1) = \{i_1 = i_2 \wedge j_1 = j_2 \wedge k_2 < k_1\}$$

At this point, we have a symbolic graph which captures each (symbolic) operation and the most recent source on which it depends. The important result of the polyhedral model is that for polyhedral programs, this analysis is exact, which means that there is

not under or over approximation of these dependences in the final result of the computation.

Remark. It is important to notice that the three conditions in proposition 1 are *not specific* to the case of arrays, but are sufficient and necessary conditions for (non aliasing) memory cells to be in dependence.

From the result of the dataflow analysis, the polyhedral model propose to compute a schedule, which serves as basis to future code generation. This is the object of the next two sections.

2.6 Scheduling

Scheduling is a core part of the polyhedral model and the methods used to compute schedules evolved a lot [VGGC14] since the first propositions. In its most basic form a schedule can be defined as the following:

Definition 10. A *schedule* $\theta : S \times \mathbb{N}^d \mapsto \mathbb{N}^d$ is a function from the set of statements S times the set of iteration vectors (here, $\mathbb{N}^d$) to a set of logical dates (here, $\mathbb{N}^d$), compatible with dependences: if two operations (s_1, i) and (s_2, j) are in dependence such that (s_2, j) depends on (s_1, i) then $\theta((s_2, j)) < \theta((s_1, i))$ has to be satisfied. This constraint is called *causality* by Feautrier [Fea92a].

An affine schedule is a refinement where the schedule is affine in the second variable. Here we use the definition of affine schedule from the seminal paper from Feautrier.

Definition 11. A schedule is affine [Fea92a] if it is of the form:

$$\theta(s, i) = \tau_s i + \sigma_s n + \alpha_s.$$

where τ_s, σ_s and α_s are rational matrices and n is a vector of constants which appear in the program (such as loop bounds).

A more recent representation of schedules has been proposed in [VGGC14]. This representation called schedule trees not only allows to represent a schedule but also to perform transformations on this schedule by keeping structural information. It also provides a convenient way to express and compose partial schedule (*i.e.* schedules which deal only with a part of the instructions) and to update them.

In more details, schedule trees have the following properties by construction: a compact structural representation, a compatibility with schedule transformations, the expressiveness of partial schedules, the ability to express that some statements should be excuted in a fixed order or in any order, the fact that each operation is given a different execution date and is not executed more than once, and lastly their compatibility to relaxed lexicographic order (*i.e.*, the dimensions of the vectors may not match).

2.6.1 Computation of a schedule

In order to compute an affine schedule, a classical method that we recall here is based on the following result:

Theorem 1. *(Farkas' lemma)*

Let D be a nonempty polyhedron defined by p affine inequalities:

$$D = \{a_k x + b_k \geq 0, 1 \leq x \leq p\}$$

Then an affine form ψ is nonnegative everywhere in D iff it is a positive affine combination:

$$\psi(x) = \lambda_0 + \sum_k \lambda_k (a_k x + b_k).$$

To derive a valid affine schedule, the method consists in fixing a *template* form for the schedule to be found, and compute a system of equations from:

- causality constraints (coming from the dependences analysis): they express that a given statement should be done *strictly after* the statements they depend on.
- positivity constraints: they express that all statements should have positive dates.

The method using Farkas' lemma is informally described below for our running example. We will more formally define its steps when required in this manuscript, in section 6.3.

Example 14. Let us now compute one possible schedule for the product of matrices.

The domains of statements s_1 and s_2 are respectively: $D_1 = \{0 \leq i, j \leq n\}$ and $D_2 = \{0 \leq i, j, k \leq n\}$ and we are searching a schedule which is a non-negative anywhere on those polyhedra, which leads to searching schedules of the following form:

$$\begin{bmatrix} \theta(s_1,(i,j)) \\ \theta(s_2,(i,j,k)) \end{bmatrix} = \begin{bmatrix} \lambda_{s_1,0} & \lambda_{s_1,1} & \lambda_{s_1,2} & 0 & \lambda_{s_1,3} \\ \lambda_{s_2,0} & \lambda_{s_2,1} & \lambda_{s_2,2} & \lambda_{s_2,3} & \lambda_{s_2,4} \end{bmatrix} \begin{bmatrix} 1 \\ i \\ j \\ k \\ n \end{bmatrix}$$

Moreover, we know that the causality relation have to be satisfied:

$$\begin{aligned} d_1 &= \theta(s_2,(i,j,0)) - \theta(s_1,(i,j)) - 1 && \geq 0 \\ d_2 &= \theta(s_2,(i,j,k+1)) - \theta(s_2,(i,j,k)) - 1 && \geq 0 \end{aligned}$$

On which we can apply the Farkas lemma to rewrite them as equalities involving Farkas

multipliers.

$$\begin{bmatrix} d_1 \\ d_2 \end{bmatrix} = \begin{bmatrix} \mu_{s_2s_1,0} & \mu_{s_2s_1,1} & \mu_{s_2s_1,2} & \mu_{s_2s_1,3} & \mu_{s_2s_1,4} \\ \mu_{s_2s_2,0} & \mu_{s_2s_2,1} & \mu_{s_2s_2,2} & \mu_{s_2s_2,3} & \mu_{s_2s_2,4} \end{bmatrix} \begin{bmatrix} 1 \\ i \\ j \\ k \\ n \end{bmatrix}$$

And expanding d_1 and d_2 leads to the following formula:

$$\begin{bmatrix} d_1 \\ d_2 \end{bmatrix} = \begin{bmatrix} (\lambda_{s_2,0} - \lambda_{s_1,0} - 1) & (\lambda_{s_2,1} - \lambda_{s_1,1}) & (\lambda_{s_2,2} - \lambda_{s_1,2}) & 0 & (\lambda_{s_2,4} - \lambda_{s_1,4}) \\ (\lambda_{s_2,3} - 1) & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ i \\ j \\ k \\ n \end{bmatrix}$$

By identification we get the following equalities:

$$\begin{aligned} &\mu_{s_2s_1,0} = \lambda_{s_2,0} - \lambda_{s_1,0} - 1 \geq 0 && \mu_{s_2s_1,1} = \lambda_{s_2,1} - \lambda_{s_1,1} \geq 0 \\ &\mu_{s_2s_1,2} = \lambda_{s_2,2} - \lambda_{s_1,2} \geq 0 && \mu_{s_2s_1,3} = 0 \geq 0 \\ &\mu_{s_2s_1,4} = \lambda_{s_2,4} - \lambda_{s_1,4} \geq 0 && \mu_{s_2s_2,0} = \lambda_{s_2,3} - 1 \geq 0 \\ &\mu_{s_2s_2,1} = 0 \geq 0 && \mu_{s_2s_2,2} = 0 \geq 0 \\ &\mu_{s_2s_2,3} = 0 \geq 0 && \mu_{s_2s_2,4} = 0 \geq 0 \end{aligned}$$

The simplest solution is obtained with $\lambda_{s_2,0} = 1$ and $\lambda_{s_2,3} = 1$ and all other multipliers set to 0, which leads to:

$$\begin{bmatrix} \theta(s_1,(i,j)) \\ \theta(s_2,(i,j,k)) \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ i \\ j \\ k \\ n \end{bmatrix} = \begin{bmatrix} 0 \\ k+1 \end{bmatrix}$$

Which means, that all the initialization can be done in parallel at time 0 and that, at we can execute all instances of the form (i,j,k) for all values all i and j at times $k+1$, as shows Figure 2.6.

2.7 Code emission

State of the Art Once we have a schedule that express the new dates of computation for each statement of the program, we have all the necessary information necessary to generate code respecting this schedule. Code generation is one of the most important

```
// time 0
for (int i = 0 ; i < N ; ++i)
  // the following loop is fully parallel
  for (int j = 0 ; j < N ; ++j)
     M[i][j] = 0;

for (int k = 0 ; k < N ; ++k)
  // time k + 1
  // the following loops are fully parallel
  for (int i = 0 ; i < N ; ++i)
    for (int j = 0 ; j < N ; ++j)
       M[i][j] += A[i][k]*B[k][j];
```

Figure 2.6: Standard matrix with a parallel schedule, after code emission

steps of the polyhedral model. Indeed, the quality of the code produced directly impact the final performance of the programs. The algorithms used for the code generation steps evolved a lot since the inception of the model.

The first algorithm used for code generation was the Boulet-Feautrier's algorithm [BF98]. This algorithm behaves like an automaton. It generates instructions incrementally each time computing the next instructions to be generated. The computation of the next instructions involves solving a parametrized Integer Linear Program.

This algorithm was replaced by Quilleré's algorithm [QRW00] which was later extended by Bastoul et al. [Bas04]. This algorithm which is the one we will use later in Chapter 6, relies on successive projections, and work on multi-dimensional schedules.

This algorithm was further refined into the PLUTO and the PLUTO+ algorithm [BAC16] which are at the core of the PLUTO tool and which are currently used by the isl [Ver10].

In the rest of this section we present the main steps of the Quilleré's algorithm as well as a simple example.

The Quilleré's Algorithm Given a whole global schedule θ for all statements. First, we project θ on each statement, this gives us a family of functions: $\theta_s : \mathbb{N}^d \mapsto \mathbb{N}^d$, one for each statement $s \in S$. Each statement $s \in S$ has a polyhedral iteration domain D_s. The Quilleré algorithm, described next, takes as input the images $\theta_s(D_s)$.

The Quilleré algorithm depicted in Algorithm 1 takes a context (a symbolic polyhedron made of inequalities concerning the symbolic parameters), a list of polyhedra associated with statements (the partial schedules $\theta_s(D_s)$ associated with their respective statement s, from now on we will note $P \rightarrow \{s_1, ..., s_q\}$ a polyhedron P associated with the set of statements $\{s_1, ..., s_q\}$), a dimension d on which to project (which is a constant); and can be subsumed by the following steps:

Algorithm 1 `GenerateCode`$(\mathcal{D}, \rho, d)$)

procedure LOOPGEN$(i, \mathcal{P})$ ▷ dimension i
  if $i = d$ **then**
    return `Statements`$(\mathcal{P})$ ▷ 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

1. Reduce each polyhedron against the context;
2. Project all polyhedron against the dimension d;
3. Combine the polyhedra;
4. Sort the polyhedra according to lexicographic order;
5. Call recursively on the next dimension.

In more details,

1. The first step is here to cut the polyhedra and make them as much as big the context.
2. Next, we project all polyhedra against dimension d. At this point we have the collection $\theta_s(D_s)|_d \rightarrow s$ of bands (polyhedra of dimension 1) associated to their statement but it is possible that there is a pair of statement s_1 and s_2 such that $\theta_{s_1}(D_{s_1})|d \cap \theta_{s_2}(D_{s_2}) \neq \emptyset$.
3. In this step, we combine each overlapping polyhedra. That is, for each statement s_1 and s_2 such that $\theta_{s_1}(D_{s_1})|_d \cap \theta_{s_2}(D_{s_2})(D_{s_2}) \neq \emptyset$ we replace those two bands by: $\theta_{s_1}(D_{s_1})|_d \setminus \theta_{s_2}(D_{s_2})|_d \rightarrow \{s_1\}$, $\theta_{s_1}(D_{s_1})|_d \cap \theta_{s_2}(D_{s_2})|_d \rightarrow \{s_1, s_2\}$, $\theta_{s_2}(D_{s_2})|_d \setminus \theta_{s_1}(D_{s_2})|_d \rightarrow \{s_2\}$.
4. The bands computed in the previous step are sorted with respect to the lexicographic order.
5. For each band associated with a set of statements : $B \rightarrow S$, we recursively apply this algorithm.

After the execution of this algorithm, we can generate code from the band tree.

Example 15. On the running example, we denote by (i, j, k) the iteration dimensions,

and take the identity schedule,

$$\theta_{s_1}(i,j) = \theta(s_1,(i,j)) = (i,j)$$
$$\theta_{s_2}(i,j,k) = \theta(s_2,(i,j,k)) = (i,j,k)$$

where s_1: `M[i][j] = 0` and s_2: `M[i][j] += A[i][k]*B[k][j]`, the image of those two functions are the following polyhedra.

$$Im(\theta_{s_1}) = \{0 \leq i,j \leq N-1\}$$
$$Im(\theta s_2) = \{1 \leq i,j,k \leq N-1\}$$

The first projection gives $[P_1, P_2]$ where $P_1 = P_2 = \{0 \leq i \leq N-1\}$, which is then repartitioned into $[(P_1, \{s_1, s_2\})]$ (we track the associated statements). Now we generate the loop inside P_1, that is we project onto the second direction, and we get $[P_{11}, P_{12}]$ where $P_{11} = P_{12} = \{0 \leq j \leq N-1\}$, and again the repartition is $[(P_{11}, \{s_1, s_2\})]$. In the end, we project on the remaining dimension $[P_{112} = \{0 \leq k \leq N-1\}]$ which only covers the statement s_2.

```
for (i = 0 ; i <= 0 ; i += 1) // P1
  for (j = 0 ; j <= 0 ; j += 1)  // P12
    M[i][j] = 0;
    for (j = 0 ; j <= N - i - 2 ; j += 1) // P22
      M[i][j] += A[i][k]*B[k][j];
```

2.8 Conclusion

In this chapter, we presented an alternative way to represent programs which do not rely on syntax anymore. This alternative encodes programs as graphs and provides proper structure to the objects (instances of statements within loops) which are manipulated. This presentation makes it easier to relax hypothesis on programs since it is only a matter of relaxing hypothesis on the form of a graph. The other steps of the polyhedral model are, however, left untouched.

CHAPTER 3

COMPLEX DATA STRUCTURES

Research Questions

? How can complex data structures be integrated within a polyhedral model like framework?

? What are opportunities to optimize tree-shaped data structures?

The previous chapter presented the main steps behind the polyhedral framework and laid out semantic foundations to reason theoretically on polyhedral programs and "nearly polyhedral". In this chapter, we come back to the second research question: How are other data structures than arrays handled by the polyhedral model and its extensions; and to what extent can we handle algebraic data types within the polyhedral model or by reusing ideas from this model ?

In this chapter, we will firstly review prior work on array-like data types, and then argument in favor of a deeper study of non sharing algebraic data types, namely, trees.

3.1 Array-like Data Types

3.1.1 Existing works in the polyhedral model community

Most recent extensions of the polyhedral model are not fully static, but they strive to keep the dynamic analyses to a bare minimum.

Sparse Matrices Sparse matrices are matrices with so many zeroes that storing them individually is a waste of space. Such matrices make up a large part of scientific com-

putations such as partial differential equations solvers, graph analysis (especially those with few pairwise connections such as network graphs) or machine learning (especially natural language processing). Many dense representations of sparse matrices exists such as compressed sparse row (CSR) and compressed sparse columns (CSC) among others [Pis84], but their contrived access patterns which require indirect access makes extracting parallelism much more complex [MYC+19].

Due to those complex access patterns, the equations which describes dependences between cells cannot be fully resolved statically. The dependence relations which cannot be resolved at compile time are conserved and checked at runtime by an inspector. The role of the inspector is to verify whether the dependence relation is real before using parallel code. Since those checks happen at runtime, the more check there are, the longer it takes. The main line of research in this field is finding methods to offload as much as possible of the inspector work to the compiler [MYC+19][1]. In practice, it is frequent that the inspector code in iterative solvers cost more than the computation itself but since the inspector code is run once whereas the computation is run many times, the cost of the inspector is amortized.

Array compacting Not all programs which work on arrays always need to keep the whole array in memory. The classical example of this phenomenon is the computation of the Fibonacci sequence:

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

Wen computing, the Fibonacci numbers we could keep all previous Fibonacci numbers, but we only need to keep the last two to compute the next. Obviously, this is a very simple example but the key idea is here: reduce the memory usage by reusing cells as much as possible.

Finding which cells can be reused is very dependent on how the operations in the program are scheduled. Polyhedral programs are a class of program where the schedule can be statically computed. Work on array compression [BBC16c, QR00, WR96, BBC16a] have been done into two directions: intra-array compression [ABD07, BBC16b] (which try to save space only within one array) and inter-array compression (which try to save space in distinct but interdependent arrays). In practice, the array compression works well when the lifetimes of the memory cells can be explicitly computed, for example when the array access patterns are Presburger formulae.

3.1.2 Lists and Dynamic Arrays

The objective of this section is to handle the case of simple dynamic data structures, which are not explicitly in the scope of the polyhedral framework but which can be handled seamlessly.

[1] A review of the literature of this field of research can be found in this paper as well.

Operation	Dynamic Array	List
Insertion (Head)	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Insertion (Tail)	$\mathcal{O}(1)^*$	$\mathcal{O}(n)$
Insertion (Random)	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Deletion (Head)	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Deletion (Tail)	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Deletion (Random)	$\mathcal{O}(n)$	$\mathcal{O}(n)$

Table 3.1: Complexity of operations on lists and dynamic arrays (Complexities with an * are amortized.)

Even though lists are overrepresented in the litterature due to easy to describe algorithms, they do not offer any true benefit over dynamic arrays. One of the few advantages of lists is that they never use more memory than what is strictly necessary. Indeed, list cells are allocated on demand whereas dynamic arrays use preallocation of cells to avoid useless copies.

Definition 12 (List)**.** A list is either the empty list or a value followed by a list.

Definition 13 (Dynamic Array)**.** A dynamic array is an array which can be resized dynamically.

Both present similar theoretical performance (see Table 3.1), list are better at inserting at the head, dynamic array are better at inserting at the tail. Lists have a theoretical advantage when it comes to insert at a given position, however this is only really an advantage if we have a pointer to the position, otherwise we need to traverse the list.

We notice that except for the case where the position is already known both dynamic array and lists share the same complexities. Lists, are therefore better when it comes to insert a list within a list, however, a good implementation of dynamic array should provide such a function as well with a similar complexity which means that from a theoretical standpoint both data structures behave the same.

Since both data structures support the same set of operations with similar complexities, it is enough to support only one of them, preferably the one with the best performance in practice.

Lists are usually stored non contiguously in memory which makes theirs traversals very inefficient due to the cache policy even when taking into account that allocation functions are designed to return contiguous blocks across successive calls. Since, we want to optimize speed, it seems better to only support dynamic arrays and convert all instances of lists into dynamic arrays.

Another advantage of dynamic arrays is that they are supported by the polyhedral model (as long as the loop we are trying to optimize do not modify the length of the

array). Indeed, in the polyhedral model, the array length is a structure parameter and as long as it does not vary during the loop execution no problem occurs.

3.2 Algebraic Data Types

So far, we have seen data-structure which are "linear" in nature, i.e., without branching. Algebraic Data Types (ADTs for short) are used for a wide variety of purposes such as representing context (render trees, ray tracing, abstract syntax trees); tree-based datastructures (AVL [AVL62], red-black or B-Trees [GBY91]); or model domain-specific data. These structures are widely used in immutable functional languages for everyday programming, but also in more performance-sensitive contexts to build data-structure. They can then rely on in-place mutations to modify the corresponding term. Sadly, their use in the High-performance community is so far limited. Our opinion is that one main reason is the lack of highly optimizing compilation techniques dedicated to operations manipulating ADTs, which is precisely what we hope to start addressing in this thesis through combination with the polyhedral model. In this section we define ADTs and present some examples.

3.2.1 Definitions

Definition 14 (Sum Types)**.** A *sum type* is an umbrella for multiple types. The type $\mathtt{T}$ umbrella for the types $\mathtt{T}_1$, $\mathtt{T}_2$ is denoted by $\mathtt{T} = \mathtt{T}_1 \mid \mathtt{T}_2$. Roughly speaking, a sum type can be seen as a disjoint union.

Definition 15 (Product Types)**.** A *product type* $\mathtt{T}$ over $\mathtt{T}_1$ and $\mathtt{T}_2$ is the type made of tuples of elements of $\mathtt{T}_1$ and $\mathtt{T}_2$ denoted by $\mathtt{T} = \mathtt{T}_1 \times \mathtt{T}_2$. From a set theoretic point of view, where types are represented as sets, a product type is a product set.

Definition 16 (Algebraic Data Type)**.** An *algebraic data type* (ADT) $\mathtt{T}$ is a sum of product types where each alternative is labeled, and each label can be also considered as a function which yield an instance of $\mathtt{T}$ which can be defined with the following grammar (where the overline over the Constr part means list of constructors):

$$
\begin{aligned}
\tau \in \mathtt{Types} ::= {} & \mathtt{Integer} \mid \mathtt{Float} \mid \mathtt{String} \mid \ldots && \text{(Base types)} \\
\mid {} & (\tau_0, \ldots, \tau_{n-1}) && \text{(Product type)} \\
\mid {} & \mathrm{Constr}_0(\tau_0)\ \text{`|'} \cdots \text{`|'}\ \mathrm{Constr}_{n-1}(\tau_{n-1}) && \text{(Sum Types)}
\end{aligned}
$$

Example 16. The type $\mathtt{T} = A(\mathtt{T}_1) \mid B(\mathtt{T}_2, \mathtt{T}_3)$ is an algebraic type with two constructors named A and B which can be viewed as functions of type: $A : \mathtt{T}_1 \to \mathtt{T}$ and $B : \mathtt{T}_2 \to \mathtt{T}_3 \to \mathtt{T}$.

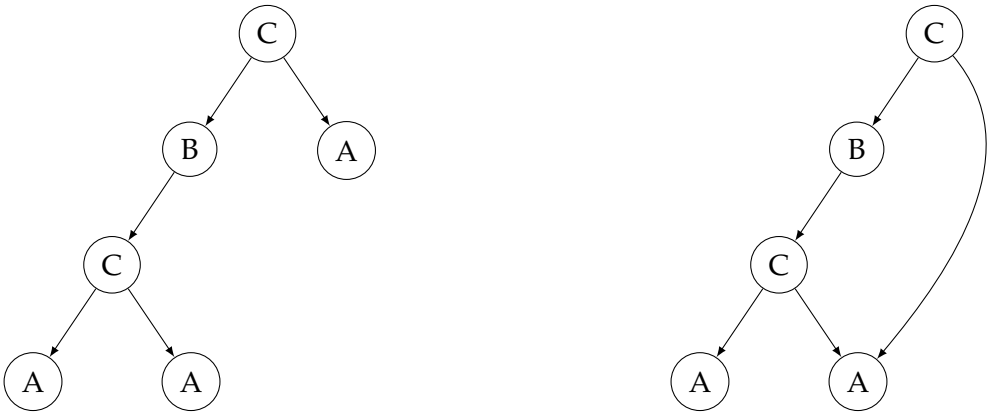
Example 17. Arbitrary List types can be expressed by ADTs, for example, the type $t = \mathtt{End}(\mathtt{String}) \mid \mathtt{B}(\mathtt{Int}, t) \mid \mathtt{C}(\mathtt{Float}, t)$, is a list type which can hold both integers and float values and which last values is always a string.

Similarly, we can use algebraic data types to define trees, and graphs.

3.2.2 Trees: terms without sharing

In the rest of the manuscript, we will mainly focus on terms without sharing. The fact that there is no sharing means that a node has at most one parent (in-degree of at most 1). A type with such a constraint is a tree. (See Example 18).

Example 18. Let $\mathtt{t} = \mathtt{A}(\alpha) \mid \mathtt{B}(\beta, t) \mid \mathtt{C}(\delta, t, t)$ the definition of an Algebraic Data Type. Here is an example of a value of type $\mathtt{t}$: $\mathtt{C}(\delta_1, \mathtt{B}(\beta, \mathtt{C}(\delta_2, \mathtt{A}(\alpha_1), \mathtt{A}(\alpha_2))), \mathtt{A}(\alpha_2))$. This value can be represented by two ways in memory, as seen in fig. 3.1: either represent each node separately, or try to share similar nodes, for instance the one containing α_2. We see that depending on whether we allow sharing, there is at least one node with an in-degree bigger than 1.



(a) Without sharing (b) With sharing

Figure 3.1: Two values of the type of example Example 18

Applications using trees can be roughly divided into two categories. For the first one, trees are used to store values that will be later queried, and in the second trees to represent points in a spatial area.

Trees as Storage This category can be further divided into two main sub-categories: binary trees and non-binary trees. Those trees are always designed to behave well and not degenerate into lists, this is mostly achieved by equipping those trees by a balancing mechanism. The most common tree families of balanced trees are AVL [AVL62] and red-black trees [GBY91] which are both binary trees; and B-trees [GBY91] and their variants for the case of non binary trees. They are at the root of efficient data-structures which need to be frequently queried such as sets, maps or dictionaries. Hence, they are often found in traditional databases. The growing need of analyzing large amount of data gathered from the Internet and stored in gigantic databases requires harnessing the computing power of high performance parallel machines at their fullest. Improving

the processing of trees is part of this endeavor and, a first step in this direction has been made by Blelloch et al. [BFS16], [SB19, SFB18] who investigated the benefits of bulk operations to increase parallelism.

Spatial Data Partitioning with Trees The other category, which is used to represent spatial properties can similarly be subdivided into two categories. First, trees can be used to divide a spatial region: for example, bk-trees are element in a metric space and make it simple to find elements which are at a fixed distance; kd-trees, quad-trees, or oct-trees which are used to store points in space. Repeated traversals of such trees, and the application of polyhedral model techniques to handle them have been studied [JK11, JK12, GJK13, HLSK17, SK19, KRV+21] and yield promising results when it comes to parallelize a bulk of traversals and improve data-locality of those traversals. Secondly, render trees are trees used to store a scene. Scenes can range from 3D scenes to rendering the tree representing a web page in your browser. Such rendering processes features a great number of passes that needs to be applied to each node. In order to limit the number of traversals, they are merged to some extent [SSNK19].

3.3 Conclusion

In this chapter we have presented how the polyhedral model has been or could be adapted to work on relaxed arrays (sparse arrays, small arrays whose cells can be reused). When dealing with other datastructures, the next step is to be able to express algebraic data types without sharing, namely, trees.

Work on integrating trees in the polyhedral model was first considered by Feautrier and Cohen [Fea98] and are the subject of Cohen's PhD thesis [Coh99]. While this thesis lay a solid basis to work on tree, the proposed algorithms rely heavily on transducers which suffers from expressiveness problems. Transducers also suffer from the fact that, like automata, transforming them into deterministic transducers takes exponential time. This leads us to explore other directions.

In the rest of this manuscript, we propose a slightly different approach which aims to be more general and apply to arbitrary tree shapes data types. We will propose algorithms and tools for efficient compilation of such data structures. The case of recursive terms and sharing (DAGs and general graphs) are left out of scope of the current manuscript.

CHAPTER 4

TREES AND THEIR LAYOUTS

Research Questions

? What are the operations to optimize on trees?

? What are alternative layouts for trees, and their strength and weaknesses?

In the preceeding chapter we have focused our interest on trees (as terms without sharing). The objective of this chapter is twofold: first, present the frame of our study of trees: which trees, which operations on trees? And then, study their representation in memory as *linear* layouts.

4.1 A Brief Overview on Trees

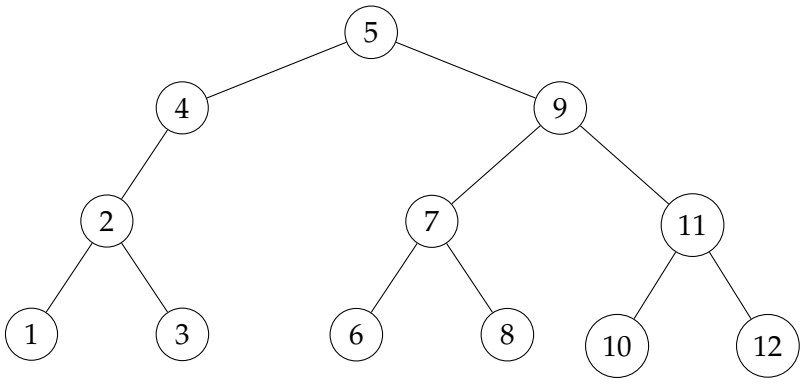


Figure 4.1: A tree

A tree is an inductive data structure which can be described recursively this way: a tree is either the empty tree or the tuple made of a list of trees and a value. This can be expressed with the following definition:

$$\text{Tree} := \text{Empty} \mid \text{Node}(\text{List}(\text{Tree}), \text{Value}).$$

This structure is used to store values with additional spatial properties. The most general definition, presented above, tells us that a value or point can have an arbitrary number of neighboring nodes. The nature of the spatial property is strongly linked with the purpose of the tree.

4.1.1 Context and Inspiration

Modern literature on data-structures algorithms classically distinguish *cache-oblivious* algorithms from *cache-aware* algorithms. In *cache-aware* algorithms, the complexity is optimized regarding a given size of cache lines whereas *cache-oblivious* [FLPR99] ones are optimized up to an unknown block size.

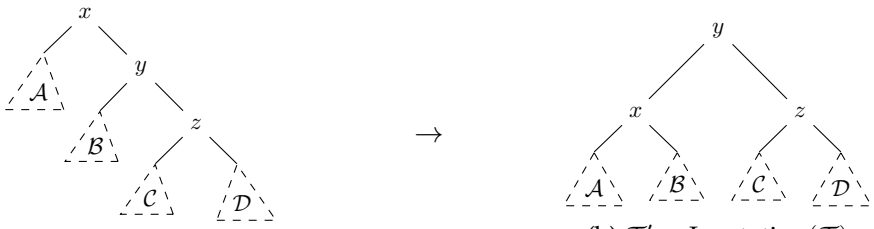
Since Frigo et al.'s seminal article [FLPR99], a huge number of algorithms have been studied, from matrix transpositions to FFT or sorting, the most recent and impressive descendant of these works being the *Piecewise Geometric Model index* data-structure [FV20], which relies on clever dynamic adaptation of the data-structure according to the history of queries.

Many of these work rely on variants of *search trees*. A fascinating line of research propose to encode variants of B-trees into static arrays [BDF00, BFJ02] and shows promising experimental evidence for practical efficiency. These algorithms are often quite sophisticated but still rely on in-place structural transformations on the structure of the search tree. One key ingredient in this line of work is the *layout* of the tree in memory. In this chapter, we present several layouts and demonstrate them on concrete algorithms.

4.1.2 Tree Operations

We now focus on binary search trees (in which values of the left subtree are all less than to the parent's and that values of the right subtree are all superior), and more precisely, AVL trees. AVL trees [AVL62, GBY91] are a classical tree representation combining the advantages of binary search trees with a self-balancing behavior which ensures performances do not degrade when data is inserted or deleted by maintaining the depth of the tree at $\lg n$ (where n is the number of nodes). We now recall the key definitions and algorithms.

Definition 17 (AVL Tree)**.** An AVL tree is a binary search tree such that both of its children are AVL trees and that the absolute difference of theirs heights (we also call it depth) is strictly less than one.



(a) A binary tree $\mathcal{T}$ (b) $\mathcal{T}' = \text{L-rotation}(\mathcal{T})$

Figure 4.2: A left rotation

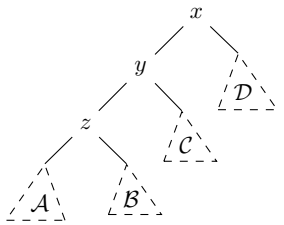
AVL trees support the same operations as standard binary search trees (*insert*, *delete* and *find*). However, in order to keep them balanced when inserting or removing an element, they also provide a mechanism called *rotation* (Figure 4.2)

Insertion When inserting an element in an AVL tree, the process is largely the same as if inserting an element in a binary search tree: we walk the tree until we find the right place for the new element. Figure 4.3 contains the algorithm performing the insertion in an AVL tree. After an insertion, the tree might be left in an unbalanced state (this is determined thanks to the depth field of the node and the depth field of its children) and may need a rotation (Figure 4.4).

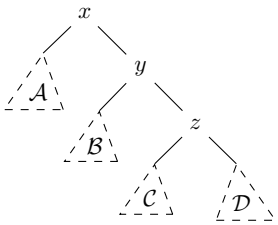
```
def avl_insert(node, val):
  if tree is Empty:
    return(newNode(val));
  if (val < node.val)
    node.left = avl_insert(node.left, val);
  else if (val > node.val)
    node.right = avl_insert(node.right, val);
  else
    return node;

  node.depth = # Update depth
    1 + max(depth(node.left),depth(node.right))
  balance(node) # Rebalance the tree
  return node
```

Figure 4.3: Insertion in an AVL

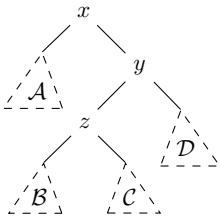
Deletion Again, the deletion process of AVL trees is very similar to the deletion process of standard binary search trees. First, we need to find the node that we want to remove, if this node is a leaf it can be removed directly, however, it this node has at least one child, we need to swap it with either the biggest leaf of the right child or the smallest leaf of the right child. Once the node and the leaf have been swapped, we



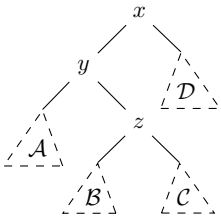
(a) Needs an R-rotation



(b) Needs an L-rotation



(c) Needs an RL-rotation



(d) Needs an LR-rotation

Figure 4.4: Unbalanced trees

can remove the leaf. Additionally, since we want to preserve the balanced property we have to update the depth field of each node and balance accordingly.

An insertion needs at most one rotation to preserve the balance while a deletion may require as much as $\mathcal{O}(depth)$ rotations.

Search The search algorithm, depicted in Figure 4.5 in an AVL tree is exactly the same as the search in a binary search tree. Indeed, AVL are binary search trees, however, the search complexity is guaranteed to be at most $\mathcal{O}(\ln n)$, where n is the number of nodes of the tree.

```
1 def avl_search(node, val):
2   if tree is Empty:
3     return Empty;
4   if (val < node.val)
5     node.left = avl_search(node.left, val);
6   else if (val > node.val)
7     node.right = avl_search(node.right, val);
8   else
9     return node;
```

Figure 4.5: Search in an AVL

4.1.3 Classical representation of trees and their drawbacks

Trees, and as such AVL trees which make the core of our examples, are usually implemented a *pointer-based* representation which can be seen in Figure 4.6: each node

contains the data, pointers to its children and its depth.

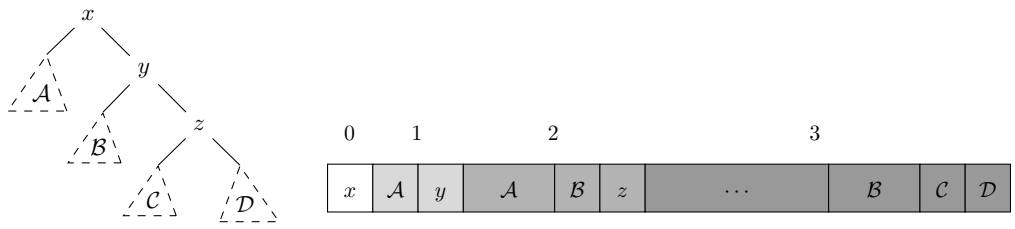
This representation is the most common due to being very convenient for the main operations on trees: insertion, deletion and search, tree rotations are only a matter of swapping pointers. However, despite the fact that consecutive memory allocations are optimized such that they will allocate consecutive strands of memory, the memory locality of those trees is not optimal, since there is no guarantee on the proximity in-memory of neighboring nodes. This means that each traversal incurs hard to avoid cache-misses. Moreover, the unpredictable nature of the address of the nodes makes it impossible to optimize the traversal as a loop.

```
1 struct tree {
2   int data, depth;
3   struct tree *left, *right;
4 };
```

Figure 4.6: Pointer-based tree in C

4.2 Layouts

In order to perform more locality-aware optimisations, one possibility is to avoid representing trees with pointers. The objective of this section is thus to describe how to store trees inside a linear *contiguous* amount of memory, as Figure 4.7 shows a first example and Figure **??** shows the three layouts that we will study in details in this chapter. Such layouts are said to be "implicit" in the literature. We introduce the notion of *layout function* for describing different layouts and give examples of such functions for the case of n-ary trees and also specialize for our binary search trees. We also show how these storages can be compressed (while keeping some of their properties) in some cases.



(a) A binary tree $\mathcal{T}$ (b) Depth First Layout

Figure 4.7: A binary tree and one of its possible linear layouts

Layout functions In order to single out each node of a tree, we use an encoding based on the path from the root to the (sub)tree which contains the value as root, which is

defined as the set of positions in a tree.

$$\mathrm{Pos}(t) = \bigcup_{0<i<n_t} \{i.j \mid j \in \mathrm{Pos}(c_i)\}$$

where n_t is the number of children of t and c_i is the i-th child of t, and $|$ denotes concatenation. The only position for an empty tree is ε.

Definition 18 (Layout function)**.** A layout function is a function f which associates to each position an index *i.e.*, an address.

Let us now discuss what would be a good layout function:

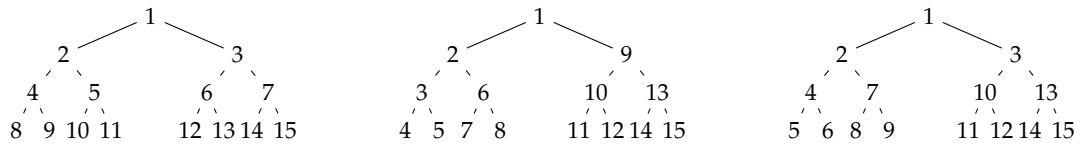
- With a layout function, we have a mapping from positions inside the tree to an index in the array. If this function is bijective, and if we are able to easily compute its inverse, then the adaptation of an algorithm on trees to its version on linearized trees is trivial.
- Similarly, as functions on trees usually use parent/child relations to walk on trees, a good layout function would be one for which these relations are easy to compute.
- Classically, the choice of a datastructure should be done according to the most frequent operations; and chosing a good layout is the same. In particular, some layouts will fit best if the tree is not modified too often.

Pointer-based layout The traditional layout based on a collection of pointers stores additional information to compute the address of each value, that each tree does not only embed a value but also the list of the addresses of its neighbors. Let t be a tree and $p \in \mathrm{Pos}(t)$ a position in t, f is defined as

$$f_t(p) = \begin{cases} f_{c_i}(j) & \text{if } p = i.j, i \in \mathbb{N}, j \in \mathrm{Pos}(c_i) \\ v & \text{if } p = \varepsilon \end{cases}$$

v is the final address. In the general case, the computation of f_t depends on the computation of c_j which in turn depends on the form of c_j. If we want to write a function which can store all the values of a given tree into a contiguous array we need a function which gives us the space needed by all those c_j. In the general case, since the number of children is not bounded such a function cannot be computed statically. Therefore, in the following we will focus on n-ary trees, that will be sufficient for our needs.

From traversals to layouts (n-ary trees) If we fix a tree, we notice that tree traversals induce a layout. Indeed, a traversal goes through all the nodes of the tree in a fixed order which is enough to define a function which map each position of the tree to the position it appears in the traversal. The layout function defined from a traversal on a fixed tree are overly dependent on the shape of the tree, and they are difficult to capture with a closed formula. For this reason, we can think of adding empty nodes which will make it easier to express with a simple formula the layouts induced by traversals.



(a) Breadth First (b) Depth First (c) van Emde Boas

Figure 4.8: Three different memory layouts

Hole-free storage A hole-free storage stores the value collected during a traversal without ever leaving a cell empty for future use. For example, the tree from Figure 4.1 is stored as $[5, 4, 9, 2, 7, 11, 1, 3, 6, 8, 10, 12]$ according to breadth first traversal, or as $[5, 4, 2, 1, 3, 9, 7, 6, 8, 11, 10, 12]$ according to preorder depth first traversal. Despite the compactness of the representation, the structure of the tree is lost and without additional information it is not possible to reconstruct the structure of the tree, and therefore it is not possible to compute the accessors function (parent, children) directly on those representations. It would be possible to reconstruct the tree structure by either knowing that the tree is a binary search tree or by storing additional information about nodes in tables. In any case, this is an expensive process.

Holed Storage A representation with holes trades off space for regularity. For example, the tree of Figure 4.1 is stored as $[5, 4, 9, 2, [], 7, 11, 1, 3, [], [], 6, 8, 10, 12]$ according to breadth first traversal, and $[5, 4, 2, 1, 3, [], [], [], 9, 7, 6, 8, 11, 10, 12]$ according to preorder depth first traversal. Despite the space loss, those storages can be compacted and reduce holes. And, what is more, the regularity allows us to compute the accessors functions.

Compression Holed storages waste space for regularity and, allow writing insertion and deletion functions easily. However, when the tree is modified the waste space tends to grow. Henceforth, when there are too many holes we compress the tree by creating a tree with the same content but with fewer holes. The compression process reorder the nodes so that the tree is as balanced as possible. Formally, a compression algorithm constructs a map (and more precisely a permutation) over the elements of the tree so that the resulting tree needs fewer holes.

Notations In the following, we will use the following notation to refer to trees, expressed with different layouts. Let $\mathcal{T}$ be a tree, $[\dots]_{pr}$ describes the values in the order they are found in a prefix depth first traversal, $[\dots]_{in}$ in an infix depth first traversal, $[\dots]_{po}$ in a postfix depth first traversal, $[\dots]_b$ in a breadth-first traversal and $[\dots]_v$ in a van Em Boas traversal.

In the following, we will only use holed-storage.

4.2.1 Depth-First Representation

This representation is well-suited for binary search trees since the default search algorithm is depth first search. There are multiple variants of the depth first representation, prefix, infix and postfix.

Prefix and Postfix Representation Here we only present the prefix representation. The access functions are pretty expensive to compute: the index of the right children have to be stored, and they are dependent on the depth of the node.

$$\begin{aligned}
\texttt{parent}(i) &= \max\{j < i | h(j) = h(i) - 1\} \\
&= \begin{cases} i - 1 & \text{if } h(i-1) = h(i) - 1 \\ i - 2 * 2^{h(i)-1} & \text{if } h(i - 2 * 2^{h(i)} - 1) = h(i) - 1 \end{cases} \\
\texttt{left_child}(i) &= i + 1 \\
\texttt{right_child}(i) &= i + 2^{h(i)-1} + i
\end{aligned}$$

The subtree function is easily computed as well and a nice property is that subtrees are also contiguous subarrays. The subtree at index i is $[\texttt{left\_child}(i), \texttt{right\_child}(i) - \texttt{nb\_nodes\_below}(i)]$.

Infix Representation The infix representation is interesting because the access functions are easier to compute and do not require separate look up tables to preserve the tree structure. Indeed, since we are considering a holed-representation, the array which store the values is of size $2^h - 1$ with h the height of the tree. The root is center value (at position $2^(h-1)-1$ (counting from 0)), its left child is the center value of the left part (at position $2^(h-2)-1$) and its right child is the center value of the right part (at position $2^(h-1)+2^(h-2)-1$). This representation is of special interest when it comes to binary trees since the infix order is a sorted sequence.

Example 19. In infix representation, the tree of Figure 4.1 becomes $[1, 2, 3, 4, [], [], [], 5, 6, 7, 8, 9, 10, 11, 12]_{in}$, its root is at position 7 ($= 2^{(4-1)} - 1$), its left child is at position 3 ($= 2^{(4-2)} - 1$) and its right child is at position 11 ($= 2^{(4-1)} + 2^{(4-2)} - 1 = 8 + 4 - 1$).

Insertion & Deletion Both operations closely follow the standard algorithm for binary search trees.

Search The search for a value always start from the beginning of the array, if the value is less than the one under the cursor we can continue with the next cell (the left child) unless the next cell is actually bigger which means that the value we are searching for is not in the tree. Otherwise, when the value is superior to the one under the cursor we have to look up the right child in the look up table which store the position of all right children.

Compression In infix representation, the compression algorithm removes all the holes and fill the array with zeroes until the next power of two minus one.

Example 20. In infix representation, the tree of Figure 4.1 becomes $[1, 2, 3, 4, [], [], [], 5, 6, 7, 8, 9, 10, 11, 12]_{in}$, this tree cannot be compressed more, so the compression algorithm only moves the holes to the end. Henceforth, after compression it becomes $[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, [], [], []]_{in}$.

In other representations, prefix and postfix, the compression is less straightforward. In the following I describe the compression scheme for the prefix representation (postfix is similar). The algorithm is performed in two phases:

- First we traverse the whole array to retrieve all the elements in sorted order;
- The second phase uses a property of AVL search trees. When values are inserted in ascending or descending order, the final shape of the tree is known, it is a perfect tree. Therefore, we can simply reinsert all the values such that it forms a perfect tree. For that purpose we complete the list obtained in the first part with hole values until we have $2^n - 1$ values for an arbitrary n. The middle value will go into the first cell of our array, at this point the values are split in two the values smaller than the middle and the values bigger than the middle. The middle value in the bigger part as well as the starting index and ending index of the bigger are enqueued, and we repeat the initial process with the smaller values, taking the middle one, splitting in two, enqueue the value of the middle of the bigger set of values as well as its range. When we don't have smaller values we can start to pop values from the queue and repeat the process, each time we pop a value from the stack we also have to update the right child lookup table for the prefix representation, if at some point we happen to run into a hole we just ignore it and use as middle the middle on its left.

Example 21. In the infix depth first layout the tree Figure 4.1 is stored as $[5, 4, 2, 1, 3, [], [], [], 9, 7, 6, 8, 11, 10, 12]_{in}$. First we sort the values and fill with holes so that the number of elements is a power of two minus one: $1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, [], [], []$. Once this is done, we can perform the second phase which will compute the compressed tree.

Step 1 In the first step, we start with an empty array and a stack containing the array with the sorted values.

$[]_{pr}$

Stack
[1,2,3,4,5,6,7,8,9,10,11,12,[],[],[]]

Step 2 We take the values at the middle, and split the stack in two arrays.

$[8]_{pr}$

Stack
[1,2,3,4,5,6,7]
[9,10,11,12,[],[],[]]

Step 3 We take the middle value of the array on the top of the stack, and we split it in two.

$[8, 4]_{pr}$

Stack
[1,2,3]
[5,6,7]
[9,10,11,12,[],[],[]]

Step 4

$[8, 4, 2]_{pr}$

Stack
[1]
[3]
[5,6,7]
[9,10,11,12,[],[],[]]

(Steps 5 to 8 omitted)

Step 9 At this point we have done the first half.

$[8, 4, 2, 1, 3, 6, 5, 7]_{pr}$

Stack
[9,10,11,12,[],[],[]]

(Steps 10 to 14 omitted)

Step 15 In the end, the stack is empty, and we have the compressed representation.

$[8, 4, 2, 1, 3, 6, 5, 7, 12, 10, 9, 11]_{pr}$

Stack

4.2.2 Breadth-First Representation

This representation has the advantages that the access functions are very easy to compute and lead to a very regular memory model where insertion, deletion, (and even rotations when used to store rotation-based self-balanced trees) are easy to express. Its main problem is that it can lead to very sparse array if the tree gets unbalanced.

- `parent`(i) = $i/2$ with $i \neq 0$.
- `left_child`(i) = $2i$.
- `right_child`(i) = $2i + 1$.

Again, since holed-storage based layout can only handle tree where the number of children per node can be bounded beforehand, we run into the same limitations as before.

Insertion & Deletion Both operations follow closely the standard algorithms used for binary trees.

Search The search is performed the same way as with standard binary search but with the $2i$, $2i+1$ scheme.

Compression The compression algorithm uses the fact that when values are inserted in order in an AVL tree, the resulting tree is a prefect tree. Without loss of generality, we will assume that the values to be inserted are in increasing order. This means that when an element is inserted into the tree it is inserted in the bottom right position. This also means that when the tree becomes unbalanced it always triggers a left rotation. This can be used to compute how the elements will move.

The compression process can be reformulated as the following. Let N be an integer and $u = \{u_i \mid 0 \leq i \leq N-1\}$ the sorted sequence that we want to insert, we will construct a function $\varphi : [0, N-1] \mapsto [0, N-1]$ such that $[v]_b$ where $v = \{v_i = u_{\varphi(i)} \mid i \in \mathbb{N}\}$ is an AVL tree. The function φ is a map between the index of the elements of the sorted sequence and the elements of the compressed bread first array.

The following lemma will give us a mean to recursively construct the function φ and effectively compute the position of the elements in the compressed breadth first tree.

Lemma 1. *Let u be a sequence of size N sorted in increasing order, if all elements of u are inserted in order into an AVL tree, the root element of the resulting tree is the element at position $p(N)$ where:*

$$p(n) = \begin{cases} 2^{\lfloor \log_2\left(\frac{n}{3}\right)+1 \rfloor} - 1 & \text{if } n > 3 \\ 0 & \text{if } n < 3 \\ 1 & \text{if } n = 3 \end{cases}$$

The root is the first element in the breadth first representation, therefore $\varphi(0) = p(N)$. This splits the sequence u into two parts $\{u_i \mid 0 \leq i \leq p(N)-1\}$ and $\{u_i \mid p(N)+1 \leq i \leq N\}$ on which we can recursively apply the same technique to find theirs roots and the roots of their subparts recursively.

Example 22. Compression of the tree of Figure 4.1 (Depth First Layout) In the breath first layout tree of Figure 4.1 is represented as $[5, 4, 9, 2, [], 7, 11, 1, 3, [], [], 6, 8, 10, 12]_b$. The same sequence sorted becomes $[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]$.

Step 1 First, we compute $\varphi(0) = 7$, the 7^{th} element (0-based-indexing) is 8 and will be the root of the new tree. We are now left with two subsequences $[1, 2, 3, 4, 5, 6, 7]$ and $[9, 10, 11, 12]$.

Step 2 In this step, we will compute $\varphi(1)$ and $\varphi(2)$. $\varphi(1)$ is the position of the root in $[1, 2, 3, 4, 5, 6, 7]$ and $\varphi(2)$ is the position of the root of $[9, 10, 11, 12]$ with an offset to

account for the fact that the subsequence $[9, 10, 11, 12]$ starts at position 8 in the original sequence. Henceforth, we have $\varphi(1) = p(7) = 3$ and $\varphi(2) = 8 + p(1) = 8 + 1 = 9$.

Step 3 In this step, we will compute $\varphi(3)$ (associated to $[1, 2, 3]$ with an offset of 0), $\varphi(4)$ (associated to $[5, 6, 7]$ with an offset of 4), $\varphi(5)$ (associated to $[9]$ with an offset of 8), $\varphi(6)$ (associated to $[11, 12]$ with an offset of 10).

$$\varphi(3) = 0 + p(3) = 1 \qquad \varphi(4) = 4 + p(3) = 5$$
$$\varphi(5) = 8 + p(1) = 8 \qquad \varphi(6) = 10 + p(2) = 10$$

Step 4 In this last step we will compute $\varphi(7)$ (associated with $[1]$ with an offset of 0), $\varphi(8)$ (associated with $[3]$ with an offset of 2),$\varphi(9)$ (associated with $[5]$ with an offset of 4),$\varphi(10)$ (associated with $[7]$ with an offset of 6), $\varphi(11)$ (associated with $[]$, since there is nothing below 9),$\varphi(12)$ (associated with $[]$ for the same reason), $\varphi(13)$ (associated with $[]$ since there is nothing between below 11 in $[11, 12]$), $\varphi(14)$ (associated with $[12]$ with an offset of 11.)

$$\varphi(7) = 0 + p(1) = 0 \qquad \varphi(8) = 2 + p(1) = 2$$
$$\varphi(9) = 4 + p(1) = 4 \qquad \varphi(10) = 6 + p(1) = 6$$
$$\varphi(11) = \texttt{nothing} \qquad \varphi(12) = \texttt{nothing}$$
$$\varphi(13) = \texttt{nothing} \qquad \varphi(14) = 11 + p(2) = 11$$

In the end, the compressed tree is $[8, 4, 10, 2, 6, 9, 11, 1, 3, 5, 7, [], [], [], 12]_b$

As we will later see (in Section 5.1.5), the compression algorithm is a necessary tool for great performance and compacity of implicit breadth first trees.

4.2.3 Van Em Boas Representation

Previous representation comes from standard traversal and are easy to understand, however their locality properties are not the best. Indeed, in those representations a subtree can span over very large memory regions and regardless of how small the subtree is, there is no guarantee that the memory will be contiguous. The van Em Boas Representation was first presented by Prokop in his thesis [Pro99]. This representation is notably popular in the domain of cache-oblivious algorithms since it guarantees that, when walking down a tree, subtrees are more contiguous in memory.

The idea behind this representation is to consider a tree $\mathcal{T}$ of height h (a tree with one element has a height of 1). $\mathcal{T}$ is then divided into two subsets $\mathcal{T}_{h^-}$ — the set of elements which have a depth lower than half the total height — and $\mathcal{T}_{h^+}$ — the set of elements which have a depth greater than the total height. The set $\mathcal{T}_{h^-}$ contains only one root, the original root of $\mathcal{T}$, however the set $\mathcal{T}_{h^+}$ contains $\sqrt{2^h}$ roots which corresponds to the children in the original tree $\mathcal{T}$ of the leaves in $\mathcal{T}_{h^-}$. Figure 4.9 shows a tree after it has been split in two sets : $\mathcal{T}_{h^-}$ and $\mathcal{T}_{h^+}$.

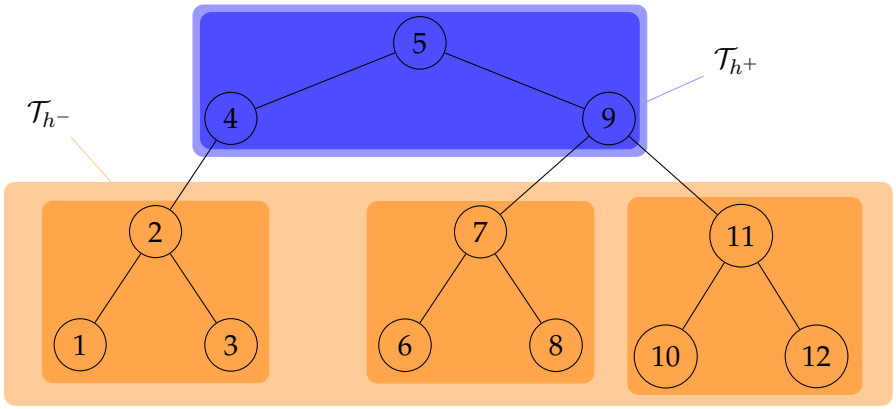


Figure 4.9: The tree of Figure 4.1 partitionned into $\mathcal{T}_{h^-}$ and $\mathcal{T}_{h^+}$, the darker rectangles are the elements of those sets.

Example 23. Let $\mathcal{T}$ be the complete tree with 1 element described in breadth-first representation as $[a]_b$. It is a tree of height one, and its van Em Boas representation is $[a]_v$.

Example 24. Let $\mathcal{T}$ be the complete tree with 3 elements described in breadth-first representation as $[a, b, c]_b$. It is a tree of height 2. It is subdivided into two sets $\mathcal{T}_{h^-} = \{[a]_b\}$ and $\mathcal{T}_{h^+} = \{[b]_b, [c]_b\}$. At this point we have to compute the van Em Boas representation of $[b]_b$ and $[c]_b$, which are one-element tree, hence their representation is trivial and the final van Em Boas representation of $\mathcal{T}$ is $[a, b, c]_v$.

Example 25. Let $\mathcal{T}$ be the complete tree with 7 elements described in breadth-first representation as $[a, b, c, d, e, f, g]_b$. It is a tree of height 3. It is subdivided into two sets $\mathcal{T}_{h^-} = \{[a]_b\}$ and $\mathcal{T}_{h^+} = \{[b, d, e]_b, [c, f, g]_b\}$. We have to apply recursively this process on each subtree of the previous sets. According to the previous examples, the van Em Boas representation of $\mathcal{T}$ is $[a, b, d, e, c, g]_v$.

Example 26. Let $\mathcal{T}$ be the complete tree with 15 elements described in breadth-first representation as $[a, b, c, d, e, f, g, h, i, j, k, l, m, n, o]_b$. It is a tree of height 4. It is subdivided into two sets $\mathcal{T}_{h^-} = \{[a, b, c]_b\}$ and $\mathcal{T}_{h^+} = \{[d, h, i]_b, [e, j, k]_b, [f, l, m]_b, [f, n, o]_b\}$. We have to apply recursively this process on each subtree of the previous sets. According to the previous examples, the van Em Boas representation of $\mathcal{T}$ is $[a, b, c, d, h, i, e, j, k, f, l, m, g, n, o]_v$.

Definition 19. A *local root* with respect to the van Em Boas representation is a terminal position: it is not possible to subdivide the tree anymore.

Before being able to define the access function `parent`, `left_child` and `right_child` we need to define a function `pos` (Figure 4.10) which constructs the path to the location where the index we are looking for is a local root (finding the local root takes $\log_2 h$ steps with h the height of the tree). The intuition on how the position of the local root is computed is as follows. At first, we have the whole tree. This tree is first split into $\mathcal{T}_{h^+}$

and $\mathcal{T}_{h^-}$. $\mathcal{T}_{h^-}$ contains one tree whereas T_{h^+} contains many trees which exactly the same number of elements (some might be holes). These trees can be considered as buckets. Our goal is to find in which bucket is the value we are searching the location of, and repeat this process on the bucket (which is also a tree) until we end up on a bucket with only one element.

Example 27. The tree $[1,2,3,4,5,6,7]_v$ represented with different buckets that appear when subdividing it.

$$\Big[[1]\Big[[2][3][4]\Big]\Big[[5][6][7]\Big]\Big]$$

```
def pos i,h # i = index, h = height
  if h == 1 then [0]
  else
    h1 = h / 2
    if i < (2 ** h1) - 1 then
      [0] + pos(i, h1)
    else
      bucket_sz = 2 ** (h - h1) - 1
      bucket_no = (i - (2**h1 - 1)) / bucket_sz
      bucket_ps = (i - (2**h1 - 1)) % bucket_sz
      [1 + bucket_no] + pos(bucket_ps, h - h1)
    end
  end
end
```

Figure 4.10: Pos function for the VEB layout: This function takes an index and the height of a tree and returns the list of buckets that needs to be traversed before landing on a local root.

Example 28. Let's take again the tree $\mathcal{T} = [a,b,c,d,h,i,e,j,k,f,l,m,g,n,o]_v$ from Example 26, the value which is stored at index 0 has position $\texttt{pos}(0,4) = [0]$, that means that it is directly the local root of the main tree. The value which is stored at index 7 is given by $\texttt{pos}(7,4) = [2,1,0]$ that means that we have to look in the second bucket, then look into the first bucket of this bucket and now this element is its own local root.

The computation of the function `pos` can easily be precomputed by using memoization. Once this function is computed it is possible to express the access functions `parent`, `left_child` and `right_child` in the same as previously, that is the position where the parent, the left child or the right child is locally a root. However, since we do not plan to use them in the rest, we do not provide formula for them. The fact, that they are computable proves that this layout preserve the tree structure. Moreover, the computation of those functions can be benefit from memoization due to the recursive nature of this layout.

4.2.4 Relative sequential performance of the layouts

Brodal et al. [BFJ02] gives an extensive comparison of four such "implicit" layouts and the associated tree operations (insertion, computation of the children of a node at position i, search, range queries–all elements with keys within a given interval–, deletion) and their relative memory transfer complexities. It also provides experimental evaluation of these operations, that we quickly summarize here:

- All static layouts perform better than their equivalent pointer versions for trees that do not entirely fit in a cache line.
- In implicit layouts, the *van Emde Boas layout* performs better than all other layouts, particularly at very large sizes (when the size of the trees does not fit in memory and partially reside on disk). The *Breadth First layout* behaves similarly or better at smaller size, and only slightly worse on very large trees.

In the rest of this thesis, we will focus on the *Breadth First layout*, as it is much simpler than the *van Emde Boas layout* and provides similar performances in most cases.

4.3 Conclusion

This chapter presented operations on trees that will be under concern for the rest of the thesis. For this study, we restricted ourselves to AVL trees. Inspired by the tradition of *cache-oblivious* algorithms, we decide to abandon the classical pointer-based layout. We explore three different linear memory layouts for trees and their characteristics, their strong points and drawbacks, and analyze each of them. The breadth-first search layout which will be the one used in the next chapters offers a regular layout with a simple way to access elements and can be easily updated: it thus provides a tradeoff between simplicity and performance.

CHAPTER 5

TARBRES: AVL TREES AS ARRAYS

Research Questions

? What are the optimization opportunities for AVL trees laid as arrays ?

? How to express polyhedral-like optimizations in our new setting ?

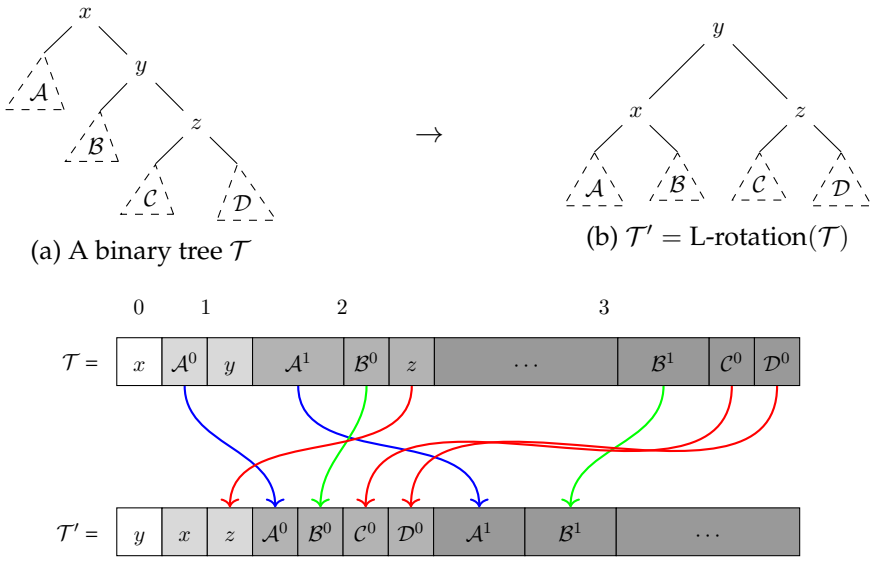
As we saw in the previous chapter, we can perform an efficient restructuration of trees, and especially binary trees, in linear layouts. In this chapter, we propose to make a step further and reexpress the classical operations on AVL trees (insertion, deletion) in terms of adequate composition of new *structural* low-level operations on the new array-based layout called Tarbre to make this memory representation efficient; each of them exposing nice opportunities regarding locality and parallelism. We propose an implementation and an experimental evidence of the pertinence of the approach.

5.1 Tarbres

We propose in this chapter to focus on Tarbres.

Definition 20. We call Tarbres AVL trees which have been laid out in an array according to a breadth-first numbering (cf Section 4.2.2). In the rest of this chapter, we denote them in calligraphic letters: $\mathcal{T}$.

As stated before, our goal is to demonstrate the potentiality of parallelism of Tarbre operations. In this section, we will show how classical operations on AVL trees can be expressed in terms of low-level operations operating on Tarbres, for which we provide sequential algorithms and a complexity analysis.



(a) A binary tree $\mathcal{T}$

(b) $\mathcal{T}' = \text{L-rotation}(\mathcal{T})$

(c) L-rotation on a Tarbre

Figure 5.1: Left (L) rotation applied on to the unbalanced tree T, and the associated transformation on the Tarbre representation.

5.1.1 Layout and tree operations

Before delving into more details, we present here for reference a simplified version which captures the gist of the internal structure which we used to represent Tarbres in memory:

```
struct tarbre {
  int * elems; // Array of elements
  size_t * depths; // Array of depths
  size_t nb_elems; // Number of values
  size_t len; // Real size of the support
}
```

Our tree is stored in a memory support consisting of two arrays, `elems` (for the values) and `depths` (for the depth of the subtree rooted in this element). `nb_elems` contains the real number of elements in the tree while `len` is the length of the underlying support. In the rest of the chapter, we will focus only on layout changes and movements of elements, and thus forget the `depths` field; however it should be kept in mind that all operations should also modify this field.

To describe the behavior and operations of Tarbres, we make use of some nice properties and notations coming from the breath-first numbering described in Section 4.2.2 and recalled below.

Prop 2. *In Tarbres:*

- *The two children indices of node i are indexed by $2i$ and $2i+1$.*
- *Unless i is the root, its parent has the index $\frac{i}{2}$.*
- *Each level k of the tree (defined by the distance to the root) is stored in the sub-array with indices belonging to the range $[2^k, 2^{k+1}-1]$ (thus is of size 2^k).*

Definition 21. Notations and access functions for a given Tarbre $\mathcal{T}$:

- `size(`$\mathcal{T}$`)` denotes the size of the Tarbre, *i.e.*, the number of allocated cells for $\mathcal{T}$. This quantity is always greater or equal to the number of nodes of the underlying tree.
- `depth(`$\mathcal{T}$`)` denotes the depth of the corresponding tree.
- The adjacent cells of the Tarbre $\mathcal{T}$ corresponding to level k will be denoted by $\mathcal{T}^k$ and called k^{th} *layer* of $\mathcal{T}$.
- All layers of the subtree whose root is indexed by i can be computed using the function `layers(`$\mathcal{T}$`,`i`)`.
- $i_{\mathcal{T},\mathcal{A}}$ denotes the root index of the subtree $\mathcal{A}$ in the tarbre $\mathcal{T}$. $\mathcal{T}$ is omitted whenever possible.
- The function `copy(`$\mathcal{T}$`, src, dest, nb)` makes a copy of nb (adjacent) cells from src to dest in the Tarbre. $\mathcal{T}$ is omitted whenever possible. dest should be of size equal or greater than src; the function fills the remaining cells with placeholders if required.

Most tree data-structures are read much more often than they are modified. As such an implicit layout offers many advantages: read-only operations benefits from the cache-friendly compact structure to offer great practical speed. The challenge is naturally to still provide reasonably fast modifications.

5.1.2 Tarbres operations

Tarbres support read operations such as `find`, `range` queries (returns all values for keys in a given range of values), or iterations (`iter`, `map`, `fold`, ...).

Find The code of the `find` operation (which return true if the value is found, false otherwise) is shown in Figure 5.2 and is an adaptation of Figure 4.5 specialized with the access function of the breadth first layout. All functions on AVL trees can be adapted to to Tarbres this way.

Range and iteration operations have similarly simple code.

Insert and rotations The implementation of `insert` is the same as traditional AVL trees. It relies on a balancing operation implemented through rotations. Traditionally, a tree-rotation is a cheap operation which: 1. moves around two pointers and 2. updates the information about the depths and the balance ratio of the nodes. However, when trees are internally represented as arrays, rotations incurs several large-scale memory copies that can potentially affect the whole array. This has no influence on the correctness of these operations, however this can strongly impact the performance.

```
def find(𝒯, val)
"""Dichotomic search. Returns True is `val' is
  found, False otherwise"""
 node = 1
 while (node < 𝒯.len && 𝒯[node].depth) > 0):
   if (𝒯[node].val > val):
     node = 2*node # left node
   elif (𝒯[node].val < val):
     node = 2*node + 1 # right node
   else
     return True
 return False
```

Figure 5.2: Pseudo-code for the `find` operation in a Tarbre.

To demonstrate this, let us take the example of an L-rotation illustrated in Figure 5.1 already depicted in the preceeding chapter.

The Tarbre representation using an array is shown in Figure 5.1c. Each section of the array is identified by its content, which corresponds to a layer of a given subtree. As stated in Définition 21, the k^{th} layer of a tree $\mathcal{A}$ is noted $\mathcal{A}^k$. For instance $\mathcal{A}^0$ is the root of $\mathcal{A}$, while $\mathcal{B}^1$ is the children of $\mathcal{B}$ at depth 1. By looking carefully at the representation of the Tarbre before and after the rotation, we can distinguish several categories of memory movements. The subtree $\mathcal{A}$ is moved down one depth lower. This is directly transcribed in Figure 5.1c by the blue arrows, which move the specified memory sections to the right (i.e., "downwards" in the tree). The subtree $\mathcal{B}$ stays at the same depth, but moves to another position, as represented by the green arrows. The subtree rooted in z (and containing $\mathcal{C}$ and $\mathcal{D}$) moves upward, which corresponds to a left move in the array representation, represented by the red arrows.

As illustrated in this example, rotations are implemented by a set of memory movements on subtrees. A subtree is represented by several memory ranges, exactly one per depth level of the subtree. We can also note that memory ranges of the moved subtrees overlap, even among ranges belonging to a single subtree (see subtree $\mathcal{A}$ for instance). Although Figure 5.1 might thus give the impression that we need to copy the whole Tarbre in a new array while doing a rotation, rotations can be implemented in-place efficiently thanks to a careful decomposition into lower-level operations which we present in the next section.

5.1.3 Low-level operations on Tarbres

Rotations are an instance of more general structural transformations on Tarbres. Many such transformations can be decomposed as compositions of low-level operations. As presented in Section 4.2.2, the breadth-first ordering of Tarbres provides a convenient index scheme which allows to view the array as a collection of layers. We can manipu-

late these layers thanks to low-level classes of operations, *shifts* and *pulls*.

Shifts (Figure 5.3) move a subtree from one position to another in the tree, as long as these positions do not overlap. In Figure 5.1, the movement of $\mathcal{B}$ represented by **green** arrows is a shift. Values which are moved overwrite previous values if any. The shift operation can be performed on subtrees rooted at any place in the Tarbre. The pseudo-code for the shift operation is described in Figure 5.4 and proceeds simply by doing memory copies of each layer. Note that since the subtrees do not overlap, these memory copies can occur in any order, which will be important for parallelism. The cells "emptied" by the move of $\mathcal{A}$ are not filled with special values: this operation will be followed by other operations that will fill the layers with relevant values.

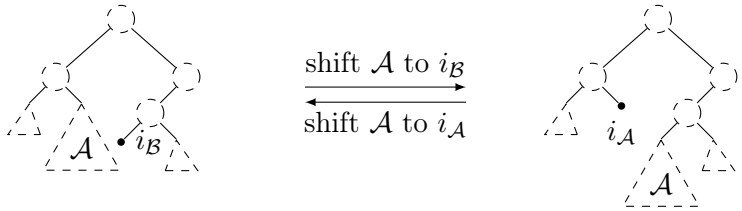


Figure 5.3: Subtree shifts

```
def shift(T, i_A, i_B):
  A = layers(T, i_A)
  B = layers(T, i_B)
  for k = 0 to depth(A):
    # copy each layer from A to B
    copy(A^k,B^k,2^k)
```

Figure 5.4: Pseudo-code for the `shift` operation.
Shifts the subtree of $\mathcal{T}$ indexed by $i_{\mathcal{A}}$ at the position denoted by the index $i_{\mathcal{B}}$. The considered subtrees should not overlap.

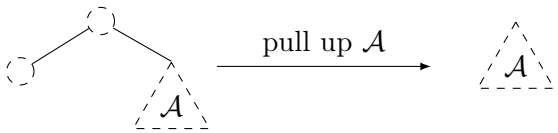


Figure 5.5: Subtree pull-ups and pull-downs

```
def pull_down_left(T,i_A):
  A = layers(T, i_A)
  for k = depth(A) to 0: # from bottom to top
    copy(A^k,A^{k+1},2^k)
```

Figure 5.6: Pseudo-code for the `pull_down_left` operation.
Pulls down the subtree of $\mathcal{T}$ indexed by $i_{\mathcal{A}}$ to the left.

Rotation		Right Figure 4.4a	Left Figure 4.4b
Steps	1.	pull-down-right($\mathcal{T}, i_{\mathcal{D}}$)	pull-down-left($\mathcal{T}, i_{\mathcal{A}}$)
	2.	shift($\mathcal{T}, i_{\mathcal{C}}, i_{\mathcal{C}} + 1$)	shift($\mathcal{T}, i_{\mathcal{B}}, i_{\mathcal{B}} - 1$)
	3.	pull-up($\mathcal{T}, i_z$)	pull-up($\mathcal{T}, i_z$)
	4.	move-values($\mathcal{T}, i_x, i_y, i_z$)	move-values($\mathcal{T}, i_x, i_y, i_z$)
Rotation		Right-left Figure 4.4c	Left-right Figure 4.4d
Steps	1.	pull-down-left($\mathcal{T}, i_{\mathcal{A}}$)	pull down-right($\mathcal{T}, i_{\mathcal{D}}$)
	2.	shift($\mathcal{T}, i_{\mathcal{B}}, i_{\mathcal{T}.\text{left.right}}$)	shift($\mathcal{T}, i_{\mathcal{C}}, i_{\mathcal{T}.\text{right.left}}$)
	3.	pull-up($\mathcal{T}, i_{\mathcal{C}}$)	pull-up($\mathcal{T}, i_{\mathcal{B}}$)
	4.	move-values($\mathcal{T}, i_x, i_y, i_z$)	move-values($\mathcal{T}, i_x, i_y, i_z$)

Figure 5.7: Tarbres rotations decomposed as low level operations. The function `move-values` reorders the three indexed elements to keep them sorted

Pull-downs (Figure 5.5) take a subtree and graft it at the place of its left or right child. In Figure 5.1, the movement of $\mathcal{A}$ represented by **blue** arrows is a left pull-down. Unlike the previous operation, this operation is not destructive, but instead "frees" a subtree, which can later be filled through one of the other operations. The pseudo-code for the left pull-down operation is described in Figure 5.6. It copies each layer to the layer directly underneath. This time, the copies must be done from bottom to top to avoid overlaps. Note that the cells only occupy half of their new layers: the other subtree is left empty.

Pull-ups (Figure 5.5) take a subtree and graft it in place of its parent. In Figure 5.1, the movement of z, $\mathcal{C}$ and $\mathcal{B}$ represented by **red** arrows is a pull-up. This is a destructive operation in the sense that the parent and one of its children subtree are overwritten. This operation cannot be performed on the root. The code for this operation is not presented, as it is similar to the pull-down operation, but in reverse.

These low-level operations can be applied on all kinds of breadth-first arrays, and do not preserve the balancing property of AVLs by themselves. We can now combine them to implement rotations.

5.1.4 Rotations as sequences of low-level operations

Intuitively, we can see that pull-downs free a subtree, shifts consume then free a subtree, and pull-ups consume a subtree. In order to use these operations to implement structural transformations such as AVL rotations which do not erase any data, we must compose sequences of operations similarly to a sliding puzzle[1]. Such sequences are

[1] https://en.wikipedia.org/wiki/Sliding_puzzle

```
def compress(T):
  # A DFS of T iterate in sorted order
  sorted_values = DFS(T)
  # Inserting monotone values in a tarbre
  T = tree_of_sorted_values(sorted_values)
```

Figure 5.8: Pseudo-code for Tarbre compression

presented for each rotation (from Figure 4.4) in Figure 5.7. Each sequence is composed of three large-scale movements on subtrees, one pull down, one shift, one pull up, along with trivial movements on individual values.

The left rotation, which was already presented in Figure 5.1, can thus be synthetically described as the following sequence: pull-down $\mathcal{A}$ to the left, shift $\mathcal{B}$ to the left, pull-up the subtree rooted in z and place correctly the values x, y and z. Other rotations are described similarly in Figure 5.7. For each shift, we note that the subtrees are clearly disjoint in the concrete operations used for the rotations.

5.1.5 Layout Density and Compression

When inserting an element in a Tarbre (cf Figure 4.3), there is a temporary state in which the Tarbre is possibly not balanced, which in our layout means that we might have to allocate a new layer. However, unless the considered tree is perfectly balanced, the Tarbre is filled with placeholders waiting to be assigned a value and the allocation is useless. This property avoids intermediary allocations, but might result in a lot of wasted space as layers are allocated which would not be required by a better balancing.

Concretely, we want to avoid Tarbres with low density (the number of relevant values divided by the size of the underlying array). Thanks to Fibonacci trees[2], we can evaluate how low this density can be. The Fibonacci tree of depth n is composed of two subtrees, the Fibonacci trees of depth $n-1$ and $n-2$ while the trees 0 and 1 are the trees with zero or one element, respectively. A Fibonacci tree is "the most imbalanced AVL": the depth difference of two sibling subtrees is always exactly one.

Prop 3. *Let us consider Fibonacci Tarbres, where a Fibonacci tree uses the Tarbre representation. If F_n is the n^{th} Fibonacci number, the Fibonacci Tarbre of depth n has density $(F_n - 1)/2^n = \mathcal{O}((\phi/2)^n)$.*

Proof. The Fibonacci tree of depth n has $F_n - 1$ elements. Its Tarbre support is of size 2^n, which gives us a density of $F_n/2^n$. Furthermore, we have $F_n \sim_{n\to\infty} \phi^n/\sqrt{5}$, which concludes. □

In practice, this means that density can be arbitrarily low for large trees. For depth $n = 15$, we can already obtain densities as low as 0.018. To solve this issue we use

[2] https://en.wikipedia.org/wiki/Fibonacci_number#Computer_science

a simple compression algorithm like in Section 4.2.2, recalled in Figure 5.8, in which sorted values are extracted from the Tarbre and used to build a perfectly balanced Tarbre. Building a Tarbre from sorted data is linear through a simple partitioning algorithm.

Prop 4. *The compress algorithm has time complexity $\mathcal{O}(n)$ where n is the size of the considered sub-Tarbre.*

This compression algorithm is triggered by insertion and deletion operations when the density of the considered Tarbre falls below a chosen threshold. We determine the best threshold experimentally in Section 5.3.

5.1.6 Algorithmic complexity

From the pseudo-codes of the low-level operations shifts and pulls, we immediately obtain an algorithmic complexity of $\sum_{k=0}^{d-2} 2^k = O(2^d)$ copies, where d is the depth of the Tarbre; thus:

Prop 5. *Rotations, thus insertions (and deletions) in Tarbres are of complexity $\mathcal{O}(2^d)$ where d is the depth of the considered sub-Tarbre, ie $\mathcal{O}(n)$ if n is the size of the underlying sub-arrays in memory.*

The Tarbre layout induces a clearly worse complexity than classical AVLs for which all the operations have complexity $\mathcal{O}(depth)$ (cf Section 4.1.3). However, the code of the low-level operations expose some parallelism that we will exploit in order to reduce the practical complexity of Tarbre operations.

5.2 Parallelism

So far, we have defined Tarbre operations in terms of simple low-level operations that copy pieces of the tree from one area in memory to the other. While such transformations already offer good locality, performances can be further enhanced by parallelizing the structural transformations implemented in the previous section.

5.2.1 Shifts

The pseudo-code implementation of the `shift` operation (Figure 5.4) exposes a sequence of independent memory copies. The implementation in C described in Figure 5.10 makes use of this independence and Figure 5.9 schematizes its action on the layers of a Tarbre. The source region does not overlap with the destination region and the source pointer and region pointer are guaranteed not to alias. The C99 signature of `memcpy`, which uses restrict pointers, helps the compiler detect and exploit that property. The C code also makes it clear that each layer can be moved independently of the others, allowing us to directly parallelize the `for` loop thanks to an OpenMP `#pragma`.

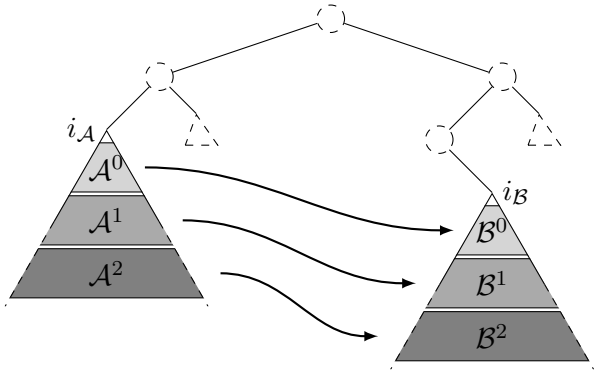


Figure 5.9: Shift memory movements

```
1 void shift (int* t, size_t len, int idxA, int idxB) {
2    int start_lvl = _greatest_bit_pos(idxA + 1);
3    int end_lvl   = _greatest_bit_pos(len - 1);
4
5  #pragma omp parallel for
6  for (int i = 0 ; i < end_lvl - start_lvl + 1 ; ++i) {
7    int depth = 1 << i;
8    int size  = depth * (sizeof *t);
9    int src   = depth * (idxA + 1) - 1;
10   int dst   = depth * (idxB + 1) - 1;
11   memcpy(t + dst, t + src, size);
12 }
13 }
```

Figure 5.10: Parallelized C implementation of the shift operation.
The two first lines compute the range of levels to be copied from $\mathcal{A}$. Adjacent cells belonging to the same level are copied independently of the other levels.

Another room for improvement comes from the fact the data could be moved in small chunks whose size should depend on the features of the processor such as its cache size and the size of the registers used by its vector unit. This is especially helpful for big layers since `memcpy` is always single-threaded. Splitting the data in chunks allows even more parallelism since the copy can be distributed evenly on all processors.

5.2.2 Pull-ups and pull-downs

Pulls are operations which move the content of a subtree either upwards or downwards. As before we use `pull-down-left` as our leading example but all this section also applies to other pulls. Contrary to shifts, the memory movements made by the naive implementation of pulls (such as Figure 5.6) are *not* independent, which prevents direct parallelization.

Fortunately, we can do better thanks to clever code transformations. We first showcase

```c
void naive_pull_down_left(int* t, size_t len, int idx) {
  int start_lvl = _greatest_bit_pos(len - 1);
  int end_lvl   = _greatest_bit_pos(idx + 1);

  for (int i = start_lvl ; end_lvl <= i ; --i) {
    int depth = 1 << (i - end_lvl);
    int size  = depth * (sizeof *t);
    int dest  = depth * (2 * (idx + 1)) - 1;
    int src   = depth * (idx + 1) - 1;

    memcpy(t + dest, t + src, size);
  }
}
```

Figure 5.11: Naive C implementation of Pull-down Left
The layers to be copied start respectively at index $log_2(len-1), \ldots, log_2(idx)+1, log_2(idx)$. Each layer is moved to the index just below.

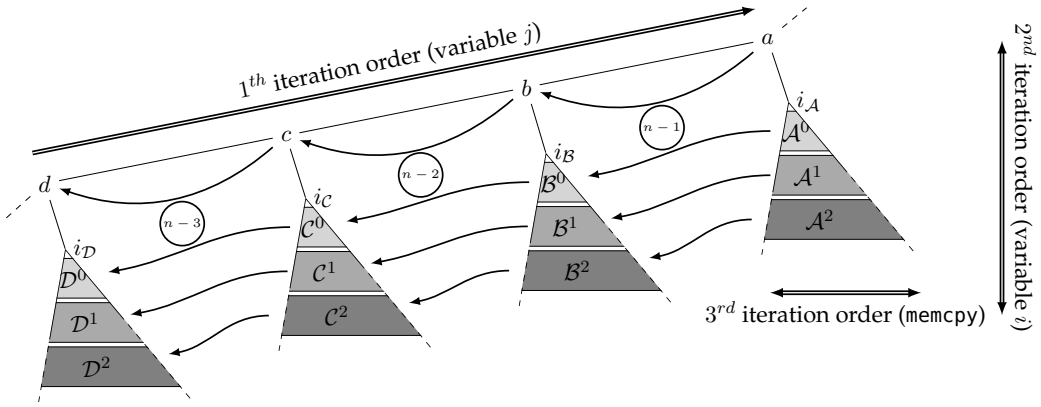


Figure 5.12: Optimized `pull_down_left` memory movements

a naive C implementation of the pull-down left operation in Figure 5.11 which corresponds to the pseudo-code in Figure 5.6. This naive implementation proceeds by iterating from the bottom of the subtree to the top, and moving each layer downward and to the left. As we noted before, the memory copies made by each loop of the naive pull operation conflicts with each other. Therefore, this version would only be parallelized through chunking, as previously described in the `shift` implementation.

However, we can transform this loop to be much more amenable to parallelization thanks to a transformation similar to loop skewing [Laf10]. The idea is presented in Figure 5.12 and implemented in Figure 5.13. The double arrows in Figure 5.12 present the iteration axes in regard to the tree structure. We iterate along the leftmost branch of the subtree (variable j). For each node in this leftmost branch, we copy each layers of the right subtree downwards along the branch (variable i). For instance, in the case presented Figure 5.12, the $(n-2)$-th step of the iteration operates on the node b and its

```c
void pull_down_left(int* t, size_t len, int idx) {
  int start_lvl = _greatest_bit_pos(idx + 1);
  int end_lvl   = _greatest_bit_pos(len - 1);

  for (int j = end_lvl ; j >= start_lvl ; --j) {
    int cur_root = (idx + 1) * (1 << (j - start_lvl));
    t[2 * cur_root - 1] = t[cur_root - 1];

    #pragma omp parallel for
    for (int i = 0 ; i < end_lvl - (j + 1) + 1 ; ++i) {
      int depth = 1 << i;
      int size  = depth * (sizeof *t);
      int src   = depth * (2 * cur_root + 1) - 1;
      int dest  = depth * (4 * cur_root + 1) - 1;

      memcpy(t + dest, t + src, size);
    }
  }
}
```

Figure 5.13: Optimized Parallelized C implementation of Pull-down left
Layers as described in Figure 5.12. For each layer, elements are copied by chunks of a predefined size in order to maximize the performance of `memcpy`.

right subtree $\mathcal{B}$. We then move each layer of $\mathcal{B}$ downward to the root $i_{\mathcal{C}}$. This "frees" the subtree rooted in $i_{\mathcal{B}}$. We then look at a and its right subtree $\mathcal{A}$, and copy its layers to the root $i_{\mathcal{B}}$.

The internal iteration now moves layers from one subtree to another with non-overlapping roots. This exactly corresponds to a shift, and we can apply the same optimizations and move the layers in parallel in a chunked manner. The loop nesting also opens up opportunities to *pipeline* the operations between layers. Indeed, if we look at the example in Figure 5.12, if layer $\mathcal{B}^1$ has been moved, regardless the status of the other layers, we can start moving $\mathcal{A}^1$ in its place. Unfortunately, while this optimization should provide significant gains, it is so far not automatically done by compilers as we will now see. We did not implement it due to the complexity of implementing it by hand.

These transformations, which are unlocked by skewing the iteration, directly apply to other pull operations, by iterating either top-bottom or bottom-up along the left- or right-most branch of the subtree.

5.2.3 Polyhedral Interpretation

The optimizations on the low-level operations `shift`, `pull-up` and `pull-down` resemble the kind of transformations that can be automatically performed by the polyhedral model. We thus now study these optimizations under this light.

Shifts (cf Figure 5.4 and Figure 5.10)

When copying $\mathcal{A}$ into $\mathcal{B}$, each layer $\mathcal{A}^k$ is independent (there is no read/write dependence) with all layers of $\mathcal{B}$ (globally, without looking at the iteration number k). This information is enough to ensure that the parallel `for` in Figure 5.10 is correct and could be used further to enable vectorization.

However, there are some difficulties to prove that the moves are independent:

- `idxA` and `idxB` are "sufficiently disjoint" when calling the `shift` operation. This could be added as assertions (since it comes from the calling context depicted in Figure 5.7).
- Even with this information, we still need to prove that $\mathrm{t} + \mathrm{src} + \mathrm{size}$ never overlaps with $\mathrm{t}+\mathrm{dest}$; which also means being able to make arithmetical proofs which captures the exponential behavior of 2^i.

Although rather simple by hand, integer arithmetic with exponentials is not easily automated and the independence of the loops over i is not captured by the polyhedral model. In particular, we tried to use PLUTO [BBK+08, BHRS08] on a modified program where the memory copy has been expanded to a sequence of individual cell copies. Without any surprise, the tool fails to parallelize the loops as the number of iterations of the inner loop depends on the iteration variable of the outer loop in a non-linear fashion.

Pull-down-left (cf Figure 5.6, Figure 5.11 and Figure 5.13)

The skewing process consists in making a base change from $\langle i \rangle$ to $\langle j, i \rangle$, the component j being the direction "along the left-most branch", and i the depth direction. After this base change, the instance of $\texttt{memcopy}_{\langle i,j \rangle}$ executed in loop number $\langle i, j \rangle$ only directly depends on $\texttt{memcopy}_{\langle i,j-1 \rangle}$, which enables parallelization of the loops in direction i, and pipelining along the j direction. For the same reasons as for shifts (morally, this operation is a clever variant of shifts), even after a manual skewing, the polyhedral model tools are not capable of finding those dependences and cannot parallelize. Finding the skewing direction automatically is even harder; however it seems natural to identify the "left-most branch direction" or "the right-most branch direction" as the new "base axis" when dealing with trees. This example also shows room for the definition of a new shape for tiling, resembling staircases with steps of size 2^i.

Through this prism, the base operations on Tarbres highlight a clear relationship between our manual parallelization effort and the philosophy of "regularity" that is at the core of the polyhedral model. Our operations exhibit regularity in two ways: a structural regularity captured by the "father-son" relation along the left-most or right-most branch; and a computation regularity through "shapes" which are not affine but of size 2^i.

5.2.4 Implementation

A first prototype for Tarbres has been implemented in C++ and his available online [3].

We also provide a reference implementation of traditional pointer-based AVLs, which is used for testing and benchmarks. For these two versions, the operations of insertion, find, delete and maps are provided. We implemented parallelization through two means:

- an OpenMP implementation using `#pragma` as described above. However, it turns out that OpenMP fails to capture the parallelism of the Pull operations. Indeed, OpenMP cannot parallelize the inner loop because the ending condition $i < \text{end\_lvl} - (j+1) + 1$ is dependent on the iteration of the outer loop j.
- We therefore also made an implementation with manual uses of pthreads with a reusable pool of threads. This implementation uses the fact that the layers in the shift operation can be moved independently and, further split the `memcpy` calls into multiple `memcpy` calls (chunking). This is because `memcpy` is single-threaded, and it is better to launch as many as `memcpy` tasks as possible.

5.3 Experiments

Our experiments are performed on micro-benchmarks as well as two representative scenario usages. They demonstrate that the performance of the sequential version of Tarbres is similar or better than the explicit pointer representations, and the optimized version of Tarbres show great acceleration on parallel machines.

In this section, we present and analyze our experimental results. Our goal is to: 1. tune our implementation, notably compression 2. evaluate Tarbres on various types of loads and complete use-cases 3. evaluate our parallel implementations on synthetic micro-benchmarks. The experiments have been done on a machine equipped with an Intel® Xeon™ Gold 6130 CPU @ 2.10GHz, for a total of 32 cores, 377GB of RAM and 22.5MB of cache. All code is compiled with `-O3`.

5.3.1 Tuning and compression

In Section 5.1.5, we use a notion of *threshold* to trigger compression. When density is too low, locality and access performances starts to degrade and size overhead grows. We must therefore preserve the density of Tarbres to preserve good performances. This is similar to [BFJ02] which uses a density criterion to balance cache-aware search trees. To properly tune this parameter, we measure the time required to insert 2^x elements in a Tarbre, depending on this density threshold. The results are given in Figure 5.14 with one curve per x.

[3] https://gitlab.inria.fr/paiannet/calv/-/tree/next

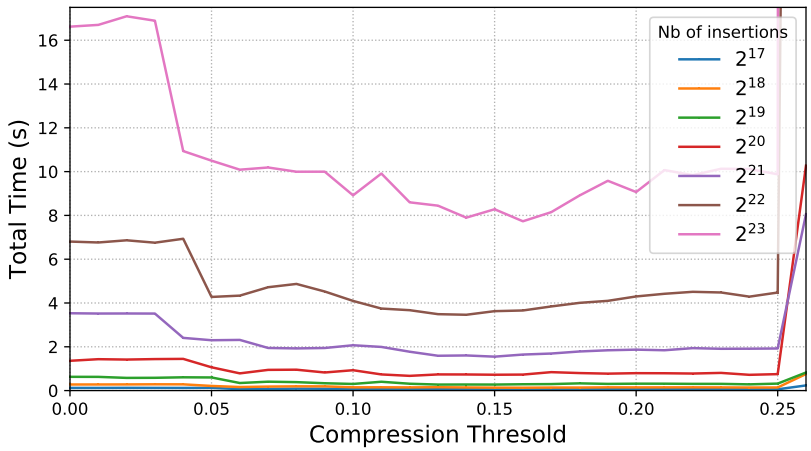


Figure 5.14: Performance of Tarbre creation in function of the compression threshold for several sizes.

We immediately observe that the time taken explodes as soon as we use a threshold greater than 0.25, which is an immediate consequence of the fact that AVLs trees are at most unbalanced of one level: from a tree with a density slightly above 0.25, adding one element will very often add a new level, and in consequence decrease the density drastically. We also observe that for small numbers of elements (less than $2^{17} \approx 130000$), compression does not really matter. On bigger Tarbres, while the measures are fairly unstable, it seems a threshold of around 0.15 provide the best performances. We set this parameter for the remaining benchmarks.

5.3.2 Macro benchmarks

Map operations

As a first simple measurement, we consider the in-place sequential `map` operation on trees. This benchmark aims to capture the performance behavior of all iterations on whole trees, which are usual operations provided on dictionaries for instance. Our expectation is that iterations should be very favorable to Tarbres compared to AVLs, as this is where cache-friendliness should shine. To focus on this aspect and avoid mixing parallelism concerns, we only consider *sequential* implementations here. Figure 5.15 shows the performance results for trees of size 2^{13} to 2^{25}. The vertical axis shows the time in seconds on a log scale. As expected, Tarbres, using the implicit layout, are around 10 times faster than pointer-based AVLs. Other iteration and range operations are also similarly fast using Tarbres.

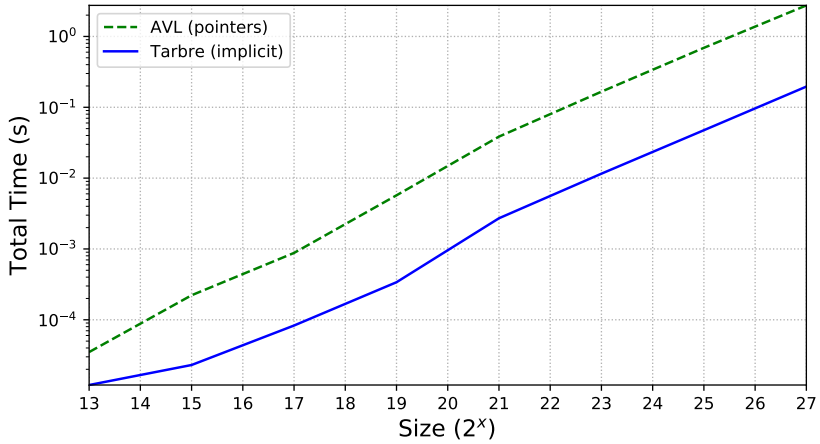


Figure 5.15: Performance of the in-place `map` operation on trees
Average timing of a single operation (log/log scale), for growing sizes of Tarbres (2^x is the size of the underlying array). The computation of the average is done by a program that sequentially runs 100 times the shift operation.

Scenario – Naming Environment

We now consider a more complex scenario to evaluate the practical performances of Tarbres: a dictionary used as naming environment in a compiler. Indeed, compilers, in particular during name resolution and type-checking, heavily rely on efficiently adding and finding names of functions and variables. To evaluate this use-case, we considered the OCaml type-checker. OCaml [LDF+21] is a functional statically-typed language known for its rich type system and efficient compiler. Typing in general and name resolution in particular is a fairly performance-sensitive operation, and the type-checker uses a pointer-based AVL as name environment. An additional challenge is that this environment is used in a *persistent* manner. Since variables respects lexical scoping in OCaml, new variables are registered in a new independent naming environment that is discarded when the scope under consideration is closed.

We instrumented the implementation of the naming environment to log all its operations, so that we can replay them with different tree implementations. Naming environments and names are represented by unique identifiers. We also logged when a scope is closed, to indicate that a particular version of the naming environment is freed. A short excerpt of the log is shown below which demonstrates the type of usage pattern found in this scenario. The `add` and `addl` operations indicate *persistent* insertions of one or multiple values, `find` des a lookup in the tree, and `free` frees the tree. Tree 0 is empty.

```
1223 <- addl(0,kind_478,layout_479) // Tree creation
1224 <- add(1223,arr_483) // Tree extension
find(1224,arr_483) // Lookup in tree 1224
find(1224,c_init_460)
...
free(1224) // Tree 1224 is freed
find(1223,create_430) // Lookup in tree 1223
```

We created the log of operations done by the naming environment when compiling the OCaml standard library and replay it in our Tarbres setting. The results are shown in Figure 5.16. As we have seen, the persistent usage in this scenario makes it particularly advantageous for pointer-based representation, since persistence is cheap for such implementations. Nevertheless, sequential Tarbres are slightly faster than pointer-based AVL trees on this benchmark. This shows that even on persistent workloads, the overhead of linear operations is well-compensated by the improved locality on small to medium-sized trees. We also observe that the trees stay dense (above 50%), which makes the compression irrelevant in this particular case.

	AVL (pointers)	Tarbre (implicit)
Time (s)	$1.5 \pm 0.01s$	$1.4 \pm 0.01s$
Average memory	11.5Mo	12.5Mo

Figure 5.16: Experimental results for name-resolution operations

Scenario – A key-value database

Key-value databases are big dictionaries which are generally implemented using variants of B-trees. To evaluate Tarbres in a similar context, we used the scenario described by the Influxdb team [4] to compare several key-value stores such as LevelDB (which uses Log Merge Trees) and LMDB (which uses B+-Trees). We do not attempt to compare our performance against their (highly fine-tuned) implementation, but we use their scenario to evaluate our sequential Tarbre implementation against the pointer-based one. The scenario is as follows:

1. Insert n random keys in a fresh database;
2. Delete $n/2$ random keys;
3. Compress the database;
4. Read $n/2$ random keys;
5. Insert $n/2$ random keys.

The experiment has been conducted with $n = 100$K and $n = 10$M with similar results. The results of this experiment with $n = 1$M can be seen in Figure 5.17. Timewise, our Tarbre layout either beats or is as good as standard AVL trees. In particular, there is a

[4] https://www.influxdata.com/blog/benchmarking-leveldb-vs-rocksdb-vs-hyperleveldb-vs-lmdb-performance-for-influxdb/

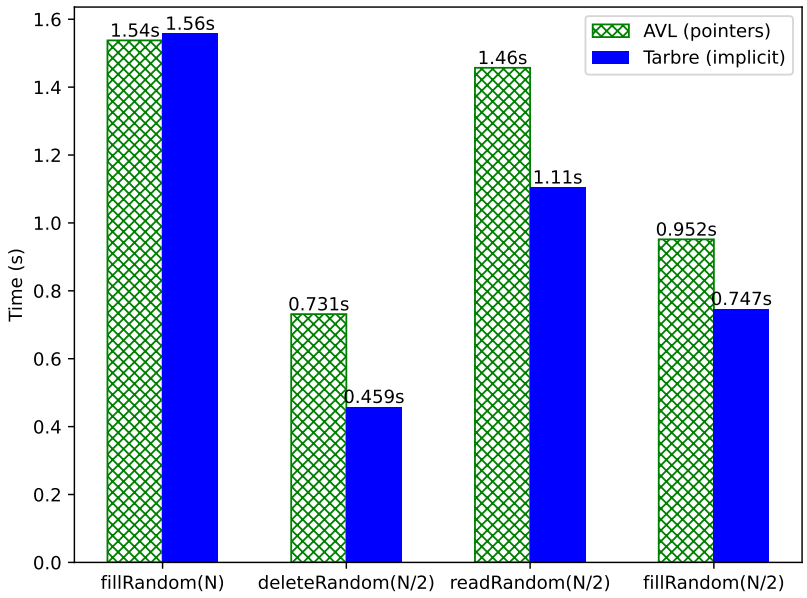


Figure 5.17: Performance of sequential Tarbres vs standard AVL trees on each part of the scenario with $n = 1000000 (\approx 2^{20})$

noticeable speed-up on 'delete' and 'read' operations. The speedup becomes even more noticeable as n grows. However, this speed up as a price, for $n = 1$M, standard AVL trees use 73MB where Tarbres use 206MB, this difference is mostly due to the number of placeholders the Tarbre implementation has to keep tabs on. As is common in key-value stores, the compression step is explicit and called manually. While compression is automatic for Tarbre, an explicit call before starting the read heavy section to maximize the Tarbre density helps. The total time for Tarbres is 4.01s without the manual compression step and 3.86s with manual compression. The total time for AVL trees is 4.77s.

This scenario shows that the *sequential* implementation of Tarbres is already competitive with traditional AVL trees on varied workloads. Let us now look at the gain provided by parallelization.

5.3.3 Parallel Micro Benchmarks

To evaluate our parallelization effort in isolation, we measure each low-level operation in turn: Shift, Pull-up et Pull-down. We compare three implementations: a *sequential* one, which is a straight C++ implementation of the pseudo-code presented in Section 5.1.3. We also implemented two parallel implementation using the strategies described in Section 5.2: one using OpenMP, the other directly with pthreads.

Figure 5.18, Figure 5.19 and Figure 5.20 show the results for `shift`, `pull-down` and `pull-`

`up` respectively. They report the average time of the given operation as a function of the size of the Tarbres. As expected, for all operations under study, the sequential version is clearly better than the other ones for "small" Tarbres ($\leq 2^{21}$); and its experimental complexity grows linearly with the size of the Tarbres (as proved in Proposition 5). As we already mentioned, the OpenMP version performs badly, our main hypothesis being that the allocation of threads follows a pattern that is incompatible to our parallelization scheme. In contrast, the pthread version which uses a pool of threads tailored to call `memcopy` behaves well; for big sized-arrays, the performance is an order of magnitude better than the sequential version.

These benchmarks show that we were able to manually exploit the parallelism that we had in mind by proposing the low-level operations on Tarbre. We nevertheless believe that there is still room for improvement using pipelining or vectorization.

5.4 Conclusion

Throughout this chapter, we have used *AVL trees* to demonstrate the capabilities of the breadth first layout when it comes to low-level structural reshaping, notably via our Tarbre library implementation. We posit that the optimizations we exhibit directly apply to a broader class of self-balanced trees which inherently rely on rotations such as red-black trees or splay trees. There are currently two limitations to our approach. The first one is that we have yet to thoroughly vectorize the low-level operations presented in this chapter. And the second one, which is also the main reason why we do not present a full parallel scenario in our benchmark is that when all the low-level operations are put together, all the benefit is lost when the threads synchronize. This is mostly due to the fact that the low-level operation presented here are still too coarse grained when it comes to pipelining. Pipelining is tedious and error-prone to do by hand, and the next chapter presents our solution to generate pipelined code.

Furthermore, we also believe the optimizations we presented in this chapter could be performed to some extent automatically by compilers. In particular, our model exhibit vectorization opportunities that could be exploited through tiling and pipelining which have been successfully applied in the context of the polyhedral model. In the next chapter, we will provide a first step toward a polyhedral model operating on tree-shaped data structures.

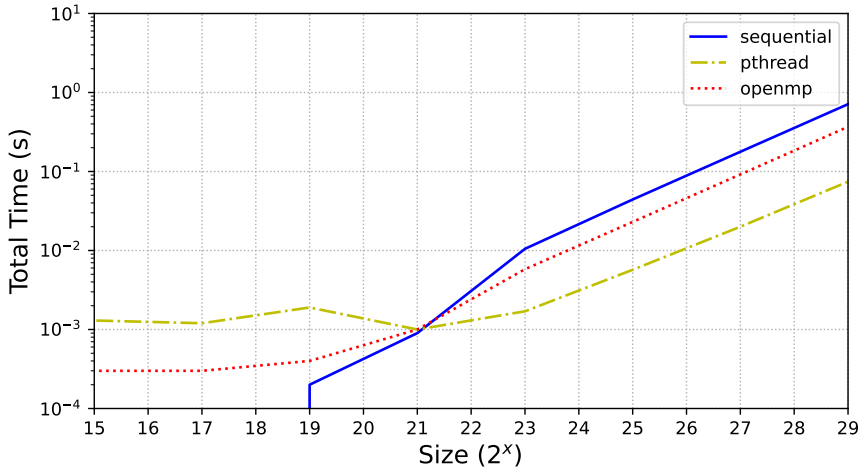


Figure 5.18: Performance of the `shift` operation on Tarbres

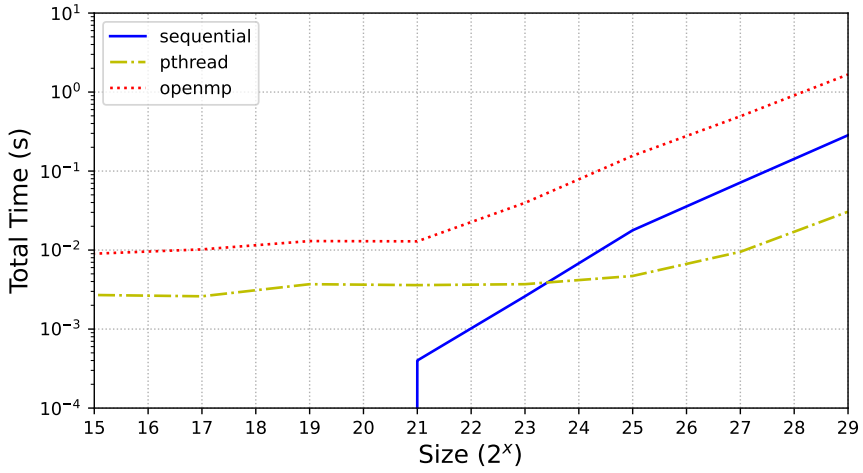


Figure 5.19: Performance of the `pull-down` operation on Tarbres

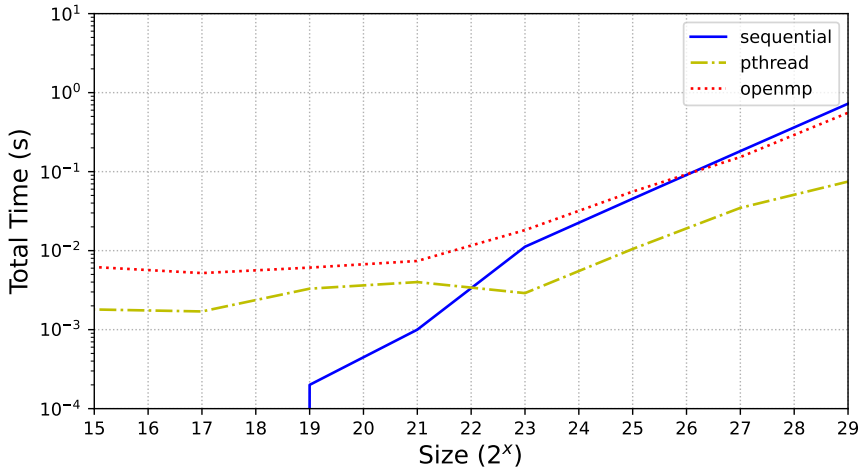


Figure 5.20: Performance of the `pull-up` operation on Tarbres

CHAPTER 6

IN-PLACE TRANSFORMATIONS ON ADTS

Research Questions

? How to generalize the approach of Tarbres on arbitrary ADTs ?

? How to adapt the polyhedral model algorithms within this setting ?

As demonstrated in the previous chapter, *structural operations* on trees (like rotations) can be optimized "in the polyhedral model fashion" if we provide both a compact representation and low-level description. In this chapter, we propose to elaborate on this new point of view and propose a *language-based* enhanced compilation for Algebraic Data Types, that reuses key ideas and algorithms from the classical polyhedral model.

6.1 Towards ADT compilation

Traditional techniques to optimize pattern matching [Mar08] relies on immutability and don't leverage any parallelism. More recent work [JK11,JK12,GJK13,SSNK19,SK19] 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.

In this chapter we propose an efficient compilation of structural pattern matching on ADTs terms. In the rest of the chapter, 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, ie for instance, in the Breadth-First Layout of the two preceding chapters.

In this context, terms are a subpart of a *support*, which spans the whole underlying array.

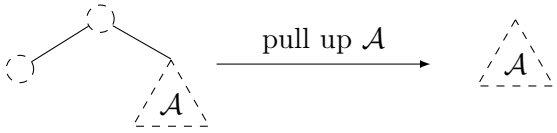
Approach We define REW, a core language to rewrite algebraic terms (Section 6.2). We first use a language-based approach to derive dependences between operations on subterms (Section 6.3.1). We then rely on the polyhedral compilation techniques to schedule these operations (section 6.3.2) and emit the appropriate sequential code (Section 6.3.3). We also developed a prototype implementation, which we used to illustrate our examples.

6.2 From Pattern Matching to Memory Moves

REW allows describing Algebraic Data Types and simple rewriting rules on them. For conciseness, we omit the type system and dynamic semantic for this simple system (see [Mar08, GPJS20] 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 29 (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 the transformation `pull-up`(Figure 5.5) 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
}
```



6.2.1 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 22 and demonstrated in Example 29.

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 [GPJS20]. 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 22 (Syntax of the rewrite language REW)**.**

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

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).

6.2.2 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, 23 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 23 (Subterm locations and movements)**.**

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

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).

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

Example 30. On the first clause in Example 29, we obtain the following moves:

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

Given a set of movements on subterms, we must decide if some movements should be done before the other. For instance, in Example 30, the moves of b and j must be executed before the move of c, as this last move will erase the location $.2/\mathit{tree}$, which originally contains b and j. For this purpose, we define the notion of *conflict* between two locations.

Definition 24 (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.

We can give a primitive notion of "must happen before" relation on moves: Given two moves $(\!| a : t \mid \ell_p \to \ell_e |\!)$ and $(\!| a' : t' \mid \ell'_p \to \ell'_e |\!)$, a must happen before a' if $\ell_p \bowtie \ell'_e$.

6.2.3 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 30. 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 6.1 by decomposing each subtree move into a collection of memory moves on *paths*. 25 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 25 (Paths and memory movements)**.**

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

In the rest, we freely use properties of regular languages on paths.

The $\mathrm{Atomize}$ 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 $\mathrm{Atomize}$ on an example.

Example 31. In Example 30, we obtained the following subterm moves for the first clause of Example 29:

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

The tree can be further decomposed into independant layers (eg. in the case of the breadth first search layout, layers are regions of the form $(2^i, 2^{i+1})$), the memory moves can thus be further broken down into generalized moves which are moves about memory regions. In the end of the application of $\mathrm{Atomize}$ we will find the following moves, where we sometimes shorten paths of the form $f.f^k$ as f^{k+1}.

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

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 Figure 6.1, 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 on whether such subterms are scalars or terms, we decompose them further into layers.

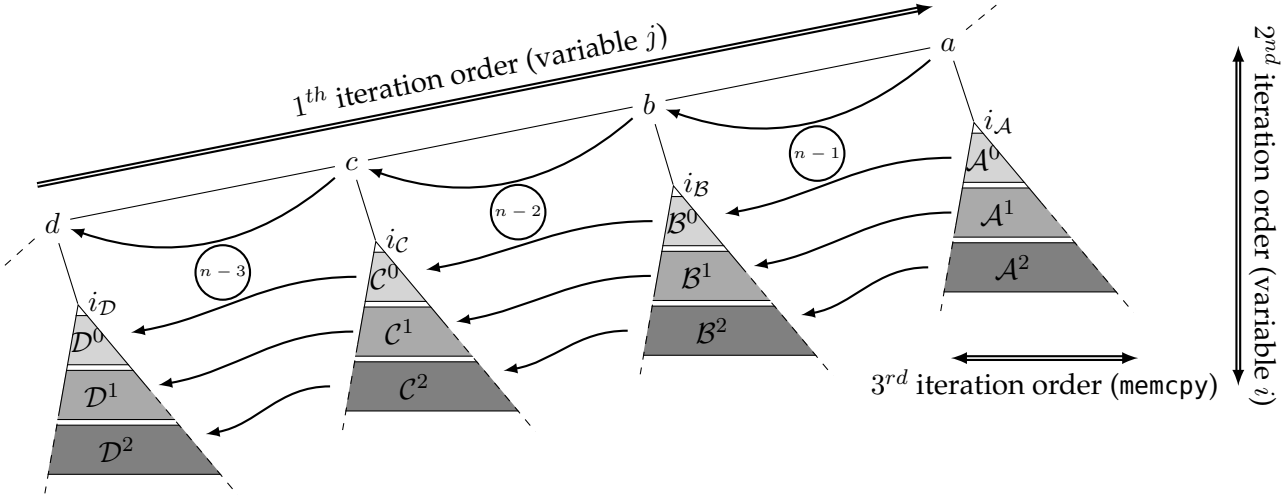


Figure 6.1: Memory movements (c), $(c0)$, $(c1)$ in Example 29.

Let us now define auxiliary functions used in Atomize.

Definition 26 (Type of locations)**.** The type of non-external locations is $\mathrm{Type}(\ell \,.\, x/T) = T$.

Definition 27 (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 2. 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 depends on whether they 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 $\ell = \mathrm{diff}(\ell_p, \ell_e)$ to direct the iteration. All the subterms not present along the iteration direction are then given by $\mathrm{Compl}(\ell)$. 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).

Algorithm 2 `Atomize`(m)

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

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 28 (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:

$$
\begin{aligned}
|\ell \,.\pi| &= |\ell| + |\pi| & |\varphi^k| &= k \\
|\ell^k \,.\pi| &= |\ell| * k + |\pi| & |\varepsilon| &= 0
\end{aligned}
$$

Since this is a linear form on $\vec{k}$, we write L_π and $\vec{l_\pi}$ such that $|\pi|(\vec{k}) = L_\pi.\vec{k} + \vec{l_\pi}$.

Definition 29 (Domain of a move)**.** We consider a move $m = (\!|\, \pi \to \pi' \,|\!)$. The domain of m is written $\mathcal{D}_m$ and defined

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

Since $|\pi|$ is a linear form on $\vec{k}$, the domain can also be defined as a polyhedron. We write D_m and $\vec{d_m}$ such that $\mathcal{D}_m = \left\{ \vec{k} \mid D_m\vec{k} + \vec{d_m} \geq \vec{0} \right\}$

6.3 Memory moves scheduling and code generation

Now that we have a fine grain characterisation of memory moves, we need to *schedule* them in order to generate code. We propose a three-steps approach, in the polyhedral model's style:

- first, we compute a compact representation for (read/write) dependences,
- from this representation, use optimization to compute logical dates compatible with these dependences.
- from this schedule, generate code.

6.3.1 Dependencies computation

A dependence happens if two moves might potentially overlap. Since the language of expression in REW is pure, the only dependences 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 dependences, 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 dependences, but also *when* this dependence happens, in term of the iteration variables k. We thus annotate this relation with constraints on the ks.

Inspired by Proposition 1, we propose the following definition:

Definition 30 ((R/W) Dependencies between moves)**.** We consider two moves $m = (\!|\, \pi_p \to \pi_e \,|\!)$ and $m' = (\!|\, \pi'_p \to \pi'_e \,|\!)$ The dependences between (the source of) m and (the destination of) m' is written $\mathcal{Q}_{(m,m')}$ 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:

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

Computing $\mathcal{Q}_{(m,m')}$ 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 $\mathcal{Q}_{(m,m')}$ thanks to careful syntactic manipulations on paths.

Lemma 2. *$\mathcal{Q}_{(m,m')}$ is a union of polyhedrons and computing its finite representation is decidable.*

Proof. The proof, based on properties of regular expressions of star height one, was done by Gabriel Radanne. □

Example 32. (Domain and Dependencies) We now give the domain and dependences of some memory moves from Example 31 We recall the moves (b) and $(c0)$.

$$(\!| \,.2/tree.0/tree.\varphi^{k_1} \to .0/tree.\varphi^{k_1} \,|\!) \tag{b}$$

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

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

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

We also compute the following dependences:

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

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 descendants. We also remark that $(c0)$ has a self dependence, 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.

Remarkably, we now have into hands a similar result as we add after Array Dataflow analysis, in Section 2.5, namely, 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 "dependence constraint" $\mathcal{Q}_{m,m'}$.

6.3.2 Scheduling via constraint solving

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 [KMW67, Fea92a, Fea92b] as well as in termination proofs [CS02, ADF+09], which we already described in Section 2.6. In this section, we recall more formal definitions and adapt them to our setting.

A schedule is a function that assigns *positive logical dates* to each move computation such that all dependences are satisfied (a computation that depends on another one is done *strictly* after). This is captured in Definition 31.

Definition 31. 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:

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

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

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

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 [Sch99] (since $\mathcal{D}_m$ and $\mathcal{Q}_{m,m'}$ are polyhedra).

Lemma 3 (Constraints $\mathcal{C}$)**.** *There exists a computable affine set of constraints $\mathcal{C}$ 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 3). 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 paragraph.

Multidimensional schedules As all schedules are not of dimension 1, we use a greedy algorithm, described in 4, similar to [KMW67, Fea92b, CS02] where each component of the schedule ρ is constructed one after the other. At each loop it makes a call to $\mathrm{compute1D}(\mathcal{C}, T)$. 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 dependences are exact [ADF+09]), thus it always gives an affine schedule.

Example 33. (Schedule) From Example 32 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.

6.3.3 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.

For this purpose, we could reuse any algorithm for code generation for the polyhedral framework [QRW00, Bas04], like the ones depicted in Section 6.3.3. We chose to recall Algorithm 1 in Algorithm 5, for which we highlight the only adaptation to our setting.

Classically, the inner procedure LoopGen 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

Algorithm 3 `Compute1D`$(\mathcal{C}, T)$ where $T \subseteq \mathcal{T}$

1: `MaximizeLP`$(\sum_{t \in T} \epsilon_t$ on $\mathcal{C})$ ▷ LP instance
2: **if** $\sum_{t \in T} \epsilon_t = 0$ **then**
3: **return** None ▷ No solution
4: **else**
5: From the result, compute σ
6: $T_{rem} \leftarrow \{t \mid \epsilon_t = 0\}$ ▷ Transitions that are not increasing
7: **return** Some (σ, T_{rem})
8: **end if**

Algorithm 4 `ComputeSchedule`$(\mathcal{M}, \mathcal{T})$

$\mathcal{C} \leftarrow$ `ComputeConstraints`$(\mathcal{M}, \mathcal{T})$
$i \leftarrow 0$; $T \leftarrow \mathcal{T}$
while T is not empty **do**
  ret $\leftarrow$ `compute1D`$(\mathcal{C}, T)$;
  if ret = None **then**
    return None ▷ No affine schedule.
  else if ret = Some (σ, T_{rem}) **then**
    $\rho_i \leftarrow \sigma$ ▷ σ is the i-th component of ρ
    $T \leftarrow T_{rem}$; $i \leftarrow i + 1$
  end if
end while
return Some(ρ) ▷ There is a i-dimensional ranking

Algorithm 5 `GenerateCode ForClause` $(\mathcal{D}, \rho, d)$

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 $\{$LOOPGEN$(i + 1, \mathcal{P}') \mid \mathcal{P}' \in \overline{\mathcal{P}'}\}$
      ▷ Decompose along the inner dimensions
  end if
end procedure
$\mathcal{P}_1 \leftarrow \{\ Im(\mathcal{D}_m, \rho_m) \mid m \in \mathcal{M}\ \}$
$r \leftarrow$ LOOPGEN$(1, \mathcal{P}_1)$
Generate code from r

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 34. If we recall Example 31, we can see that there are 5 moves. The first two are telling us that the subtree `a` is going into the void and should be ignored, and the last which is about coping over the structure will also be ignored. It remains to generate code for the three last ones. On the running example, we denote by (i, j) the iteration

dimensions. The initial set $\mathcal{P}_1$ contains the three following images:

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

From these polyhedra, we can build a partition on the first dimension: first, projecting the former images on the first dimension i gives three polyhedra $P_1 = \{i = 0\}$ (associated to b) and $P_2 = P_3 = \{1 \leq i \leq N-2\}$ (associated with $c0, c1$ moves), and 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 necessary to compute memory moves for b and $c0, c1$. In the final code these polyhedra will generate two successive for loops. It now remains to generate their inner body.

The two other recursive calls to LoopGen(g)enerate inner loops inside these "P_1, P_2 loops". Let us focus on the $(c0, c1)$ part. From projections on j we obtain two polyhedra $P'_{1,2} = \{j = 0\}$ and $P'_{2,2} = \{0 \leq j \leq N-i-2\}$ that depict the iterations for $c1$ and $c0$ respectively, it remains to generate the corresponding `for` loops and their body. The final code is thus [1]

```
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:  memcopy of 2^j adjacent cells
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
```

Figure 6.2: Final code for pull-up

6.4 Implementation and preliminary results

The first two steps (Section 6.2,Section 6.3) have been implemented as a prototype by Gabriel Radanne[2] using OCaml and Z3 as SMT and linear programming solver. The tool strictly follows the formalization described in this chapter. The last step is so far performed with ISCC, a standard tool in the polyhedral community [3].

We now demonstrate this prototype on our running example originally described in Example 29.

[1] Each move is implemented as a `memcopy` of adjacent cells

[2] https://github.com/Drup/adtr/

[3] A list of tools can be found at the following URL :https://polyhedral.info/

Example 35. Here is the version of the running example accepted by our tool:

```
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
}
```

For the first rule, the tool immediately outputs all the rules according to the approach presented in Section 6.2.3:

```
(b₀:tree | .tree@0.φ^k₀ ← b:.tree@2.tree@0.φ^k₀)
(c[]₀:tree | .tree@2^(k₁ + 1) ← .tree@2^(k₁ + 2))
(c.int@1₀:int | .tree@2^(k₁ + 1).int@1 ← .tree@2^(k₁ + 2).int@1)
(c.tree@0₀:tree | .tree@2^(k₁ + 1).tree@0.φ^k₂ ← .tree@2^(k₁ + 2).tree@0.φ^k₂)
(j₀:int | .int@1 ← j:.tree@2.int@1)
```

While the syntax present minor differences (target and destination are reversed, notably), we obtain as expected the output shown in Example 31. We recognize the significant rules, expressed in term of the two iteration variables, noted k_0, k_1 and k_2 here.

The tool can present the rules, along with their dependences, as a graph, as shown in Figure 6.3. Each rule that concerns a scalar is represented by an ellipse, while the other rules are represented by a rectangle. Nodes are annotated with their domains. Rules have an arrow between them if there is a conflict leading to a dependence. The arrows are annotated with the description of the conflict as a boolean formula. This corresponds exactly to the graph $\mathcal{Q}$ defined in Section 6.3.

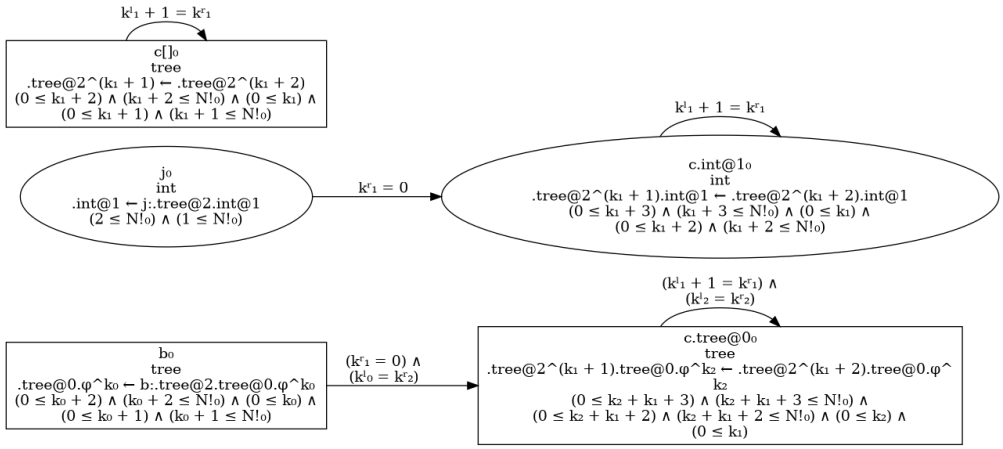


Figure 6.3: Dot output for dependences of the running example

Then, a schedule is computed with the help of Algorithm 4, and a schedule whose parameters are the iteration variables, and the size `N` is output:

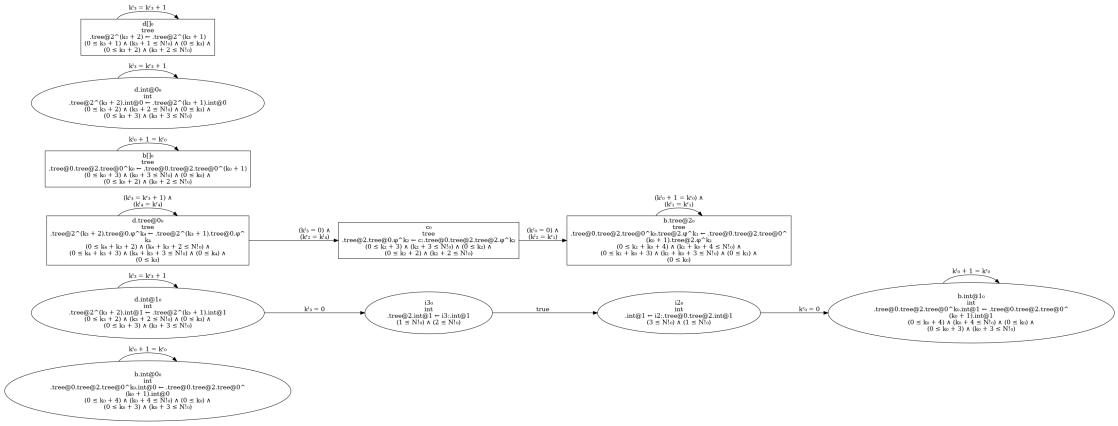
```
b0 -> (0*k0 + 0*N!0)
c.int@10 -> (k1 + 0*N!0 + 1)
c.tree@00 -> (0*k2 + k1 + 0*N!0 + 1)
c[]0 -> (k1 + 0*N!0 + 2)
j0 -> (0*N!0)
```

Our prototype can accommodate fairly complex programs, as shown in Example 36.

Example 36. Rotation in REW The following program (partially) implements rotations in an AVL:

```
rotate (t:tree) : tree = rewrite t {
  | Leaf (i) -> Leaf (i)
  | Node (Node(a,i1,Node(b,i2,c)),i3,d) -> Node (Node(a,i1,b),i2,Node(c,i3,d))
  | Node (Node(Node(a,i1,b),i2,c),i3,d) -> Node (Node(a,i1,b),i2,Node(c,i3,d))
}
```

Each rule of this program is successfully translated into a dependence graph, here is the one for the 2nd rule:



Our tool then provide a final schedule:

```
b.int@00 -> (k0 + 0*N!0 + 3)
b.int@10 -> (k0 + N!0)
b.tree@20 -> (0*k1 + k0 + N!0 + -1)
b[]0 -> (k0 + 0*N!0 + 3)
c0 -> (0*k2 + N!0 + -2)
d.int@00 -> (-1*k3 + N!0 + -2)
d.int@10 -> (-1*k3 + N!0 + -3)
d.tree@00 -> (0*k4 + -1*k3 + N!0 + -3)
d[]0 -> (-1*k3 + N!0 + -2)
i20 -> (N!0 + -1)
i30 -> (N!0 + -2)
```

The prototype was used extensively to test our technique to non-trivial programs. We can furthermore feed the output of this tool to standard polyhedral toolkits to compile a final optimized program.

6.5 Future extensions

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:

6.5.1 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 Our expression language can be easily extended to arbitrary functions on scalars such as arithmetic operations. In practice, moves become instructions of the form $\pi \leftarrow f(\pi_0, \ldots, \pi_n)$. The notion of domain and dependences immediately extend to this new context.

Functions on terms Our framework transforms a function on terms into a set of memory movements. This leads to a notion of "inlining" for function calls on subterms, since we know the prefix for both source and destinations of the nested rules. Once inlined, the rules behaved as currently, and all rules can be scheduled and compiled together.

Self-recursion The previous remark on inlining gives a way to handle some limited form of self-recursion. Indeed, we can inline the function itself at the position of the recursive call. More concretely, given the type of tree in Example 29 and a clause of the form `Node(a,i,b) -> Node(f a,i,f b)`, it suffices to prefix all the moves by $(\texttt{.}0/tree \mid \texttt{.}2/tree)^k$. This prefix spans all the subterms on which f is called recursively. This requires adding alternatives inside paths. We believe our techniques to compute domains and dependences still hold up with this extension. While this is not a fully general recursion scheme, it applies to functions such as `map` or transformations such as constant folding.

6.5.2 Richer pattern language: guards

Guards, i.e., Boolean tests inside patterns, can easily be added to our framework by equipping moves with conditions governing their applicability. This would have the added benefit to allow us to compile the whole pattern matching, including the choice of which pattern to apply. The difficulty here lies in emitting code which properly factorize the tests during iterations to avoid breaking memory locality.

6.5.3 Better code generation, evaluation

For the moment, the implementation only uses a fairly naive, sequential code generation; we have not used parallel code generation. In particular, our setup is ideally designed to produce parallel and vectorized code. We can also leverage our knowledge of the memory layout to improve memory locality when iterating through subterms.

Improve code size Currently, we do not implement any technique to improve the size of the generated code unlike the papers [QRW00, Bas04] we use as our basis, which are remove dead code if any and try to unify single point polyhedra with adjacent polyhedra.

6.6 Conclusion

In this chapter we have presented a first language-based proposal for efficient Algebraic Data Types laid out as arrays. Thanks to our experience on designing and implementing efficient low-level operations on (terms as) trees (Chapter 5), we propose an adaptation of the polyhedral model sequence of tools in order to compute dependences, schedule and generate code. The main insight of the technique is the precise characterisation of memory movements from the language description REW, whose output ressembles the output of the dependence analysis of the polyhedral model. We are then able to reuse the affine scheduling computation and code emission.

CONCLUSION

The starting point of my PhD was like many others work when it comes to contributing to the polyhedral model. Nevertheless, it takes a different angle than most of the work I have seen. Indeed, most contributions to the polyhedral model revolves around improving the tools and the algorithms central to the model: efficient compilation of imperative loop-based programs. In this work, however, we tried to push the boundary of the model in new directions. More precisely, the idea which guided us all along is the fact that, even though the polyhedral model do wonders on its original domain, it has yet to handle programs which fall out of its reach. This very same idea also made us realize that despite the fact that, even though complex data structures such as trees play a major role in many algorithms, they are not first class citizen in the HPC community and compilers have to yet to optimize them as well as they optimize arrays and loops.

Contributions

In this section, I detail the two directions I explored in this thesis to optimize a broader class of programs.

Flowchart programs with watched variables One of the main problem we face when we want to extend the polyhedral model is that the hypotheses on the input programs are made on the syntax of the programs rather than on its behavior. This leads to tedious and verbose explanations when trying to prove properties of slightly complex programs. We proposed a model of input language which directly embeds the structure of the programs, and characterizes the features of polyhedral programs directly in a more semantic manner. The polyhedral model in itself remains unchanged, only the input language is different.

This input language uses flowcharts since they perfectly capture general programs with conditions and loops without any restrictions on them. On top of those flowcharts,

we added a notion of watched variables in place of the notion of iteration variables. Watched variables are more convenient than their counterpart(the "iteration variables") since they can be introduced not only by special statements (for loops in the polyhedral model) but by any statement. This new formalism allows a definition of polyhedral programs as a subclass of general flowchart programs. This subclass can then be extended by studying the effect of each hypothesis and how they can be relaxed. The mathematical formalism also paves the way to relaxing strict polyhedral constraints by abstract interpretation.

Trees and the polyhedral model The polyhedral model brings excellent support for array and affine transformation into optimizing compilers. Nevertheless, trees and more complex data structures are not well handled by the polyhedral optimization techniques. This can be explained by the fact that contrarily to arrays that are a built-in type in most programming languages, more complex types are not first class citizens and are merely types defined by the programmers. The great diversity of user-defined types and the fact that a same data structure can be represented in many ways prevents compilers from recognizing them efficiently.

Those user-defined data structures are regular enough to be expressed concisely but, due to the recursive nature of those data structures, are difficult to handle within the polyhedral model. Indeed, one of the pillar of the polyhedral model is that arrays cells can be accessed in constant time. This is not the case with recursive data structures which needs to be, at least partially, traversed to reach an element.

In this thesis, we did not try to handle all possible user-defined data structures but restricted ourself to the class of algebraic data types without sharing. These data structures already form an important sub-class of commonly seen data structures and their regularity allows building static layouts. We believe that this subclass, due to its regularity and the fact that there is no sharing is a good target for "polyhedral-like" optimizing compilers.

Our research has focused on the study of AVL trees as they already are challenging. We explored how AVL trees could be stored into arrays and to what extent optimizations of the rotation operation (here performed directly on arrays) could reuse ideas from the polyhedral model. It turned out that, contrary to preliminary expectations, even though the domain of the operations are not polyhedra a lot of the original ideas could be reused by performing transformations on the domains of the operations.

This first work on AVL trees lead to an extension, which leverages the fact that structural modifications of algebraic data types stored in arrays can be expressed as transformations close to pattern matching found in functional programming languages. This provides a clean way to express structural transformations and to generate efficient code for these transformations. It avoids writing those transformations by hand, preventing for incorrectness or a miss of optimizing opportunities such as skewing and pipelining.

Future Work and Open Problems

Flowchart programs with watched variables Our proposition follows a tradition of labeled transition systems (LTS) which are common in static analysis of programs. Its formal mathematical definition should ease future semantic-based extensions. We think that the most important opened questions is how to define "approximated polyhedral programs" that could be optimized within the same framework (the main difficulty here is code generation, we are in our opinion far from a "**fuzzy control polyhedral model**"). A more fundamental question is to formally define what it means to be "close" to a polyhedral program.

Trees and the polyhedral model We have presented our current results on applying ideas borrowed from the polyhedral model to optimize AVL trees and algebraic data types. However, there are yet many directions to explore. Until now, we have only been concerned by the breadth first layout and have yet to measure the performances of other layouts such as the depth first infix layout or the Von Em Boas layout. We also mentioned in Chapter 4 that there is no one-fit for all layout and it could be interesting to detect automatically which layout is best for a fixed data-type. There is still room for improvement when it comes to optimizing the memory movements, and we have only paved the road for further transformations such as tiling or vectorization, and lead to the complete definition of what would call the "**hyperbolic model**" (since the domains of the memory movements are hyperbolic). Vectorization in particular remains challenging: this hard task implies to carefully review the intrinsics provided by each computer processing unit. The large scale memory movements could also directly be supported by a coprocessor in the RAM (but this brings other problems since the changes in the RAM are now partially independent of the CPU). Another interesting direction is related to GPU and explore to what extent the operations we presented could be transposed to GPU which would make algebraic data types available for GPU programming.

BIBLIOGRAPHY

[ABD07] Christophe Alias, Fabrice Baray, and Alain Darte. Bee+cl@k: An implementation of lattice-based array contraction in the source-to-source translator rose. In *Proceedings of the 2007 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '07, page 73–82, New York, NY, USA, 2007. Association for Computing Machinery. 28

[ADF+09] Christophe Alias, Alain Darte, Paul Feautrier, Laure Gonnord, and Clément Quinson. Program Termination and Worst Time Complexity with Multi-Dimensional Affine Ranking Functions. Research report, 2009. 78, 79

[AVL62] G. M. Adel'son-Vel'skii and E. M. Landis. An algorithm for organization of information. *Dokladi Akademia Nauk SSSR*, 146(2):263–266, April 1962. 30, 31, 34

[BAC16] Uday Bondhugula, Aravind Acharya, and Albert Cohen. The pluto+ algorithm: A practical approach for parallelization and locality optimization of affine loop nests. *ACM Trans. Program. Lang. Syst.*, 38(3), April 2016. 23

[Ban04] Utpal Kumar Banerjee. Data dependence in ordinary programs. Master's thesis, 2004. 3

[Bar98] Denis Barthou. *Array Dataflow Analysis in Presence of Non-affine Constraints*. PhD thesis, Université de Versailles St-Quentin, Versailles, February 1998. 14, 18

[Bas04] C. Bastoul. 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.*, pages 7–16, Oct 2004. 23, 79, 85

[BBC16a] Somashekaracharya G. Bhaskaracharya, Uday Bondhugula, and Albert Cohen. Automatic storage optimization for arrays. *ACM Trans. Program. Lang. Syst.*, 38(3), apr 2016. 28

[BBC16b] Somashekaracharya G. Bhaskaracharya, Uday Bondhugula, and Albert Cohen. Automatic storage optimization for arrays. *ACM Trans. Program. Lang. Syst.*, 38(3), April 2016. 28

[BBC16c] Somashekaracharya G Bhaskaracharya, Uday Bondhugula, and Albert Cohen. Smo: An integrated approach to intra-array and inter-array storage optimization. In *POPL 2016 - ACM Symposium on Principles of Programming Languages*, pages 526–538, Saint Petersburg, United States, January 2016. 28

[BBK+08] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *International Conference on Compiler Construction (ETAPS CC)*, April 2008. 60

[BCF97] Denis Barthou, Jean-François Collard, and Paul Feautrier. Fuzzy array dataflow analysis. *Journal of Parallel and Distributed Computing*, 40(2):210 – 226, 1997. 6, 15, 18

[BDF00] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious b-trees. In *41st Annual Symposium on Foundations of Computer Science, FOCS 2000, 12-14 November 2000, Redondo Beach, California, USA*, pages 399–409. IEEE Computer Society, 2000. 34

[Ber66] A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, EC-15(5):757–763, 1966. 3

[BF98] Pierre Boulet and Paul Feautrier. Scanning polyhedra without do-loops. 11 1998. 23

[BFJ02] Gerth Stølting Brodal, Rolf Fagerberg, and Riko Jacob. Cache oblivious search trees via binary trees of small height. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA'02, pages 39–48, USA, 2002. Society for Industrial and Applied Mathematics. 34, 47, 61

[BFS16] Guy E. Blelloch, Daniel Ferizovic, and Yihan Sun. Just join for parallel ordered sets. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, page 253–264, New York, NY, USA, 2016. Association for Computing Machinery. 32

[BHRS08] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language De-*

sign and Implementation, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM. 60

[BPCB10] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC'10/ETAPS'10, pages 283–303, Berlin, Heidelberg, 2010. Springer-Verlag. 6

[CBF95] Jean-François Collard, Denis Barthou, and Paul Feautrier. Fuzzy array dataflow analysis. *SIGPLAN Not.*, 30(8):92–101, August 1995. 15

[Coh99] Albert Cohen. *Program Analysis and Transformation: From the Polytope Model to Formal Languages*. Theses, Université de Versailles-Saint Quentin en Yvelines, December 1999. 32

[CS02] Michael A. Colón and Henny B. Sipma. Practical methods for proving program termination. In *14th International Conference on Computer Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 442–454. Springer Verlag, January 2002. 78, 79

[Fea91] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, Feb 1991. 5, 14, 18

[Fea92a] Paul Feautrier. Some efficient solutions to the affine scheduling problem, I, one-dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, October 1992. 20, 78

[Fea92b] Paul Feautrier. Some efficient solutions to the affine scheduling problem, II, multi-dimensional time. *International Journal of Parallel Programming*, 21(6):389–420, December 1992. 78, 79

[Fea98] Paul Feautrier. A parallelization framework for recursive tree programs. In David Pritchard and Jeff Reeve, editors, *Euro-Par'98 Parallel Processing*, pages 470–479, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg. 32

[FLPR99] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science*, pages 285–297, 1999. 34

[FV20] Paolo Ferragina and Giorgio Vinciguerra. The PGM-Index: A fully-dynamic compressed learned index with provable worst-case bounds. *Proc. VLDB Endow.*, 13(8):1162–1175, April 2020. 34

[GBY91] Gaston H. Gonnet and Ricardo Baeza-Yates. *Handbook of Algorithms and Data Structures in Pascal and C*. Addison-Wesley Pub (Sd), 2nd edition, 1991. 30, 31, 34

[GGL12] Tobias Grosser, Armin Größlinger, and Christian Lengauer. Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22, 2012. 6

[GJK13] Michael Goldfarb, Youngjoon Jo, and Milind Kulkarni. General transformations for gpu execution of tree traversals. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, New York, NY, USA, 2013. Association for Computing Machinery. 32, 69

[GPJS20] Sebastian Graf, Simon Peyton Jones, and Ryan G. Scott. Lower your guards: A compositional pattern-match coverage checker. *Proc. ACM Program. Lang.*, 4(ICFP), August 2020. 70, 71

[GZA+11] Tobias Grosser, Hongbin Zheng, Ragesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. Polly – polyhedral optimization in llvm. In *First International Workshop on Polyhedral Compilation Techniques, IMPACT 2011, in conjunction with CGO 2011*, Chamonix, France, April 2011. Christophe Alias, Cédric Bastoul. 5

[HLSK17] N. Hegde, J. Liu, K. Sundararajah, and M. Kulkarni. Treelogy: A benchmark suite for tree traversals. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 227–238, April 2017. 32

[JK11] Youngjoon Jo and Milind Kulkarni. 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, page 463–482, New York, NY, USA, 2011. Association for Computing Machinery. 32, 69

[JK12] Youngjoon Jo and Milind Kulkarni. 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, page 355–374, New York, NY, USA, 2012. Association for Computing Machinery. 32, 69

[KMW67] R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, July 1967. 78, 79

[KRV+21] Chaitanya Koparkar, Mike Rainey, Michael Vollmer, Milind Kulkarni, and Ryan R. Newton. Efficient tree-traversals: Reconciling parallelism and dense data representations. *Proc. ACM Program. Lang.*, (ICFP), 2021. 32

[Laf10] Eric Laforest. Ece 1754 survey of loop transformation techniques, 2010. 58

[LDF+21] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system release 4.12, Documentation and user's manual*. Projet Gallium, INRIA, April 2021. 63

[Mar08] Luc Maranget. Compiling pattern matching to good decision trees. In *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*, ML '08, page 35–46, New York, NY, USA, 2008. Association for Computing Machinery. 69, 70

[MYC+19] Mahdi Soltan Mohammadi, Tomofumi Yuki, Kazem Cheshmi, Eddie C. Davis, Mary Hall, Maryam Mehri Dehnavi, Payal Nandy, Catherine Olschanowsky, Anand Venkat, and Michelle Mills Strout. Sparse computation data dependence simplification for efficient compiler-generated inspectors. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 594–609, New York, NY, USA, 2019. Association for Computing Machinery. 28

[Pad11] David A. Padua, editor. *Encyclopedia of Parallel Computing*. Springer, 2011. 3

[Pis84] Sergio Pissanetzky. *Sparse matrix technology*. AP, 1984. 28

[Pro99] Harald Prokop. *Cache-Oblivious Algorithms*. PhD thesis, MASSACHUSETTS INSTITUTE OF TECHNOLOGY, 1999. 44

[Pug91] William Pugh. The omega test: A fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 4–13, New York, NY, USA, 1991. ACM. 5

[PW96] William Pugh and David Wonnacott. *Non-Linear Array Dependence Analysis*. 1996. 6

[QR00] Fabien Quilleré and Sanjay Rajopadhye. Optimizing memory usage in the polyhedral model. *ACM Trans. Program. Lang. Syst.*, 22(5):773–815, sep 2000. 28

[QRW00] Fabien Quilleré, Sanjay Rajopadhye, and Doran Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28:469–498, 10 2000. 23, 79, 85

[RGK11] S. Rajopadhye, S. Gupta, and D. . Kim. Alphabets: An extended polyhedral equational language. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 656–664, May 2011. 6

[SB19] Yihan Sun and Guy Blelloch. Implementing parallel and concurrent tree structures. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, page 447–450, New York, NY, USA, 2019. Association for Computing Machinery. 32

[Sch99] Alexander Schrijver. *Theory of Linear and Integer Programming*. Wiley, 1999. 78

[SFB18] Yihan Sun, Daniel Ferizovic, and Guy E. Belloch. Pam: Parallel augmented maps. *SIGPLAN Not.*, 53(1):290–304, February 2018. 32

[SK19] Kirshanthan Sundararajah and Milind Kulkarni. Composable, sound transformations of nested recursion and loops. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 902–917, New York, NY, USA, 2019. Association for Computing Machinery. 32, 69

[SRC15] Aravind Sukumaran-Rajam and Philippe Clauss. The polyhedral model of nonlinear loops. *ACM Transactions on Architecure and Code Optimization*, 12(4):48:1–48:27, December 2015. 6

[SSNK19] Laith Sakka, Kirshanthan Sundararajah, Ryan R. Newton, and Milind Kulkarni. Sound, fine-grained traversal fusion for heterogeneous trees - extended version. *CoRR*, abs/1904.07061, 2019. 32, 69

[SV13] Todor Stefanov Sven Verdoolaege, Hristo Nikolov. On demand parametric array dataflow analysis. In *Third International Workshop on Polyhedral Compilation Techniques, IMPACT 2013, in conjunction with HiPEAC 2013*, Berlin, Germany, January 2013. 6

[TCE+10] Konrad Trifunovic, Albert Cohen, David Edelsohn, Feng Li, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjödin, and Ramakrishna Upadrasta. Graphite two years after: First lessons learned from real-world polyhedral compilation. In *GCC Research Opportunities Workshop (GROW'10)*, Pisa, Italy, January 2010. 5, 6

[Ver10] Sven Verdoolaege. isl: An integer set library for the polyhedral model. In Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama, editors, *Mathematical Software – ICMS 2010*, pages 299–302, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. 5, 23

[VGGC14] Sven Verdoolaege, Serge Guelton, Tobias Grosser, and Albert Cohen. Schedule trees. In *Fourth International Workshop on Polyhedral Compilation Techniques, IMPACT 2014, in conjunction with HiPEAC 2014*, Vienna, Austria, January 2014. 20

[WR96] Doran Wilde and Sanjay Rajopadhye. Memory reuse analysis in the polyhedral model. volume 7, pages 203–215, 10 1996. 28