FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Effiziente Berechnung von optimalen Representationen für abgeleitete Datentypen in MPI

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Martin Kalany BSc
Matrikelnummer 0825673

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Univ. Prof. Dr. Jesper Larsson Träff

Wien, 28. September 2015

Martin Kalany

Jesper Larsson Träff

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Efficient Construction of Provably Optimal MPI Datatype Representations

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Martin Kalany BSc
Registration Number 0825673

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ. Prof. Dr. Jesper Larsson Träff

Vienna, 28th September, 2015

Martin Kalany

Jesper Larsson Träff

Erklärung zur Verfassung der Arbeit

Martin Kalany BSc
Neustiftgasse 83 1/6/7
1070 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 28. September 2015

Martin Kalany

Acknowledgements

First and foremost, I want to thank Jesper Larsson Träff for being a highly supportive advisor for this master's thesis as well as an invaluable academic mentor throughout my academic studies. I have learned so much from you!

My eternal gratitude goes to my friends and colleagues Felix Lübbe and Antoine Rougier for all the joyful and stimulating coffee breaks we shared. Antoine is furthermore thanked for providing countless hints on how to improve this master's thesis. My colleagues at the Research Group Parallel Computing, Alexandra Carpen-Amarie, Sascha Hunold, Christine Kamper, Markus Levonyak, Angelos Papatriantafyllou, Margret Steinbuch, Francesco Versaci and Martin Wimmer are thanked for their continuous support. You have made my stay a great pleasure!

My friends Moritz, Christina, Jürgen, Carina, Simon, Thomas, Benjamin, Katherina, Jacopo and Pavel are thanked for keeping me sane and all the fun we had during the last years.

Finally, I wish to express my deepest gratitude to my family and in particular my brother Roland, for their continuous support and for always believing in me.

Abstract

Derived datatypes are an integral part of the *Message Passing Interface* (MPI), the de-facto standard for programming massively parallel, high performance applications. The mechanism is essential for efficient and portable implementations of programs working with *complex data layouts*. MPI defines several *type constructors* that are capable of describing arbitrarily complex, heterogeneous and non-contiguous data layouts. The type constructors may be applied recursively, leading to tree-like *representations* of data layouts, also called *type trees*. Typically, multiple different representations of a given data layout exist. MPI implementations require *concise* representations to process derived datatypes efficiently. It is not easy to see for users which representation is the best for a given data layout. Many non-obvious factors need to be considered, such as machine specific hardware capabilities and optimizations.

The problem of computing the most concise (or optimal) type tree representation for a given data layout was coined the Type Reconstruction Problem. It has been an open question whether this problem can be solved to optimality in polynomial time (w.r.t. to the size of the represented data layout). So far, heuristics have been used to improve a given representation, without providing any guarantees about the quality of the result.

In this master's thesis, we present an algorithm that solves the Type Reconstruction Problem for arbitrarily complex data layouts in $O(n^4)$ time and $O(n^2)$ space, where n is the size of the data layout. A recent work showed that optimal representations can be computed in $O(n\sqrt{n})$ time if the set of considered type constructors is restricted to those with only one sub-type, i.e., if the type trees are restricted to *type paths*. For this special case, called the Type Path Reconstruction Problem, we improve the currently best known asymptotic bound significantly. Our approach requires $O(n \log n / \log \log n)$ time in the worst case and can furthermore be extended to handle type constructors not considered in previous work.

For both problems, we present detailed algorithms and proofs for the claimed time and space bounds. A proof-of-concept implementation exists, but an integration of our approach into an existing MPI library implementation is not within the scope of this master's thesis. Our result for the Type Path Reconstruction Problem was published at this year's EuroMPI conference [KT15] and our approach for the Type Reconstruction Problem is currently being prepared for a submission to IPDPS [GKST] (a preliminary version as available, see [GKST15]).

Kurzfassung

Abgeleitete Datentypen sind ein integraler Bestandteil des *Message Passing Interface* (MPI), dem de-facto Standard zur Programmierung massiv paralleler Anwendungen. Der Mechanismus ist essentiell für die effiziente Implementierung von Programmen, die mit *komplexen Datenlayouts* arbeiten. MPI definiert zahlreiche *Typkonstruktoren*, welche beliebig komplexe, heterogene und nicht zusammenhängende Datenlayouts beschreiben können. Durch rekursive Anwendung dieser Typkonstruktoren entstehen baumartige Repräsentationen von Datenlayouts, genannt *Typbäume*. Oft gibt es mehrere verschiedene Typbäume die ein gegebenes Datenlayout repräsentieren. MPI-Implementierungen benötigen *bündige* Repräsentationen um abgeleitete Datentypen effizient verarbeiten zu können. Für den Nutzer ist es nicht einfach zu sehen, welche Typbaum-Representation die beste für ein gegebenes Datenlayout ist, da zahlreiche nicht-triviale Faktoren berücksichtigt werden müssen.

Das Problem, die bündigste (oder optimale) Repräsentation für ein gegebenes Datenlayout zu finden wird als Typ-Rekonstruktionsproblem bezeichnet. Es war bis dato offen, ob das Problem in polynomieller Zeit (in Bezug auf die Größe der Eingabeinstanz) gelöst werden kann. Bisher wurden Heuristiken verwendet, um gegebene Typbäume zu verbessern. Allerdings können dabei keine Garantien über die Qualität der Lösungen gegeben werden.

Wir präsentieren in dieser Diplomarbeit einen Algorithmus, der das Typ-Rekonstruktionsproblem für beliebig komplexe Datenlayouts in polynomieller Zeit löst. Für eine Eingabeinstanz der Größe n benötigt der Algorithmus $O(n^4)$ Schritte und $O(n^2)$ Speicher. Wie vor kurzem gezeigt wurde, können optimale Repräsentation effizient berechnet werden, wenn die berücksichtigten Typkonstruktoren auf jene mit nur einem Untertyp eingeschränkt werden. Die Typbäume sind in diesem Fall Pfade und das Problem kann in $O(n\sqrt{n})$ Schritten gelöst werden [Trä14]. Wir verbessern die asymptotische obere Schranke für diesen Spezialfall, genannt Typpfad-Rekonstruktionsproblem. Unser Algorithmus benötigt $O(n \log n / \log \log n)$ Schritte und kann auch zusätzliche Typkonstruktoren berücksichtigen, die bisher keine Berücksichtigung fanden.

Wir präsentieren für beide Probleme detailierte Algorithmen und Beweise. Unser Ergebnis für das Typpfad-Rekonstruktionsproblem wurde auf der diesjährigen EuroMPI-Konferenz veröffentlicht [KT15]. Das Ergebnis für das Typ-Rekonstruktionsproblem wird derzeit für eine Veröffentlichung bei der Konferenz IPDPS vorbereitet [GKST] (eine vorläufige Version ist verfügbar, siehe [GKST15]).

Contents

CHAPTER 1

Introduction

This thesis is mainly concerned with the efficient construction of *provably optimal* representations for complex datatypes occurring in the *Message Passing Interface* (MPI) [MPI15], the de-facto standard interface for programming massively parallel, high performance applications. MPI's *derived datatypes* are a powerful mechanism for describing non-contiguous, heterogeneous data layouts. The concept is orthogonal to MPI's communication operations: Complex, recursively defined datatypes may be used in all communication operations defined by the MPI standard, including basic send-receive operations as well as collective communication, remote memory access and parallel file I/O operations. The usage of derived datatypes allows for possibly more efficient implementations of parallel programs and additionally increases readability and portability of applications built with MPI.

MPI provides several *type constructors* that are capable of describing arbitrarily complex, non-contiguous and heterogeneous data layouts. Typically, a complex data layout may be represented by different combinations of these type constructors. The structural information of data encoded in such a representation enables MPI to efficiently process the data, e.g., by exploiting advanced communication hardware features. A *concise* representation is required to enable MPI libraries to fully exploit the potential performance benefits. The problem of finding the most concise (or *optimal*) representation for a given data layout was termed the Type Reconstruction Problem, the investigation of which is the core of this master's thesis. The closely related Type Normalization Problem is to transform a given representation into an optimal one. A concise representation ideally takes into account several far from trivial, machine dependent opportunities for performance optimizations. It is therefore not at all obvious for MPI users how to best construct a representation for a complex datatype. Although both problems were recently conjectured to be NP-hard, we are able to present in this work polynomial-time algorithms. However, for them to be practical, near-linear time complexity is required, which we unfortunately could not achieve (yet). Additionally, we improve on the best known

asymptotic time bound for a special case of the Type Reconstruction Problem that often occurs in practice.

Our result for the Type Path Reconstruction Problem was published in [KT15]. The result for general type tree was submitted to IPDPS [GKST] and a preliminary version is available [GKST15].

This introductory chapter provides a brief overview of MPI (Section 1.1) and in particular its derived datatypes mechanism (Section 1.2). The aim of this part is to provide the reader with sufficient knowledge about MPI to motivate and put into context the Type Reconstruction Problem. It is by no means a tutorial of MPI and covers only aspects relevant in the context of this work. A good introduction to MPI is given by Gropp et al. [GLS99b] and the full definition of all MPI operations can be found in the MPI standard [MPI15]. The Type Reconstruction Problem is introduced informally and motivated in Section 1.3. To state the problem formally, a more precise and detailed coverage of MPI's derived datatypes is necessary, as is the introduction of a formal, well-defined model. We therefore defer the formal problem statement until Chapter 2 and focus on an intuitive understanding of the problem and its context in this necessarily brief chapter. This chapter continues with a detailed statement of this master's thesis' aim (Section 1.5) and a review of the applied methodology (Section 1.6). It concludes with an overview of the subsequent chapters in Section 1.7.

1.1 The Message Passing Interface

The *Message Passing Interface* (MPI) [MPI15] is a widely used standard for programming parallel, distributed memory computer systems. It is particularly relevant in the area of high performance computing [Trä09]. It's goal is to establish *"a practical, portable, efficient, and flexible standard for message passing"* [MPI15, p. 1].

The *message passing paradigm* models computation as a collection of *processes* communicating with *messages*. Each process executes a program, which may differ from the programs executed by other processes. A process can perform operations only on data residing in local memory and processes are not assumed to be synchronized. Using Flynn's taxonomy of computer architectures [Fly72], this model of computation can be classified as *Multiple Instruction, Multiple Data Streams* (MIMD).

Gropp et al. define communication in the context of message passing systems as *"a portion of one process's address space is copied into another process's address space"* [GLS99b, p. 14]. The basic communication mechanism defined by MPI is point-to-point communication where one process sends data (a *message*) to another, receiving process. The `MPI_Send` operation sends data to a specific process, which may receive it with a matching `MPI_Recv` operation.

The MPI *collective communication* operations define advanced communication patterns that involve a group of processes: *Data movement* operations rearrange data among the participating processes, whereas *collective computation* operations combine partial results of computations carried out by different processes. Probably the simplest data movement operation is `MPI_Bcast`, where an identical message is sent from one process

(the root process) to each of the other processes participating in the operation. To give another example, `MPI_Scatter` splits the data of the root process and distributes it to the other participating processes. `MPI_Reduce` is a collective operation that combines data of each process according to some operation. Predefined operations include `MPI_MAX`, which finds the maximum value, and `MPI_SUM`, which computes the sum of all elements.

MPI furthermore defines advanced operations such as parallel file I/O, remote memory access, capabilities to query specifics of the execution environment and many more features useful for implementing massively parallel, high performance applications.

In MPI, a message consists of `count` elements of a given `datatype`, which are stored successively in memory, starting at address `buf`. The memory layout described by this triple is also called a *communication buffer*. MPI's communication operations exchange *typed values*, not just uninterpreted bytes. This alleviates the application programmer from tedious, machine specific details such as the binary representations of values in different environments (e.g., IEEE 754 [IEE08] versus hexadecimal floating point encoding used in IBM systems [zar]).

The MPI standard defines *basic datatypes* (or base types) with a one-to-one correspondence to predefined datatypes in C or Fortran. For example, the C types `char`, `int`, `float` and `double` correspond to `MPI_CHAR`, `MPI_INT`, `MPI_FLOAT` and `MPI_DOUBLE` and analogous MPI basic datatypes exist for Fortran datatypes [MPI15, p. 25-26]. The facilities described so far limit messages to be sequences of elements of the same base type, but applications often require non-contiguous or heterogeneous data to be transfered (e.g., a column of a matrix stored in row-major order; or an integer count followed by an array of double values). MPI's *derived datatype* functionality can represent arbitrary data layouts and allows to directly transfer (that is, without explicitly copying) such non-contiguous and heterogeneous data. This mechanism is orthogonal to other features of MPI and may be used in all of MPI's communication operations, including collective communication and parallel file I/O as well as remote memory access operations. Derived datatypes are the central concern of this work and a broad introduction is given in the next section. A detailed and precise presentation, covering all aspects relevant for this work in depth, is given in Sections 2.1 to 2.3.

It is important to note that MPI is a *library interface specification*, not a programming language or a library implementation. As such, it provides a clearly defined set of portable and (relatively) easy to use operations useful for implementing efficient parallel applications based on the message passing paradigm. The MPI standard does not state any implementation or performance requirements but leaves it to hardware vendors and other organizations to provide high-quality implementations. This approach allows for portable parallel applications that may nevertheless take advantage of vendor specific, specialized hardware. All operations are expressed as functions (or subroutines), with language bindings specified for C and Fortran (the C++ bindings included in previous versions of the MPI standard are deprecated in the currently most relevant versions 2.2 and 3.1). Throughout this work, we use the language agnostic syntax with which the MPI standard defines its operations, except for code listings where the corresponding

$$\begin{bmatrix} 1.1 & 1.2 & 1.3 & 1.4 & 1.5 \\ 2.1 & 2.2 & 2.3 & 2.4 & 2.5 \\ 3.1 & 3.2 & 3.3 & 3.4 & 3.5 \\ 4.1 & 4.2 & 4.3 & 4.4 & 4.5 \end{bmatrix}$$

Figure 1.1: An example matrix of 4x5 floating point (`double`) values.

C bindings are used. For predefined language specific handles like basic datatypes, C bindings are used.

1.2 Derived datatypes

As outlined in the previous section, arbitrary, non-consecutive and/or heterogeneous data layouts can be represented with derived datatypes. A data layout is a sequence of base types together with their displacements in memory. Displacements are taken relative to the start address of a communication buffer and are typically measured in bytes. In this section, we proceed by example to give an intuitive, if rather informal, introduction to this powerful mechanism as far as it is necessary for a basic understanding of the Type Reconstruction and Normalization Problems and their applications.

In the area of high performance computing, computations are often performed on large matrices and parts of matrices (e.g., a row or a column) need to be communicated between processes. Assume that a 4x5 matrix of floating point values as shown in Figure 1.1 is stored in memory in row-major order, i.e., as a linear, contiguous array

$$\begin{bmatrix} 1.1 & 1.2 & 1.3 & 1.4 & 1.5 & 2.1 & 2.2 & 2.3 & 2.4 & 2.5 & 3.1 & 3.2 & 3.3 & \dots \end{bmatrix} \quad .$$

As mentioned above, a message in MPI is defined by a triple (`buf`, `count`, `datatype`). A row of the matrix can easily be communicated with one message by setting `buf` to the address of the row's first element, `count` to the number of elements in a row and `datatype` to the appropriate MPI datatype (`MPI_DOUBLE` in our example, where the matrix elements are of type `double`).

A column of this matrix on the other hand is not stored contiguously in memory. To send the values of the second column, we can construct a contiguous communication buffer by copying the required elements into a separate array

$$\begin{bmatrix} 1.2 & 2.2 & 3.2 & 4.2 \end{bmatrix} \quad ,$$

which is then used to perform the communication as before. However, this *packing* of non-contiguous data into a contiguous, temporary communication buffer is tedious, error prone and may reduce performance. Alternatively, the derived datatype constructor `MPI_TYPE_VECTOR` can be used to directly describe such a column to the communication

operation:

```
MPI_TYPE_VECTOR(count, blocklength, stride,
                oldtype, OUT newtype)
```

The MPI_TYPE_VECTOR constructor describes a data layout consisting of count blocks, where each block consists of blocklength many concatenated elements of type oldtype and two consecutive blocks are stride many elements apart. A type COLUMN_TYPE representing a column of our example matrix can thus be constructed by

```
MPI_TYPE_VECTOR(4, 1, 5, MPI_DOUBLE, COLUMN_TYPE) .
```

COLUMN_TYPE describes a derived datatype of four elements of base type MPI_DOUBLE with a "gap" of five elements between each. In other words, if the matrix is stored in row-major order as above, COLUMN_TYPE selects every fifth element. To send the second column of the matrix, one has to set buf to the address of the element in the first row and second column, the count to one and the datatype to COLUMN_TYPE. The message specified by the triple (buf, 1, COLUMN_TYPE) consists of one element of type COLUMN_TYPE, which is stored in a memory area starting at address buf.

Note the difference in the way a row of the matrix was communicated, where several elements of the base type were specified as the message. Alternatively, one can define a ROW_TYPE to communicate a row analogously to a column. The constructor

```
MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)
```

replicates a datatype count many times into a contiguous buffer, analogously to an array. A datatype for a row of the example matrix can be defined as

```
MPI_TYPE_CONTIGUOUS(5, MPI_DOUBLE, ROW_TYPE) .
```

This constructor can be seen as a less general version of MPI_TYPE_VECTOR, since a call to MPI_TYPE_CONTIGUOUS(count,oldtype,newtype) is equivalent to

```
MPI_TYPE_VECTOR(count, 1, 1, oldtype, newtype) .
```

Listing 1.1 provides a concrete and full implementation of this example problem. To compile and execute the programs listed in this section, a reasonably recent MPI library implementation, adhering to version 3.0 of the standard, is required. Several open source implementations exist, e.g., MPICH[1] and OpenMPI[2]. To compile and execute the example program with two processes, use

```
mpicc -o example example.c
mpirun -np 2 example
```

[1] http://www.mpich.org/
[2] http://www.open-mpi.org/

```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);

    int i, rank, tag = 1;
    MPI_Status status;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // A 4x5 matrix of double values
    double matrix[4][5] = { {1.1, 1.2, 1.3, 1.4, 1.5},
                            {2.1, 2.2, 2.3, 2.4, 2.5},
                            {3.1, 3.2, 3.3, 3.4, 3.5},
                            {4.1, 4.2, 4.3, 4.4, 4.5} };

    // Non-contiguous data:
    // Derived datatype for a column of a 4x5 matrix with double values
    MPI_Datatype COLUMN_TYPE;
    MPI_Type_vector(4, 1, 5, MPI_DOUBLE, &COLUMN_TYPE);
    MPI_Type_commit(&COLUMN_TYPE);

    if(rank == 0) {
        // Send the second column of "matrix" to rank 1
        printf("Rank %d: Sending 2nd column to rank 1\n", rank);
        MPI_Send(&matrix[0][1], 1, COLUMN_TYPE, 1, tag, MPI_COMM_WORLD);
    }
    if(rank == 1) {
        // Receive column data from rank 0 into contiguous buffer
        double array[4];
        MPI_Recv(array, 4, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD, &status);
        printf("Rank %d: received:\n", rank);
        for(i = 0; i < 4; i++)
            printf("\t%.1f\n", array[i]);

    }
    MPI_Type_free(&COLUMN_TYPE);
    MPI_Finalize();
}
```

Listing 1.1: Using a derived datatype to transmit a column of a matrix.

In the following, we give only an intuitive understanding of the MPI functions used in the examples. We refer the reader to the excellent text book by Gropp et al. [GLS99b] and the MPI standard [MPI15] for an in-depth coverage.

Every MPI program has to initialize the environment with a call to `MPI_Init`, before calls to other MPI operations are permitted. This call allows to pass several parameters to MPI, e.g., the number of processes that shall execute the code, which is given as an argument (`-np 2`) when executing the program. To ensure proper termination, the last call to an MPI operation has to be that of `MPI_Finalize`. The code between these

two function calls is executed in parallel by all processes. To selectively execute portions of the code by a certain process only, the `rank` of the process can be used. It is queried by a call to `MPI_Comm_rank`, which returns a distinct integer in the range $[0, np[$ and uniquely identifies a process. The example program then proceeds with the definition of our example matrix and the declaration of the derived datatype `COLUMN_TYPE`. The call to `MPI_TYPE_VECTOR` defines a non-contiguous datatype that picks every fifth element, for a total of four. The elements are of type `MPI_DOUBLE`, which directly corresponds to the C-type `double`. The environment is then made aware of this derived, user-defined datatype by calling `MPI_Type_commit`, after which it can be used in communication operations. When the datatype is not required any more, a call to `MPI_Type_free` is used for deallocating the datatype. In our example, process 0 sends the second column of the matrix to process 1. The message it transmits is specified as a triple, which consists of *1*) the *start address* of the buffer containing the message, *2*) the *number of elements*, and *3*) the *datatype* of the elements. The appropriate values are given as the first three arguments in `MPI_Send`. The remaining arguments are *4*) the rank of the *target process*, *5*) a *tag* that can be used to distinguish messages, and *6*) a *communicator*, which defines a set of processes that may participate in the operation (`MPI_COMM_WORLD` is a predefined handle for the set of all processes of the current environment). Process 1 receives the message with a call to `MPI_Recv`. The first three arguments again specify the message, with *1*) the *start address* of an allocated memory buffer to store the received data, *2*) the *number of elements*, and *3*) their *datatype*. These arguments are followed by *1*) the rank of the *sending process*, *2*) the *tag*, *3*) the *communicator*, and *4*) a *status* variable used to return information about successful completion or an encountered error.

We briefly sketch two additional type constructors. `MPI_TYPE_CREATE_INDEXED_BLOCK` is slightly more general than `MPI_TYPE_VECTOR`, where the blocks do not have to be strided but can have arbitrary displacements. This constructor, instead of the scalar `stride` value, requires an array of displacements as argument. Even more general, `MPI_TYPE_INDEXED` allows blocks to contain a different number of elements. Note that all constructors introduced so far use a single `oldtype` argument. Although they can describe arbitrary non-contiguous data layouts, a combination of different base types is not possible. Only `MPI_TYPE_CREATE_STRUCT`, the most general type constructor, allows for the representation of heterogeneous data. It takes an array of `oldtypes` as an argument and can be used to specify e.g., a row of the example matrix (an array of floating point values) preceded by an integer value indicating the row number, without having to resort to manually packing the data into a buffer of non-typed data (bytes).

Listing 1.2 provides an implementation of this example problem. To get a complete MPI program, replace lines 11 to 37 in Listing 1.1 with the code given in Listing 1.2. Analogously to the previous example, the listing proceeds by first declaring the required C-types and an MPI derived datatype, which is then used in a basic send-receive operation. `MY_ARRAY_TYPE` is a heterogeneous datatype corresponding to the C structure `My_array`. Note that structures cannot directly be used in MPI operations, since portability and transparent communication between heterogeneous platforms could not be provided with such a low-level and machine specific description of message data. The construction

```
1     // Array of doubles plus element count
2     typedef struct {
3         int row;
4         double elements[4];
5     } My_array;
6
7     // Derived datatype for an array of doubles plus element count
8     MPI_Datatype MY_ARRAY_TYPE;
9     const int nr_blocks = 2;
10    int blocklengths[nr_blocks] = {1, 4};
11    MPI_Datatype oldtypes[nr_blocks] = {MPI_INT, MPI_DOUBLE};
12    MPI_Aint displacements[nr_blocks];
13    displacements[0] = offsetof(My_array, row);
14    displacements[1] = offsetof(My_array, elements);
15    MPI_Type_create_struct(nr_blocks, blocklengths, displacements,
16                           oldtypes, &MY_ARRAY_TYPE);
17    MPI_Type_commit(&MY_ARRAY_TYPE);
18
19    if(rank == 0) {
20        My_array array1  = {3, 3.1, 3.2, 3.3, 3.4};
21        // Send heterogenous data (1 int + 4 doubles) to rank 1
22        MPI_Send(&array1, 1, MY_ARRAY_TYPE, 1, tag, MPI_COMM_WORLD);
23    }
24    if(rank == 1) {
25        My_array array2;
26        // Receive data from rank 0
27        MPI_Recv(&array2, 1, MY_ARRAY_TYPE, 0, tag, MPI_COMM_WORLD, &status);
28        printf("Rank %d received elements of row %d:\n", rank, array2.row);
29        for(i = 0; i < 4; i++)
30            printf("\t%.1f\n", array2.elements[i]);
31    }
32    MPI_Type_free(&MY_ARRAY_TYPE);
```

Listing 1.2: Transmitting heterogeneous data with the help of derived datatypes.

of `MY_ARRAY_TYPE` is a bit more involved than the previous example: We want to describe a memory layout of two blocks, with the first block containing one `integer` and the second block containing four `double` values. Note that the start address of the array `elements` depends on the used alignment and automatic padding performed by the compiler. It is explicitly computed with the help of the C-function `offsetof`. Once the required arguments are assembled and the datatype is committed, it can be used in communication operations in the same way as any other datatype, including base types.

Several more examples can be found in the MPI standard itself [MPI15, p. 123–131].

It is not necessary that all processes participating in a communication operation use the same datatype. Indeed, in the example in Listing 1.2, the send operation used the derived datatype `COLUMN_TYPE` to send data stored locally in non-contiguous memory locations, which is received into a contiguous buffer (i.e., an array) of floating point values. Using different datatypes for the send and receive operations is possible, as long

as the two datatypes match the message data. In other words, receiving an `integer` value into a `double` variable is considered an invalid operation. In particular, the type signatures of both datatypes must be equal. Unfortunately, datatype equivalence is not well defined by the MPI standard [KGR10]. MPI libraries typically do not check for such errors and thus a type mismatch may cause a subtle error that is hard to track down.

MPI's type constructors can be applied recursively to describe extremely sophisticated data layouts. A combination of multiple constructors allows for concise representations of complex, non-contiguous and heterogeneous data layouts. The derived datatype mechanism is orthogonal to MPI's communication operations, meaning that an arbitrarily complex derived datatype can be used wherever a type is required. By providing information about the structure of the data, the MPI library may significantly improve performance by exploiting special communication hardware that is able to deal with structured data more efficiently than non-tuned self-devised packing code [GLS99b]. Derived datatypes may furthermore improve performance by obviating the need for internal buffering [THRZ99, MT08] and have been used successfully to increase the performance of classic parallel algorithms [TRH14] as well as numerical applications [BT11]. Höfler and Gottlieb [HG10] report a 3.8 fold speedup for parallel Fast Fourier Transformation. They incorporate a local memory transpose operation in the derived datatype, which significantly reduces the amount of local copy-operations and highlights the usefulness of derived datatypes beyond standard message passing. Lu et al. [LWPS04] report that the usage of derived datatypes may increase performance "due to the optimizations used in MPI implementations which are easily missed by users in their packing". This work as well as [SGH12] highlight that, apart from performance, the usage of derived datatypes also improves readability and *performance portability*, meaning that a program facilitating derived datatypes is able to perform well on a larger range of platforms, whereas manually written packing code is typically optimized for only a few.

1.3 Type reconstruction and normalization

As discussed in the previous section, type constructors may be applied recursively and can describe arbitrarily complex non-contiguous and heterogeneous data layouts. While any data layout can be defined by explicitly listing the base types of all elements plus their displacements, more concise specifications may be possible with the help of derived datatypes.

As an example, assume that from an array of n floating point values every other value has to be communicated to another process. This data layout can be described by explicitly listing the displacement (0,2,4,...) and datatype (`MPI_DOUBLE`) of the n over two values with the help of the `MPI_TYPE_CREATE_INDEXED_BLOCK` constructor:

```
MPI_TYPE_CREATE_INDEXED_BLOCK(n/2, 1, [0,2,4,...],
                              MPI_DOUBLE, EVERY_OTHER_ELEMENT)
```

Alternatively, the derived datatype EVERY_OTHER_ELEMENT can be defined as

```
MPI_TYPE_VECTOR(n/2, 1, 2, MPI_DOUBLE, EVERY_OTHER_ELEMENT) .
```

Note that the base type MPI_DOUBLE is *repeated* in the data layout with regular displacements. This regular structure of the data layout is stated explicitly by the MPI_TYPE_VECTOR constructor and may be exploited to implement MPI operations more efficiently (see e.g., [RMG03, SWP04]). Additionally, this *representation* of the data layout is much more concise. Only a constant number of parameters needs to be stored, whereas an explicit listing requires $O(n)$ memory.

For more complex data layouts it is typically much harder to find an efficient representation, since the type constructors may be applied recursively. Such a nested derived datatype can be viewed as a *type tree*, a tree-like structure where an inner node corresponds to the application of a type constructor and a leaf node to one of the base types occurring in the data layout. The original data layout can be obtained by an ordered traversal of such a type tree, which is detailed in Section 2.5.

It is natural to associate a *cost* with each inner node of the tree, reflecting the space consumption and processing cost of the corresponding constructor. Different type trees representing the same data layout thus may incur different costs. Space and processing cost efficient representations of data layouts are required by MPI implementations to fully exploit the potential performance advantages. The Type Reconstruction Problem asks for a most concise type tree representation for a given data layout. The closely related Type Normalization Problem asks to generate a cost-optimal type tree out of a given, user-defined type tree.

If the set of considered constructors is restricted to those with only one sub-type, the type trees degenerate to *type paths*. We denote this special case as the Type Path Reconstruction Problem, which has recently been shown to be solvable in polynomial time if only the MPI_TYPE_VECTOR and MPI_TYPE_CREATE_INDEXED_BLOCK constructors are considered [Trä14].

1.4 Applications

Applications typically require type normalization. In particular, the Type Normalization Problem is important for MPI implementations to handle derived datatypes efficiently [MT08, GHTT11]. The potential performance benefits of optimal type representations are huge with execution times reduced by up to 50% for simple communication operations (see Section 2.4 for details). A suitable point for performing datatype normalization is at commit time, i.e., in the call of MPI_Type_commit [PG15].

The problem can trivially be solved by performing type reconstruction on the type map represented by the type tree. This however requires to explicitly store the type map, which may be arbitrarily larger than its type tree representation. Apart from this straight-forward approach, only heuristic approaches (see Chapter 3) and one result for the special case of type paths [Trä14] have been presented so far. Thus, improvements to the Type Reconstruction Problem directly lead to better algorithms for the

Type Normalization Problem. Furthermore, type reconstruction is relevant for the automatic generation of derived datatypes, where a derived datatype representation is constructed for given packing code [KHS12, SKH13] or complex C datatypes [RP06].

One can also imagine a stand-alone tool for type normalization to determine optimal representations for a given execution environment. The resulting type representations can then be hard-coded in the application, thus saving the cost of type normalization that would otherwise have to be performed by the MPI library for each execution.

1.5 Aim of this work

The aim of this master's thesis is to advance research on the Type Reconstruction Problem and, where possible, to apply the new findings to the Type Normalization Problem.

The Type Reconstruction Problem was recently conjectured to be NP-hard (see [GHTT11, Trä14]). The main contribution of this work is to refute this conjecture by giving a polynomial-time algorithm for the problem. This algorithm was submitted to the IPDPS conference [GKST] and a preliminary version is available online [GKST15]. Contrary to scientific publications, which target experts and have to adhere to strict page limits, this master's thesis is written for a more general audience by providing a much more elaborate introduction to and motivation for the problem as well as a more detailed, step-by-step presentation of algorithms and proofs.

The second major contribution is an algorithm solving the Type Path Reconstruction Problem asymptotically faster than previously known approaches. Our approach is capable of integrating further type constructors not considered so far and was published in this year's EuroMPI conference [KT15]. It is likewise presented in a detailed and accessible manner.

1.6 Methodological approach

The assumed model of computation is the sequential *random access machine* (RAM) model, as used in the standard text book for algorithms by Cormen et al. [CLRS09]. As discussed in Section 1.2, processes participating in a communication operation do not have to employ the same datatype definition. Indeed, a process does not know which datatype is used by the other processes unless the used datatypes are communicated explicitly. Thus, datatype reconstruction has to be solved individually by each process. Therefore all presented algorithms as well as approaches in related work are purely sequential and no model of parallel computation has to be adopted.

We analyze the *worst-case time and space* requirements of all presented algorithms w.r.t. the size of the input. The input is a data layout consisting of n elements (pairs of base type plus displacement). We use standard big-O notation to state the asymptotic time and space complexity of algorithms. Let $f(n)$ and $g(n)$ be non-negative functions and $n \in \mathbb{N}$ and $c \in \mathbb{R}_{\geq 0}$, $n_0 \in \mathbb{N}$ be suitable constants. We use $f(n) = O(g(n))$ to express that $g(n)$ is an asymptotic upper bound for $f(b)$, which is formally defined as

$$f(n) = O(g(n)) \iff \exists c, n_0 \, \forall n \geq n_0 : f(n) \leq cg(n)$$
$$\iff \limsup_{n\to\infty} \frac{f(n)}{g(n)} < \infty \quad .$$

Likewise, we use $\Omega(n)$ and $\Theta(n)$ to denote lower and tight bounds.

$$f(n) = \Omega(g(n)) \iff \exists c, n_0 \, \forall n \geq n_0 : cg(n) \leq f(n)$$
$$\iff \liminf_{n\to\infty} \frac{f(n)}{g(n)} > 0$$

$$f(n) = \Theta(g(n)) \iff f(n) = \Omega(n) \wedge f(n) = O(n)$$

A simple, yet flexible cost model is used to determine the cost of a type tree representation. A type tree is *optimal*, if it is of minimal cost, i.e., there is no other type tree of less cost representing the same type map. This cost model is formally introduced in Chapter 2, where also the precise problem statement is given.

Algorithms are mostly presented in high-level pseudo-code. Proof-of-concept implementations of the developed algorithms have been written in C (ask the author for the sources). Implementation details are discussed in this work only insofar as they are necessary for proving an algorithm's correctness or space and time bounds. This, however, should suffice for a capable programmer to implement the algorithms. An integration into an existing MPI library implementation is not within the scope of this master's thesis.

1.7 Structure of this work

Chapter 2 states formally and much more precisely all of the problems introduced so far. To do so, a closer look at MPI's derived datatypes mechanism, a formal model for type trees as well as a cost model are required. We introduce several additional interesting problem variants at the end of that chapter. An overview of previous research on the Type Reconstruction Problem and related work is given in Chapter 3. Chapter 4 investigates properties of optimal type trees. These properties are crucial for the developed algorithms for the Type Reconstruction Problem and Type Path Reconstruction Problem. The algorithms are presented in detail in Chapter 5 and Chapter 6 respectively. We provide detailed proofs of their correctness and for the claimed runtime and space bounds. In Chapter 7, we discuss several interesting problem variants and the applicability of our results to these problems. Chapter 8 concludes with an overview of the results of this master's thesis, compares our findings to related work and discusses possible directions for future research.

CHAPTER 2

Type Reconstruction and Normalization

The Type Reconstruction Problem and the Type Normalization Problem are formally introduced towards the end of this chapter in Section 2.7. In order to do so, a closer look at data layouts,formally called type maps, MPI's derived datatypes and type tree representations is necessary (Sections 2.1 to 2.3). We then present a formal framework for modeling type constructors and type trees as well as a cost model measuring the conciseness of type representations (Sections 2.5 and 2.6).

Interesting variations with a significant impact on asymptotically efficient algorithms for the Type Reconstruction Problem exist: If the most general `MPI_TYPE_CREATE_STRUCT` constructor is excluded, type trees degenerate to *type paths*. Although type trees allow for a potentially more efficient representation of a type map than type paths, the latter are easier to compute. Even more concise representations are possible when the tree structure of type trees is generalized to directed acyclic graphs (DAG), where equal nodes of a type tree can be folded into a single one. The Type Reconstruction and Normalization Problems presumably are harder to solve for *type DAG* representations. Type paths and DAGs are introduced formally in Section 2.7.1 and Section 2.7.2.

2.1 Type maps

MPI defines several basic datatypes that directly correspond to the basic (or elementary) datatypes in C or Fortran (e.g., `MPI_INT`, which maps to the C-type `int`; `MPI_DOUBLE` and `double`). In addition, MPI provides a set of *type constructors* so that users can define custom datatypes, called *derived datatypes*. This mechanism can describe arbitrarily complex, non-contiguous and heterogeneous application data, e.g., a column or row vector or a sub-matrix of a multi-dimensional matrix. The layout of such data can be described

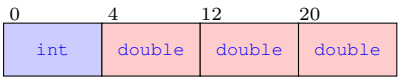


Figure 2.1: A graphical representation of the type map $\langle(\texttt{int}, 0), (\texttt{double}, 4), (\texttt{double}, 12), (\texttt{double}, 20)\rangle$.

explicitly by a sequence of basic datatypes together with their respective *displacements* in memory. Displacements are taken as offsets relative to a given start address and are measured either in bytes or in multiples of a given type. Such a sequence of pairs of a basic datatype plus a displacement value is called a *type map*. Equivalently, a type map may be seen as a sequence of basic datatypes plus a sequence of displacements. We use the two notations interchangeably, whichever is more convenient in the current context.

Basic MPI datatypes are predefined special cases of a general datatype. The base type `MPI_DOUBLE` for example defines the type map $\langle(\texttt{double}, 0)\rangle$, with one element of type `double` and displacement 0. A derived datatype that consists of one element of type `int` followed by three elements of type `double` (which in C can easily be defined with the `struct` operator), defines the type map

$$\langle(\texttt{int}, 0), (\texttt{double}, 4), (\texttt{double}, 12), (\texttt{double}, 20)\rangle \quad .$$

We assume that an `int` and a `double` take up 4 and 8 bytes of memory respectively, and ignore potential alignment issues. The same type map is represented graphically in Figure 2.1.

In this work, only basic datatypes with a one-to-one correspondence to elementary datatypes of the host language are used. Similar to the MPI standard, we forgo distinguishing between them to avoid notational overhead and instead refer to both as "base types". In examples we use elementary datatypes for type maps, which are essentially a direct image of data in memory. For type trees, which are an MPI specific concept, MPI base types are used. Formally, type maps are defined as follows.

Definition 1 (Type map)**.** *A* type map $M = (T, D)$ *of length* n *is a sequence of base types* T *plus a sequence of displacements* D*, both of length* n*. The* type sequence *(or type signature)* $T = \langle t_0, t_1, \ldots, t_{n-1}\rangle$ *consists of base types* t_i*. The* displacement sequence $D = \langle d_0, d_1, \ldots, d_{n-1}\rangle$ *consists of arbitrary integer displacements* d_i*, that is, the displacements are not required to be positive, distinct or in ascending order.*

Note that data elements need not appear in the same order as they do in memory and that an element may appear more than once. However, the ordering of elements imposed by the displacement sequence (which, as the name states, is a sequence and *not* a set) implies that data are accessed in a specific order: The type map $\langle(\texttt{double}, 0), (\texttt{int}, 8)\rangle$ is not equivalent to the type map $\langle(\texttt{int}, 8), (\texttt{double}, 0)\rangle$. A *homogeneous* type map is a type map where all base types t_i are the same. We denote homogeneous type maps as $M = (t, D)$, i.e., with a single base type instead of a sequence of base types.

A type map together with a start address buf defines a communication buffer that consists of n elements, where the i-th element has type t_i and is located in memory at

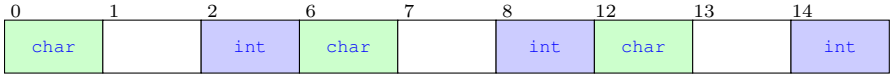


Figure 2.2: A graphical representation of the type map $\langle(\texttt{char}, 0), (\texttt{int}, 2), (\texttt{char}, 6), (\texttt{int}, 8), (\texttt{char}, 12), (\texttt{int}, 14)\rangle$.

address $buf + d_i$. All of MPI's communication operations exchange typed values and therefore have a `datatype` argument, for which any (basic or derived) datatype may be used.

2.2 MPI's derived datatype constructors

MPI provides a set of datatype constructors to specify heterogeneous and non-contiguous data layouts, that is, to *represent* arbitrarily complex type maps. Most of the type constructors *replicate* a given datatype (the `oldtype` argument, also called the *sub-type*) in some regular, semi-regular or irregular pattern. A more general constructor concatenates multiple different sub-types. Type constructors may be applied recursively, i.e., arbitrarily complex derived datatypes may serve as a sub-type.

The simplest datatype constructor,

```
MPI_TYPE_CONTIGUOUS(count, oldtype, OUT newtype)
```

replicates a sub-type (the `oldtype` argument) into `count` many consecutive locations. The resulting datatype corresponds to an array of `oldtype` values.

To determine their displacements, the sub-type's *extent* is used. A datatype's extent is defined to be the span from the first to the last byte occupied by elements of the datatype, possibly rounded up to meet alignment requirements [MPI15, p. 84]. In other words, for a type map $M = (\langle(t_0, d_0), \ldots, (t_{n-1}, d_{n-1})\rangle)$, the extent is

$$\max_i(d_i + \text{sizeof}(t_i)) - \min_i(d_i) + \epsilon \quad ,$$

where ϵ is the additional space required for correct alignment.

For a base type, the extent is typically equal to its size, i.e., to the number of bytes an element of this type occupies in memory. This, however, is not necessarily always true, since the extent of a datatype may be manipulated by the user. The definition of the datatype's extent is fairly complicated and convoluted (see [MPI15, p. 84-85, p. 104-110]). We omit this peculiar feature of MPI and simply assume that the extent of a datatype is always equal to the span from the first to the last byte occupied by its elements. It will become clear in the following sections that this does not alter the nature of the considered problems. We use a small subset of predefined datatypes, listed in Table 2.1 together with their corresponding C-type and assumed extent, for the purpose of illustrating examples. Refer to the MPI standard [MPI15, p. 26] for a full list of predefined MPI datatypes.

Table 2.1: Basic MPI datatypes used in this work and their corresponding C types and assumed extent.

MPI datatype	C datatype	Assumed extent
MPI_CHAR	char	1 byte
MPI_INT	int	4 bytes
MPI_FLOAT	float	4 bytes
MPI_DOUBLE	double	8 bytes

To give an example, assume that oldtype represents the type map $\langle(\texttt{char}, 0), (\texttt{int}, 2)\rangle$ and that the base types char and int have an extent of 1 and 4 bytes respectively. Note that oldtype's extent is 6 bytes and that the represented type map contains a gap of one byte between the two base types char and int. MPI_TYPE_CONTIGUOUS(3, oldtype, MY_TYPE) defines a derived datatype consisting of 3 consecutive replications of the type map represented by oldtype, i.e., MY_TYPE represents the type map

$$\langle(\texttt{char}, 0), (\texttt{int}, 2), (\texttt{char}, 6), (\texttt{int}, 8), (\texttt{char}, 12), (\texttt{int}, 14)\rangle \quad .$$

This type map is represented graphically in Figure 2.2.

The MPI_TYPE_VECTOR constructor,

```
MPI_TYPE_VECTOR(count, blocklength, stride,
                oldtype, OUT newtype)
```

is slightly more general than MPI_TYPE_CONTIGUOUS. It generates count many blocks that are stride many elements apart and contain blocklength many elements each. By using the same oldtype as before and setting count to 3, blocklength to 2 and stride to 4, MPI_TYPE_VECTOR creates a datatype with type map

$$\begin{aligned}\langle&(\texttt{char}, 0), (\texttt{int}, 2), (\texttt{char}, 6), (\texttt{int}, 8),\\ &(\texttt{char}, 24), (\texttt{int}, 26), (\texttt{char}, 30), (\texttt{int}, 32),\\ &(\texttt{char}, 48), (\texttt{int}, 50), (\texttt{char}, 54), (\texttt{int}, 56)\rangle \quad .\end{aligned}$$

The second block starts with the second row at displacement 24, which is equal to stride times the oldtype's extent. The third block follows at displacement 48, i.e., each two successive blocks are 24 bytes apart. The blocks are said to be *strided*, since their displacements follow a regular pattern. This constructor is generalized by the following two constructors.

```
MPI_TYPE_CREATE_INDEXED_BLOCK(count, blocklength, displacements[],
                              oldtype, OUT newtype)
```

```
MPI_TYPE_INDEXED(count, blocklengths[], displacements[],
                 oldtype, OUT newtype)
```

The MPI_TYPE_CREATE_INDEXED_BLOCK constructor allows irregular block displacements. The MPI_TYPE_INDEXED constructor additionally allows for a different replication count for each block. Using oldtype as before and letting count = 2, blocklengths = [1,3] and displacements = [1,2], MPI_TYPE_INDEXED generates a datatype with type map

$$\langle(\texttt{char},6),(\texttt{int},8),$$
$$(\texttt{char},12),(\texttt{int},14),(\texttt{char},20),(\texttt{int},22),(\texttt{char},28),(\texttt{int},30)\rangle \quad .$$

The first line contains the part of the type map generated by the first block of one replication of oldtime and the second line contains the three replications of the second block. The MPI_TYPE_CREATE_STRUCT constructor,

```
MPI_TYPE_CREATE_STRUCT(count, blocklengths[], displacements[],
                       oldtypes[], OUT newtype)
```

generates blocks of replications of different sub-types and is the most general. Contrary to the previous constructors, block displacements are given in bytes, and not in multiples of the sub-type's extent. Using count = 2, displacements = [0,8], blocklengths = [2,3] and the two types $\langle(\texttt{int},0)\rangle$ and $\langle(\texttt{double},0)\rangle$ as sub-types (the first with an extent of 4 bytes, the second with 8 bytes), MPI_TYPE_CREATE_STRUCT creates a datatype with type map

$$\langle(\texttt{int},0),(\texttt{int},4),$$
$$(\texttt{double},8),(\texttt{double},16),(\texttt{double},20)\rangle \quad .$$

For the constructors MPI_TYPE_VECTOR, MPI_TYPE_INDEXED and MPI_TYPE_CREATE_INDEXED_BLOCK, the alternate versions MPI_CREATE_TYPE_HVECTOR, MPI_CREATE_TYPE_HINDEXED and MPI_CREATE_TYPE_HHINDEXED_BLOCK exist. They specify block displacements in bytes rather than in multiples of the sub-type's extent. However, as with a type's extent, we abstract from such details and in our model consider all displacements to be measured in bytes. Therefore, these constructors do not have to be considered here. To improve readability and avoid the similar, overly long names of the type constructors, we make use of abbreviations listed in Table 2.2. The type constructors presented here are listed in order of increasing generality. The most general one, STRUCT, is in principle sufficient to represent any data layout, albeit incurring a large overhead for simple, regular layouts compared to a representation with e.g., the CONTIGUOUS or VECTOR constructors. This overhead causes notational redundancy that is inconvenient for the user and potentially incurs significant performance penalties. If a more specific constructor can be used to represent a type map, the representation is typically more *concise* and therefore leads to better performance. The next section formally defines what is meant by the *conciseness* of a type representation.

Table 2.2: Type constructors and their abbreviations.

Type constructor	Abbreviation
MPI_TYPE_CONTIGUOUS	CONTIGUOUS
MPI_TYPE_VECTOR	VECTOR
MPI_CREATE_TYPE_HVECTOR	HVECTOR
MPI_TYPE_INDEXED	INDEXED
MPI_CREATE_TYPE_HINDEXED	HINDEXED
MPI_TYPE_CREATE_INDEXED_BLOCK	INDEXED_BLOCK
MPI_CREATE_TYPE_HHINDEXED_BLOCK	HINDEXED_BLOCK
MPI_TYPE_CREATE_STRUCT	STRUCT

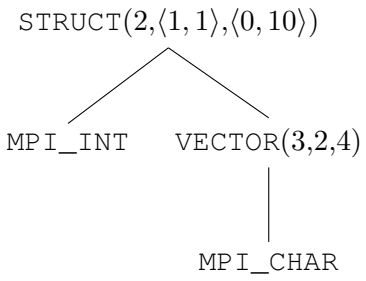


Figure 2.3: A possible type tree representation for the type map $\langle(\texttt{int}, 0), (\texttt{char}, 10), (\texttt{char}, 11), (\texttt{char}, 14), (\texttt{char}, 15), (\texttt{char}, 18), (\texttt{char}, 19)\rangle$.

2.3 Type tree representations

A derived datatype constructed with the type constructors presented in Section 2.2 can be viewed as a *type tree*.

Definition 2. *A type tree is a tree-like structure where*

- *the* leafs *correspond to the base types,*
- *inner nodes correspond to the application of a type constructor describing a repetition of a part of the type map or a concatenation of different parts, and*
- *an edge indicates that the datatype represented by a child node is used as a sub-type in the constructor associated with the parent node.*

Figure 2.3 shows an example type tree using the `VECTOR` and `STRUCT` constructors to represent a type map that consists of one integer followed by several non-contiguously stored `char` values. A constructor's `oldtype` arguments are represented by its child nodes. The `newtype` argument, which returns a handle to the constructed datatype, is

omitted. The represented type map can be obtained by an ordered traversal of the type tree, as detailed in Section 2.5 or [THRZ99].

Note that several different type tree representations may exist for a given type map. Space and cost efficient representations of user- or application-defined derived datatypes are required to enable MPI implementations to handle them efficiently. These representations are internal and usually not exposed to the users of MPI.

The problem of computing the most efficient (or optimal) type tree representation out of a given one was termed the Type Normalization Problem. The closely related Type Reconstruction Problem instead asks for the most efficient type tree representation of a given type map. Both problems are defined formally in Section 2.7. However, before being able to do so, a formal model for type tree representations and the cost model have to be introduced.

2.4 Performance impact of optimal derived datatypes

In this section, we study the impact of different derived datatype definitions. Given an $n \times n$ matrix of integer values, assume that we want to communicate the data of the first row and the first column. Since two-dimensional arrays are stored in row-major order in C, the elements of a single row are stored in a contiguous memory area, while the elements of a column are stored non-contiguously, in a regularly strided pattern.

This data layout can be described with different derived datatypes. We compare three natural variants:

- `INDEXED_BLOCK_TYPE`: Lists the displacements of all the $2n$ elements explicitly.
- `INDEXED_TYPE`: The elements of the row are described as a single block of length n, followed by an explicit list of displacements for the n elements of the column.
- `STRUCT_VEC_TYPE`: The elements of the column are described as a vector type with stride n (note that in an $n \times n$ matrix stored in row-major order, the elements of a column are n elements apart). This vector type is then concatenated with the row elements, which are again described as a single block of length n.

The code to construct these datatypes can be found in Listing 2.1 and Listing 2.2. The resulting type trees are illustrated in Figure 2.5.

The benchmarking procedure is given in Listings 2.1 and proceeds as follows. The performance impact of the three derived datatypes is measured by sending the row and column data from process 0 to process 1 and back to process 0 (lines 35 – 47). Process 0 measures the time it requires to send the data and receive it back from process 1.

First, the MPI environment is set up and the communication buffers are initialized. The two-dimensional matrix of 1000×1000 values is stored locally at process 0 in `send_buf` and the result is received into `recv_buf`. Process 1 receives the data into `sendrecv_buf`, from which the data is directly sent back to process 0. In the second step, the derived datatype description for the desired data layout is constructed.

```c
#include <mpi.h>
#include <stdlib.h>

int main(int argc, char** argv) {
    MPI_Init(&argc, & argv);
    int rank, tag = 0;
    MPI_Status status;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // set up communication buffers and initialize data
    int *send_buf = NULL, *recv_buf = NULL, *sendrecv_buf = NULL;
    const size_t n = 1000;
    if(rank == 0) {
        send_buf = malloc(n * n * sizeof(int));
        for(size_t i = 0; i < n; i++) {
            for(size_t j = 0; j < n; j++)
                send_buf[i * n + j] = i * n + j;
        }
        recv_buf = malloc(n * n * sizeof(int));
   }
   if(rank == 1)
        sendrecv_buf = malloc(n * n * sizeof(int));

    // Create derived datatype: INDEXED_BLOCK_TYPE
    MPI_Datatype datatype;
    int displacements[2 * n];
    for(size_t i = 0; i < n; i++) {
        displacements[i] = i;
        displacements[n + i] = i * n;
    }
    MPI_Type_create_indexed_block(2 * n, 1, displacements, MPI_INT, &datatype);
    MPI_Type_commit(&datatype);

    // benchmark
    for(size_t i = 0; i < 100; i++) {
        if(rank == 0) {
            double time = MPI_Wtime();
            MPI_Send(send_buf, 1, datatype, 1, tag, MPI_COMM_WORLD);
            MPI_Recv(recv_buf, 1, datatype, 1, tag, MPI_COMM_WORLD, &status);
            printf("%0.10f\n", (MPI_Wtime() - time) * 1000);
        }
        if (rank == 1) {
            MPI_Recv(sendrecv_buf, 1, datatype, 0, tag, MPI_COMM_WORLD, &status);
            MPI_Send(sendrecv_buf, 1, datatype, 0, tag, MPI_COMM_WORLD);
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }

    MPI_Type_free(&datatype);
    MPI_Finalize();
    return EXIT_SUCCESS;
}
```

Listing 2.1: Benchmarking the performance impact of different derived datatypes.

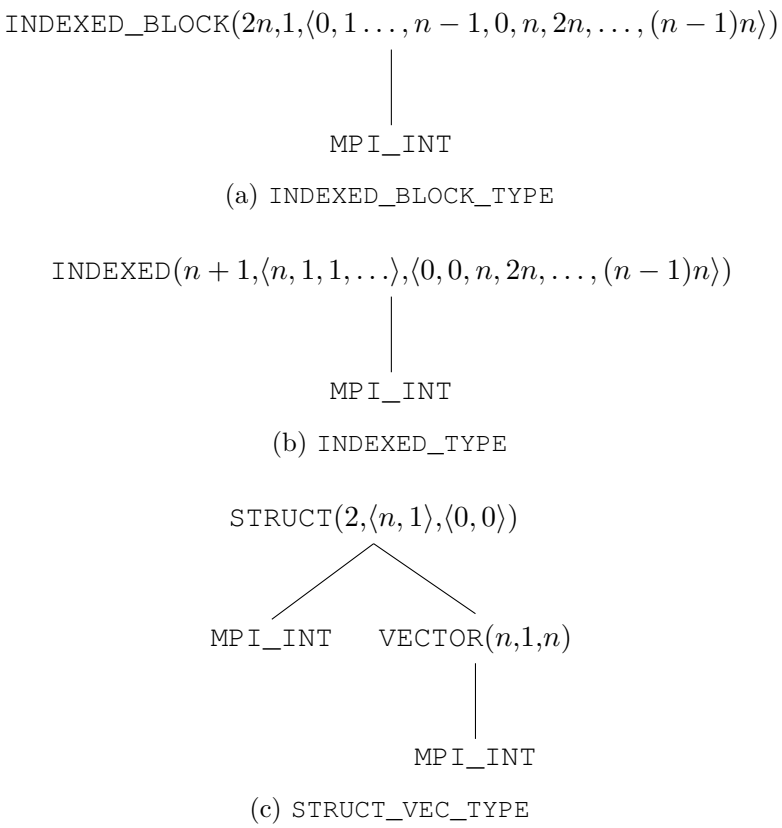


Figure 2.4: Type tree representations of three different derived datatypes describing the data of the first row plus the second column of an $n \times n$ matrix of `int` values.

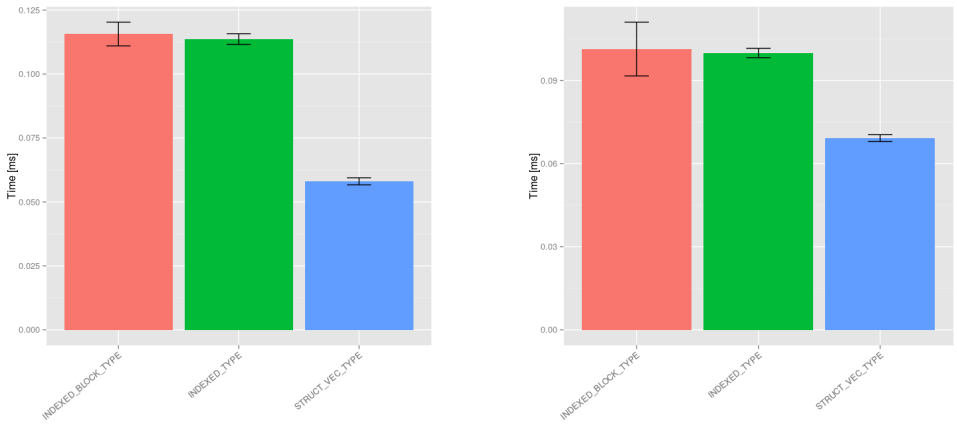
Listing 2.1 contains the code for constructing the datatype `INDEXED_BLOCK_TYPE` (lines 24 –32). The construction of the two remaining derived datatypes, `INDEXED_TYPE` and `STRUCT_VEC_TYPE`, is given in Listing 2.2. The derived datatype is then used to communicate the elements of one row and one column of the matrix from process 0 to process 1 and back, as detailed above.

The performance impact of the different datatype representations is evaluated on a medium-sized distributed memory machine at TU Wien, nick-named "Jupiter". Jupiter consists of 3 compute nodes, where each node consists of two AMD Opteron 6134 processors (2.3 GHz, 8 cores per processor) and 32 GB main memory. The nodes are connected via Ethernet and an Ifiniband-QDR switch (type MT4036). Jupiter runs Linux with Kernel version 2.6.32-573.3.1.el6.x86_64. The source code was complied with gcc version 4.4.7 and tested with two MPI implementations, namely NEC MPI 1.3.1 and OpenMPI 1.8.4. The benchmark was executed 100 times for each derived datatype and MPI library. In Figure 2.5, we report the mean execution time together with the 5% and 95% confidence intervals.

With both MPI implementations, execution times are roughly equal using `INDEXED_`–

```
1     // INDEXED_TYPE
2     MPI_Datatype datatype;
3     int blocklengths[2 * n];
4     int displacements[2 * n];
5     for(size_t i = 0; i < 2 * n; i++)
6         blocklengths[i] = 1;
7     for(size_t i = 0; i < n; i++) {
8         displacements[i] = i;
9         displacements[n + i] = i * n;
10    }
11    MPI_Type_indexed(2 * n, blocklengths, displacements, MPI_INT, &datatype);
12
13
14    // STRUCT_VEC_TYPE
15    MPI_Datatype VEC1_TYPE;
16    MPI_Type_vector(n, 1, 1, MPI_INT, &VEC1_TYPE);
17    MPI_Type_commit(&VEC1_TYPE);
18
19    MPI_Datatype VEC2_TYPE;
20    MPI_Type_vector(n, 1, n, MPI_INT, &VEC2_TYPE);
21    MPI_Type_commit(&VEC2_TYPE);
22
23    MPI_Datatype datatype;
24    int blocklengths[2] = {1, 1};
25    MPI_Aint displacements[2] = {0, 0};
26    MPI_Datatype oldtypes[2] = {VEC1_TYPE, VEC2_TYPE};
27
28    MPI_Type_create_struct(2, blocklengths, displacements, oldtypes, &datatype);
```

Listing 2.2: Alternative derived datatypes.



(a) OpenMPI (b) NEC MPI

Figure 2.5: Performance comparison of different derived datatypes for a given data layout.

BLOCK_TYPE and INDEXED_TYPE. With STRUCT_VEC_TYPE, performance increases drastically with execution times reduced to around one half to two thirds respectively. The first observation indicates that both MPI implementations perform type normalization to some extent. MPI most likely detects that the n row elements listed explicitly by INDEXED_TYPE are stored contiguously and can thus be treated as a single, contiguous memory block. As Gropp et al. [GHTT11] point out, very little type normalization seems to be performed by current MPI implementations. The optimizations are typically based on simple heuristics, which we detail in Chapter 3. The STRUCT_VEC_TYPE performs much better because the regular structure of the column data is captured well by the VECTOR constructor used as a sub-type. This derived datatype cannot be obtained by applying the heuristic optimizations on INDEXED_BLOCK_TYPE or INDEXED_TYPE. It is thus not constructed by the two MPI libraries used and has to be supplied by a knowledgeable user. This datatype is optimal in the formal model we adopt (Sections 2.5 and 2.6) and highlights the performance gains that can potentially be achieved by solving the Type Normalization Problem to optimality. Similar results were observed for other data layouts, including heterogeneous data.

2.5 Formal model

To model the problem of optimal (or least-cost) type trees, we use a convenient abstraction of the set of MPI type constructors (see Definition 3). The mapping is straight forward and detailed in Section 2.5.1.

The displacements of a type map refer to memory addresses relative to a given start address. Thus their values are limited to a finite range and it is safe to assume that a displacement value can be stored with a constant number of bits. Some of the type constructors defined by MPI measure displacements and strides in bytes, while others use multiples of the sub-type's extent. While these semantics are convenient for users, we measure all displacements and strides in bytes to reduce notational overhead.

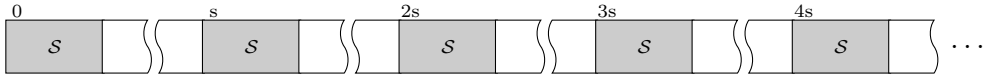
We define a *shift operation* $(+)$ for displacement sequences and type maps, which adds a scalar value to all displacements. For a displacement sequence $D = \langle d_0, \ldots, d_{n-1} \rangle$ and a scalar k, $D + k = \langle d_0 + k, \ldots, d_{n-1} + k \rangle$. For a type map $M = (T, D)$ with $D = \langle d_0, \ldots, d_{n-1} \rangle$ and a scalar k we have that

$$M + k = (T, D + k) = (T, \langle d_0 + k, d_1 + k, \ldots, d_{n-1} + k \rangle) \quad .$$

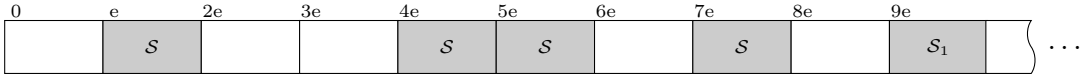
We abbreviate a shift operation with a negative scalar by $(-)$.

The *replication operation* $(*)$ replicates a type sequence a given number of times: For a type sequence $T = \langle t_0, \ldots, t_{n-1} \rangle$ and a scalar k, $k * T = \langle t'_0, \ldots, t'_{kn-1} \rangle$ with $t'_i = t_{i \bmod n}$ for $n \leq i < kn$.

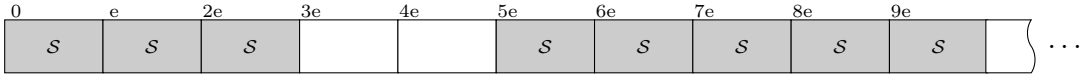
Definition 3 (Type Constructors, type tree)**.** *Any type map can be represented, possibly more concisely than listing all elements explicitly, as a* type tree *built out of the following* type constructors *(or* nodes*).*



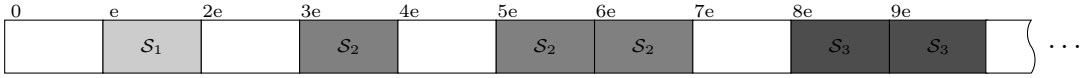
(a) Type map represented by a vec constructor with stride s. Basic idea: Sub-type is repeated at regular intervals.



(b) Type map represented by an idx constructor with displacements $\langle e, 4e, 5e, 7e, 9e, \ldots \rangle$. Basic idea: Sub-type is repeated at irregular intervals.



(c) Type map represented by an idxbuc constructor with bucket stride $s = e$, bucket sizes $\langle 3, 5 \rangle$ and displacements $\langle 0, 5e \rangle$. Basic idea: Sub-type is repeated in consecutive locations within irregularly displaced buckets.

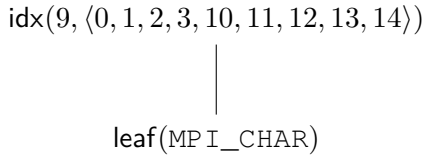


(d) Type map represented by a strc constructor with displacements $\langle e, 3e, 5e, 6e, 8e, 9e \rangle$ and sub-type array $\langle \mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_2, \mathcal{T}_2, \mathcal{T}_3, \mathcal{T}_3 \rangle$. Basic idea: Combine multiple distinct sub-types.

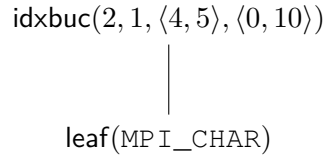
Figure 2.6: Sketches of the characteristics of type maps represented by different type constructors. All examples use a generic sub-type $\mathcal{S}$ ($\mathcal{S}_1$, $\mathcal{S}_2$) which may be any basic or derived MPI datatype. The extent of all the sub-types $\mathcal{S}$, $\mathcal{S}_1$ and $\mathcal{S}_2$ is assumed to be e.

- *A* leaf, $\mathsf{leaf}(t)$ *represents the base type* t *with displacement 0, i.e., the type map* $M = (\langle t \rangle, \langle 0 \rangle)$.
- *A* vector, $\mathsf{vec}(c, s, \mathcal{S})$, *with count* c, *stride* s *and type tree* $\mathcal{S}$ *representing a type map* $N = (T, D)$, *describes the replication of the type map* N *at relative displacements* $0, s, 2s, \ldots, (c-1)s$, *i.e., a type map*[1] $M = (c * T, \langle D, D+s, \ldots, D+(c-1)s \rangle)$. $\mathcal{S}$ *is also called the* sub-type.
- *An* index, $\mathsf{idx}(c, \langle i_0, i_1, \ldots, i_{c-1} \rangle, \mathcal{S})$, *with displacements* $\langle i_0, i_1, \ldots, i_{c-1} \rangle$, *describes the replication of the type map* N *(represented by* $\mathcal{S}$*) at relative displacements* i_0, $i_1, \ldots, i_{c-1}$, *i.e., a type map* $M = (c * T, \langle D + i_0, D + i_i, \ldots, D + i_{c-1} \rangle)$.
- *An* index bucket $\mathsf{idxbuc}(c, s, \langle b_0, \ldots, b_{c-1} \rangle, \langle i_0, \ldots, i_{c-1} \rangle, \mathcal{S})$, *with bucket sizes* $b_0, \ldots, b_{c-1}$, *describes the concatenation of* c *type maps* N_i *at relative displacements* $\langle i_0, \ldots, i_{c-1} \rangle$. *The* i*-th type map* N_i *is the replication of the type map* N *(represented by* $\mathcal{S}$*) at relative displacements* $0, s, 2s, \ldots, (b_i - 1)s$. *Thus, an index bucket*

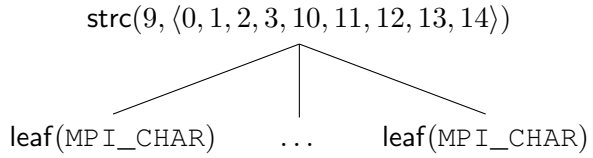
[1] We abuse notation a bit and define a sequence of sequences to be just a sequence, i.e, $\langle D, D+s \rangle = \langle \langle d_0, \ldots, d_{n-1} \rangle, \langle d_0 + s, \ldots, d_{n-1} + s \rangle \rangle = \langle d_0, \ldots, d_{n-1}, d_0 + s, \ldots, d_{n-1} + s \rangle$.



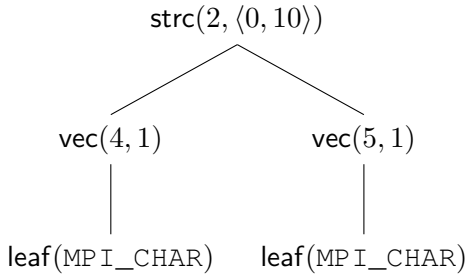
(a) Trivial representation with idx



(b) More concise representation with idxbuc



(c) Trivial representation with strc (with a total of nine child nodes)



(d) Representation using a combination of vec and strc

Figure 2.7: Examples illustrating the type constructors defined in Definition 3. The type map represented by each type tree is $M = (\langle \texttt{char}, \ldots \rangle, \langle 0, 1, 2, 3, 10, 11, 12, 13, 14 \rangle)$ (four contiguous elements of type `char` starting at offset 0 followed by five more contiguous elements starting at offset 10).

describes a type map with displacement sequence

$$\langle D + i_0 + 0s, \ldots, D + i_0 + (b_0 - 1)s, \ldots, D + i_{c-1} + 0s, \ldots, D + i_{c-1} + (b_{c-1} - 1)s \rangle$$

and type sequence $\langle b_0 * T, \ldots, b_{c-1} * T \rangle$.

- *A* struct *(or* heterogeneous index*),* $\mathsf{strc}(c, \langle i_0, \ldots, i_{c-1} \rangle, \langle \mathcal{S}_0, \ldots, \mathcal{S}_{c-1} \rangle)$*, with subtypes* $\mathcal{S}_i$ *representing type maps* $N_i = (T_i, D_i)$*, describes the concatenation of* c *type maps* $N_0, \ldots, N_{c-1}$ *(represented by* $\mathcal{S}_0, \ldots, \mathcal{S}_{c-1}$*) at relative displacements* $i_0, \ldots, i_{c-1}$*, i.e., a type map*

$$M = (\langle T_0, \ldots, T_{c-1} \rangle, \langle D_0 + i_0, \ldots, D_{c-1} + i_{c-1} \rangle) \quad .$$

The basic idea of each constructor is illustrated in Figure 2.6 and we give some examples in the following. A type map $(\langle \texttt{char}, \texttt{char}, \texttt{char} \rangle, \langle 3, 5, 7 \rangle)$ can be described

by the type tree $\mathcal{T} = \mathsf{idx}(1, \langle 3\rangle, \mathsf{vec}(3, 2, \mathsf{leaf}(\texttt{MPI\_CHAR})))$. Any type map $M = (\langle t_0, \ldots, t_{n-1}\rangle, \langle d_0, \ldots, d_{n-1}\rangle)$ can trivially be represented with the strc constructor as $\mathsf{strc}(n, \langle d_0, \ldots, d_{n-1}\rangle, \langle t_0, \ldots, t_{n-1}\rangle)$, implying that the defined set of constructors is sufficient to represent any type map. If M contains only one base type t, it can furthermore trivially be represented as $\mathsf{idx}(n, \langle d_0, \ldots, d_{n-1}\rangle, t)$. More examples illustrating each of the constructors are given in Figure 2.7.

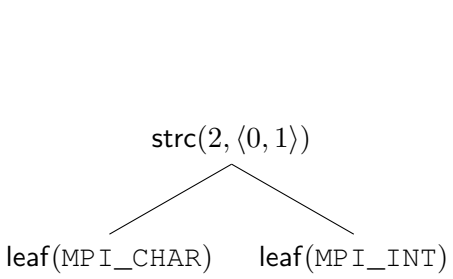
Note that child nodes represent the sub-type arguments of a constructor and that we abbreviate longer type sequences of the same base type as $\langle t, \ldots\rangle$. The number of occurrences of t can easily be derived from the length of the displacement sequence. To reduce notational overhead, we sometimes use a type map instead of a type tree as sub-type. This way, the example given before can also be stated as $\mathcal{T} = \mathsf{idx}(1, \langle 3\rangle, \mathsf{vec}(3, 2, (\langle \texttt{char}\rangle, \langle 0\rangle)))$. We may use MPI base types, C datatypes and leaf nodes with a concrete base type interchangeably, since the mapping is one-to-one. For example, we may write $\mathsf{strc}(c, \langle \ldots\rangle, \langle \texttt{char}, \texttt{int}, \ldots\rangle)$ instead of $\mathsf{strc}(c, \langle \ldots\rangle, \langle \mathsf{leaf}(\texttt{MPI\_CHAR}), \mathsf{leaf}(\texttt{MPI\_INT}), \ldots\rangle)$. Furthermore, we use $M = \mathcal{T}$ to indicate that $\mathcal{T}$ represents M. We refer to vertices of type trees also as *type nodes*, where each type node is one of the constructors. A type node representing a leaf constructor is also called a leaf node. Analogously, type nodes representing a vec, idx, idxbuc or strc constructor are called vec, idx, idxbuc and strc node respectively. The constructors that require an array of displacements (idx, idxbuc, strc) are grouped together as *irregular* constructors, whereas the vec constructor is referred to as a *regular* constructor. This distinction comes from the fact that the vec constructor concatenates the sub-type in a regular pattern, whereas the others describe an irregular pattern. Note that all leaf nodes of a type tree are necessarily leaf constructors, which however can never be used as an internal node.

Each type tree *represents* one type map, which can be obtained by an ordered traversal of the tree. The converse is not true: a non-trivial type map will often have several possible type tree representations. This is exemplified in Figure 2.7, where four different type tree representations are shown for the same type sequence. Actually, *infinitely many* type tree representations exist for *any* type map, since

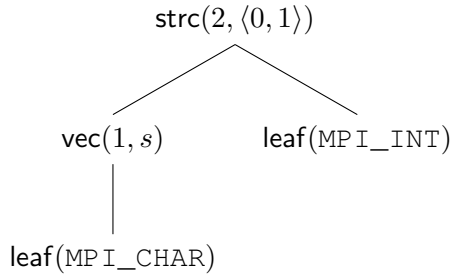
1. a representation can be blown up to arbitrary size by adding nodes that do not increase the length of the represented type map compared to the type map represented by the sub-type, and

2. for any type tree that consists of at least two irregular constructors, infinitely many type trees with the same structure but different displacement arrays exist, where all variants represent the same type map.

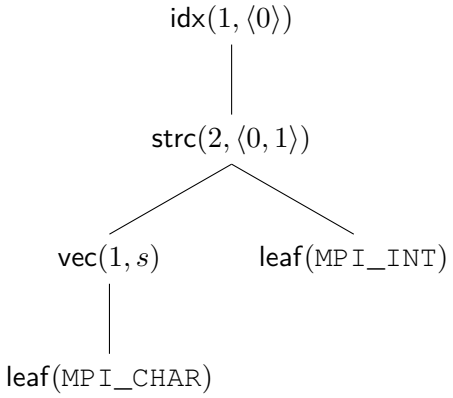
TO be more concrete, a type tree $\mathcal{T} = \mathsf{vec}(1, s, \mathcal{S})$ (with arbitrary stride s) represents the exact same type map as the type tree $\mathcal{S}$. The same is true for the type tree $\mathcal{T}' = \mathsf{idx}(1, \langle 0\rangle, \mathcal{S})$. Similar examples can easily be constructed with an idxbuc or a strc node. Furthermore, the type tree $\mathcal{T} = \mathsf{idx}(1, \langle -10\rangle, \mathsf{idx}(1, \langle 10\rangle, \mathcal{S}))$ represents the same type map as $\mathcal{T}' = \mathsf{idx}(1, \langle 0\rangle, \mathsf{idx}(1, \langle 0\rangle, \mathcal{S}))$. Figure 2.8 illustrates this issue by providing several different valid (if rather unnatural) type tree representations for a simple type



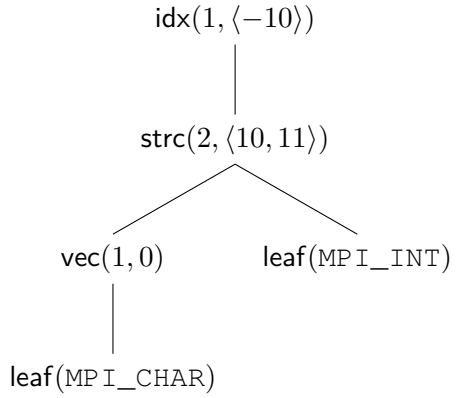
(a) A natural representation with a minimal amount of type nodes.



(b) Adding a vec node with count 1 does not change the represented type map...



(c) Neither does the addition of an idx node with count 1 and displacement array ⟨0⟩...



(d) Or different values in the displacement arrays that "cancel" each other out.

Figure 2.8: Several different type tree representations of decreasing "naturalness" for the type map $(\langle \texttt{char}, \texttt{int} \rangle), \langle 0, 1 \rangle)$.

map. Fortunately, the number of representations one would consider "useful", "sensible" or – especially – "efficient" is comparatively small, as Chapter 4 shows.

The process of obtaining the type map represented by a type tree is called *flattening*. Algorithm 2 gives a straight-forward implementation for the constructors of Definition 3, assuming that type nodes are represented with some form of C-style structures. Possible implementations for the constructor nodes are sketched in Listing 1. We make no claim that this is a particularly efficient way of implementing flattening, but it suffices for the purposes of this work. For a more efficient approach, see Träff et al. [THRZ99].

2.5.1 Mapping of MPI datatype constructors

Mapping MPI's derived datatype constructors to the type constructors of the formal model is straight forward and almost one-to-one. We first highlight their differences and then show how each of the MPI derived datatype constructors introduced in Section 2.2 is expressible by the type constructors given in Definition 3.

Listing 1: Possible structures for representing leaf, idxbuc, vec and strc nodes in type trees. The structure for idx nodes is equal to the one for idxbuc nodes minus the members *s* and *buckets*. Each type node is assumed to store as the first two members a variable *kind* indicating which constructor it encodes and a variable *cost* holding the total cost of the type tree rooted at the node (see Section 2.6 for details). We assume that any type node can be cast to the general type **Typenode**.

```
1  enum kind = {leaf, vec, idx, idxbuc, strc}
2  struct {
3  |  kind kind
4  |  int cost          /* cost of type tree rooted at this node */
5  } Typenode
6
7  struct {
8  |  Basetype basetype                                  /* base type */
9  } Leaf
10
11 struct {
12 |  kind kind ← idxbuc
13 |  int cost
14 |  int c                                                  /* count */
15 |  int s                                              /* bucket stride */
16 |  int buckets[]                                       /* bucket sizes */
17 |  int disp[]                                         /* displacements */
18 |  Typenode subtype                                      /* sub-type */
19 } IdxBuc
20
21 struct {
22 |  kind kind ← vec
23 |  int cost
24 |  int c                                                  /* count */
25 |  int s                                                 /* stride */
26 |  Typenode subtype                                     /* sub-type */
27 } Vec
28
29 struct {
30 |  kind kind ← strc
31 |  int cost
32 |  int c                                                  /* count */
33 |  int disp[]                                         /* displacements */
34 |  Typenode subtype[]                              /* array of sub-types */
35 } Strc
```

Algorithm 2: Flattening procedure constructing the type map represented by a given type tree $\mathcal{T}$. The procedure is called with a base displacement, which will normally be 0.

```
1  M ← (∅, ∅)                                              /* empty type map */
2  Function Flatten(Typenode T, int base)
3      switch T.kind do
4          case leaf                                        /* leaf */
5              Add (base, T.basetype) to M
6          case vec                                         /* vector */
7              for i ← 0; i < T.c; i++ do
8                  Flatten(T.subtype, base + i · T.s)
9          case idx                                         /* index */
10             for i ← 0; i < T.c; i++ do
11                 Flatten(T.subtype, base + T.disp[i])
12         case idxbuc                                      /* index bucket */
13             for i ← 0; i < T.c; i++ do
14                 for j ← 0; j < T.buckets[i]; j++ do
15                     Flatten(T.subtype, base + T.disp[i] + j · T.s)
16         case strc                                        /* struct */
17             for i ← 0; i < T.c; i++ do
18                 Flatten (T.subtype[i], base + T.disp[i])
19     return M
```

The type nodes in the formal model are a slight simplification of MPI's derived datatype constructors:

- As was stated in Section 2.2, we assume that a datatype's extent is the span of the first to the last byte it occupies in memory. Therefore the datatype's extent does not have to be stored explicitly.
- Contrary to MPI, which measures displacements and strides either in bytes or multiples of the extent of the sub-type, the formal model defines all displacements and strides in bytes. The constructors `HVECTOR`, `HINDEXED` and `HINDEXED_BLOCK` measure displacements and strides in bytes but are otherwise equivalent to `VECTOR`, `INDEXED` and `INDEXED_BLOCK`. Thus, they do not have to be incorporated into the formal model.
- Some of MPI's datatype constructors replicate the sub-type into several blocks, where each block contains a certain number of either contiguous or strided replications of the sub-type (given by the `blocklength` or `blocklengths[]` argument

of the VECTOR, INDEXED_BLOCK, INDEXED and STRUCT constructors). Except for INDEXED, the corresponding type constructors in the formal model (vec, idx and strc) save this extra value or sequence of values. If a sub-type is indeed such a block of contiguous or strided replications, this information can easily be encoded by the sub-type with the help of the vec constructor.

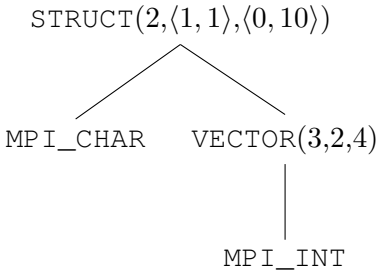
- Compared to the VECTOR constructor, CONTIGUOUS requires only one less argument. In our formal model, we map this constructor to the VECTOR constructor with stride 1.

An arbitrary MPI derived datatype T can be mapped to a type tree $\mathcal{T}$ using only the type constructors of the formal model as follows. If T is a base type, it is directly modeled by a leaf node with the base type as argument. Otherwise, T has a sub-type (oldtype) S with extent e. Assuming that $\mathcal{S}$ represents S in the formal model, T is represented by $\mathcal{T}$ as follows.

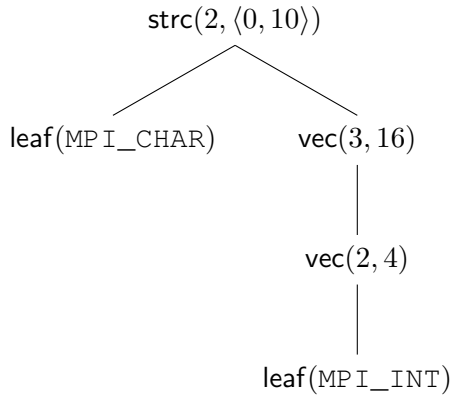
$$
\begin{aligned}
&\mathsf{T} = \texttt{CONTIGUOUS}(c, \mathsf{S}) \\
&\quad \rightarrow \mathcal{T} = \mathsf{vec}(c, e, \mathcal{S}) \\
&\mathsf{T} = \texttt{VECTOR}(c, b, s, \mathsf{S}) \\
&\quad \rightarrow \mathcal{T} = \mathsf{vec}(c, se, \mathsf{vec}(b, e, \mathcal{S})) \\
&\mathsf{T} = \texttt{INDEXED\_BLOCK}(c, b, \langle i_0, i_1, \ldots, i_{c-1}\rangle, \mathsf{S}) \\
&\quad \rightarrow \mathcal{T} = \mathsf{idx}(c, \langle i_0 e, i_1 e, \ldots, i_{c-1} e\rangle, \mathsf{vec}(b, e, \mathcal{S})) \\
&\mathsf{T} = \texttt{INDEXED}(c, \langle b_0, b_1, \ldots, b_{c-1}\rangle, \langle i_0, i_1, \ldots, i_{c-1}\rangle, \mathsf{S}) \\
&\quad \rightarrow \mathcal{T} = \mathsf{idxbuc}(c, e, \langle b_0, b_1, \ldots, b_{c-1}\rangle, \langle i_0, i_1, \ldots, i_{c-1}\rangle, \mathcal{S}) \\
&\mathsf{T} = \texttt{STRUCT}(c, \langle b_0, b_1, \ldots, b_{c-1}\rangle, \langle i_0, i_1, \ldots, i_{c-1}\rangle, \langle \mathsf{S}_0, \mathsf{S}_1, \ldots, \mathsf{S}_{c-1}\rangle) \\
&\quad \rightarrow \mathcal{T} = \mathsf{strc}(c, \langle i_0, i_1, \ldots, i_{c-1}\rangle, \langle \mathsf{vec}(b_0, e, \mathcal{S}_0), \mathsf{vec}(b_1, e, \mathcal{S}_1), \ldots, \mathsf{vec}(b_{c-1}, e, \mathcal{S}_{c-1})\rangle)
\end{aligned}
$$

Figure 2.9 gives an example illustrating the differences between MPI's datatype constructors and the formal model. In an MPI implementation, the mapped type tree is supposed to be used only as the internal representation of a user-defined derived datatype. Transforming it back to a representation using MPI's derived datatype constructors is usually not necessary. It is certainly possible to perform such a transformation by inverting the rules given above.

Note that the extent e of a base type may differ for different platforms, implying that the internal representation of a derived datatype is platform dependent. Although the structure and used constructors are the same, the mapped type trees may store different values for displacements and strides (see Figure 2.10) This however does not constitute a problem: Although MPI's communication operations exchange typed values, the type information itself is not transmitted or shared between participating processes. In other words, each process constructs its own type tree representation. MPI does however take care of correctly converting typed data if it is communicated between heterogeneous machines.

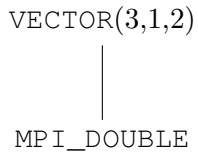


(a) Type tree with MPI datatype constructors

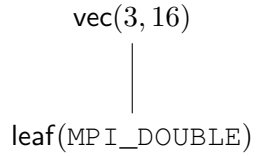


(b) Type tree in the formal model

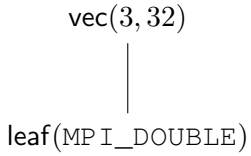
Figure 2.9: An example type tree constructed with MPI derived datatype constructors and its translation to the formal model. The represented type map is $(\langle \texttt{char}, \texttt{int}, \ldots \rangle, \langle 0, 10, 14, 26, 30, 42, 46 \rangle)$.



(a) A simple derived datatype. The stride argument of the VECTOR constructor is measured in multiples of the base type's extent.



(b) Assumed extent of MPI_DOUBLE: 8 bytes.



(c) Assumed extent of MPI_DOUBLE: 16 bytes.

Figure 2.10: Different extents for datatypes result in type trees storing different values for displacements and strides.

2.6 Cost model

Recall that a type tree $\mathcal{T}$ represents a type map M if $\texttt{Flatten}\ (\mathcal{T}, 0) = M$. As can be seen in e.g., Figure 2.7, a type map often has multiple different type tree representations. MPI's philosophy is to not define any performance requirements for its operations, instead leaving it to implementations to provide the best possible solution for a given system. It is thus not at all clear what an efficient or "good" representation looks like, which may bar users from utilizing derived datatypes, instead reverting to manual packing and unpacking routines. This issue was addressed by several authors who formulated *self-consistent performance guidelines* for MPI's communication operations [TGT10] as well as parallel I/O [GKR+08] and derived datatypes [GHTT11]. Ideally, one should be able to define the required data layout in the most natural way and trust the MPI library implementation to find the most efficient representation. The notion of efficiency of a representation is relative to how the type tree is used or processed by the MPI implementation. Different cost models may be useful, depending on the context in which derived datatypes are used as well as the target platform.

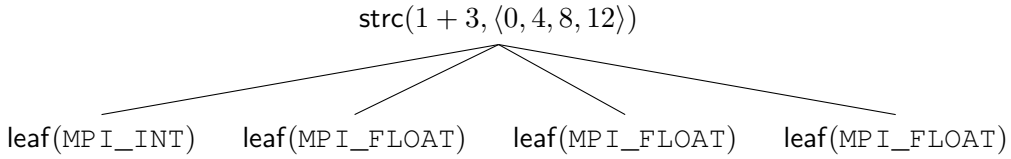
To process a derived datatype, each of the base types has to be located, e.g., by the ordered traversal shown in Algorithm 2. The processing cost as well as the storage cost of a type tree is thus proportional to the number of nodes it consists of. While leaf and vec nodes can trivially be stored using only constant memory, the irregular constructors require space proportional to the count c to store the arrays of displacements, bucket sizes and sub-types. These arrays require time for processing as well as space to be stored and the processing cost is thus also related to their size.

Träff [Trä14] introduced a simple, yet flexible additive cost model to capture these observations by defining the cost of a type node to be "the number of words that must be stored to process the node". We extend it to cover all the constructors listed in Definition 3. A strc node for example requires to store the kind of the node (leaf, vec, idx, idxbuc or strc), the count and pointers to the arrays of displacements and sub-types. Processing a strc node entails a per-element lookup cost for the arrays of displacements and sub-types.

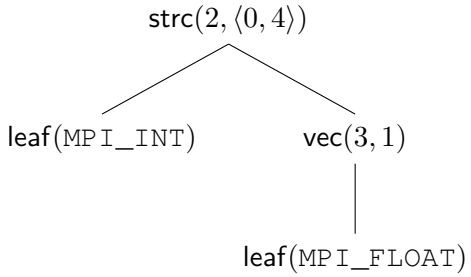
Definition 4 (Type node costs)**.**

$$\begin{aligned}
\text{cost}(\mathsf{leaf}(t)) &= K_{\mathsf{leaf}} \\
\text{cost}(\mathsf{vec}(c, s, \mathcal{T})) &= K_{\mathsf{vec}} \\
\text{cost}(\mathsf{idx}(c, \langle \ldots \rangle, \mathcal{T})) &= K_{\mathsf{idx}} + cK_{\text{lookup}} \\
\text{cost}(\mathsf{idxbuc}(c, s, \langle \ldots \rangle, \langle \ldots \rangle, \mathcal{T})) &= K_{\mathsf{idxbuc}} + 2cK_{\text{lookup}} \\
\text{cost}(\mathsf{strc}(c, \langle \ldots \rangle, \langle \ldots \rangle)) &= K_{\mathsf{strc}} + 2cK_{\text{lookup}}
\end{aligned}$$

To keep the cost model flexible, the constant cost per node and the per-element lookup cost can be adjusted to also reflect other costs. The only assumption we do make is that the cost values are always greater than zero. To match the C-style structure used



(a) Trivial representation with cost a cost of 25.



(b) Exploiting the partially regular structure of the data reduces the cost to 20.

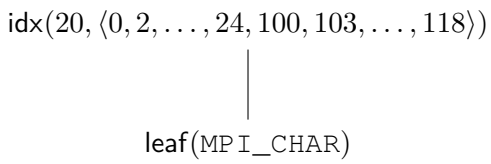
Figure 2.11: Exploiting (partially) regular structure of data may reduce cost. The represented type map is $M = (\langle \texttt{int}, \texttt{float}, \ldots \rangle, \langle 0, 4, 8, 12 \rangle)$.

to represent type nodes (Listing 1), we use the following values for examples throughout this work. For a leaf node its kind and (a pointer to) its base type t have to be stored. For efficient implementations (Chapters 5 and 6) it is necessary to store the cost of a type tree with its root node. A member *cost* was included in the structure representing a type node. For a leaf node a total of three words need to be stored, and thus we take the cost of a leaf node to be $K_{\mathsf{leaf}} = 3$. The costs for the other nodes can easily be deduced as $K_{\mathsf{vec}} = K_{\mathsf{idx}} = K_{\mathsf{strc}} = 5$ and $K_{\mathsf{idxbuc}} = 7$. We furthermore use $K_{\text{lookup}} = 1$.

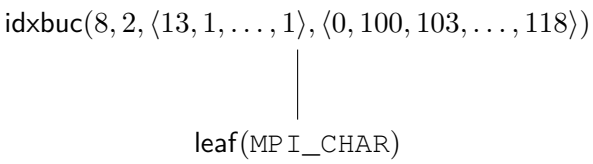
We define the cost of a type tree $\mathcal{T}$ as the sum of the costs of its type nodes $\mathcal{T}_i$:

$$\text{cost}(\mathcal{T}) = \sum_i \text{cost}(\mathcal{T}_i)$$

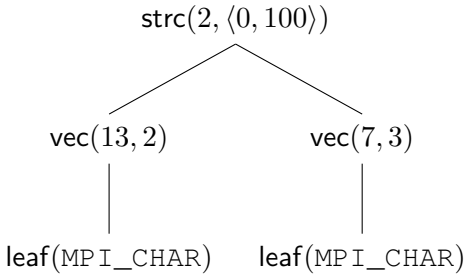
With this cost model it is clear that type trees with less and more specialized nodes are cheaper. However, a representation with a minimal amount of nodes is not necessarily the cheapest: As an example, assume a type map that consists of one `integer` value plus an array of k `float` values, i.e., $M = (\langle \texttt{int}, \texttt{float}, \ldots \rangle, \langle 0, 4, \ldots, 4k \rangle)$. Different type tree representations of this type map with $k = 3$ are given in Figure 2.11. The trivial representation with the strc constructor simply lists all $k+1$ base types. A representation of less cost is possible by taking into account that the k `float` values are stored contiguously. The trivial representation has a cost of $K_{\mathsf{strc}} + 2(k+1)K_{\text{lookup}} + (k+1)K_{\mathsf{leaf}}$, while the second one has cost $K_{\mathsf{strc}} + 2K_{\text{lookup}} + K_{\mathsf{vec}} + 2K_{\mathsf{leaf}}$. Note that the cost of the second representation is constant for any k, while the cost of the trivial representation grows linearly with k. Thus, the difference in costs can be made arbitrarily large by increasing k.



(a) Trivial representation. Cost: 28.



(b) Using idxbuc reduces the cost to 26.



(c) This representation uses the vec constructor to exploit the partially regular structure of the type map and the strc constructor to concatenate two different sub-types. Cost: 25.

Figure 2.12: Three type tree representations for the type map $M = (\langle \texttt{char}, \ldots \rangle, \langle 0, 2, \ldots, 24, 100, 103, \ldots, 118 \rangle)$ consisting of a total of 20 `char` types, highlighting the usefulness of the strc constructor for homogeneous type maps.

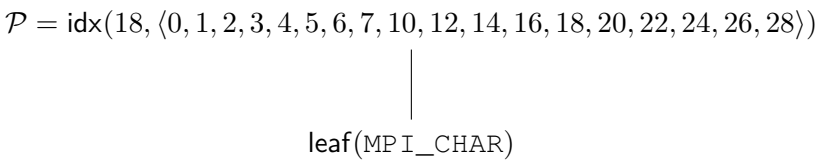
The strc constructor is useful not only for heterogeneous type maps, as the example in Figure 2.12 shows. For the homogeneous type map $M = (\langle \texttt{char}, \ldots \rangle, \langle 0, 1, 3, \ldots, 39 \rangle)$ containing a total of 21 base types, the representation consisting of the strc and vec constructors is optimal (i.e., no other type tree representation with less cost exists for M).

To give an additional example, the type trees shown in Figure 2.8 have a cost of (from 2.8a to 2.8d) 15, 20, 26 and 26. The last two type trees are of equal cost since they consist of the same nodes, only with different values in their displacement sequences.

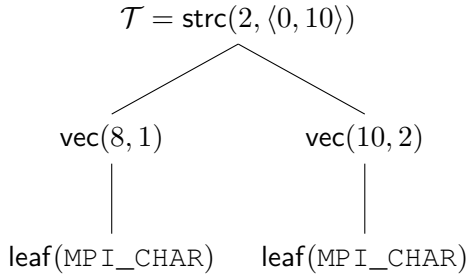
2.7 Formal problem definition

Using the formal model and cost model presented in the previous sections, we can now formally define the Type Reconstruction Problem.

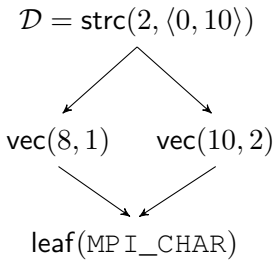
> Type Reconstruction Problem
> *Instance*: A type map M of length n.
> *Task*: Find a least-cost (or optimal) type tree $\mathcal{T}$ representing M. A type tree $\mathcal{T}$ is optimal if $\text{cost}(\mathcal{T}) \leq \text{cost}(\mathcal{T}')$ for any other type tree $\mathcal{T}'$ representing M.



(a) Type path representation. Cost: 26.



(b) Type tree representation. Cost: 25.



(c) Type DAG representation. Cost: 22.

Figure 2.13: The type map $(\langle \texttt{char}, ...\rangle, \langle 0, 1, \ldots, 7, 10, 12, \ldots, 28\rangle)$ consisting of 18 `char`'s, represented with a type path, type tree and type DAG. These representations are optimal in the sense that no type path, type tree or type DAG of less cost than the ones shown here exist. That is, no type path representation of less cost than $\mathcal{P}$ exists for the given type map.

This master's thesis is mostly concerned with type tree representations, which are used in the vast majority of related work (see Chapter 3 for an explicit list of published articles using type tree representations). However, possibly more efficient representations exist when the underlying structure is generalized to *directed acyclic graphs* (DAG), where equivalent nodes that occur multiple times in a type tree are folded into one node. We suspect that some of the related work implicitly assumes and works with a DAG structure, since the used type trees are often only loosely defined. On the other hand, type tree representations can be computed more efficiently if the set of considered constructors is restricted to those with only one sub-type, i.e., if only *type paths* are constructed. The difference between type trees, type paths and type DAGs is illustrated in Figure 2.13 and these problem variants are defined formally in the next sections.

2.7.1 Type DAGs

The definition of type trees (Definition 2) directly carries over to type DAGs. The only difference is that a node in a type DAG may have arbitrarily many predecessors (or incoming edges), i.e., a type may be used as the sub-type of multiple constructors, or even multiple times by the same strc constructor. In general, a DAG does not contain a unique root node, but may of course contain a designated *start node*. A DAG can be depicted as a hierarchical structure similar to a tree: The start node is at the very top and edges

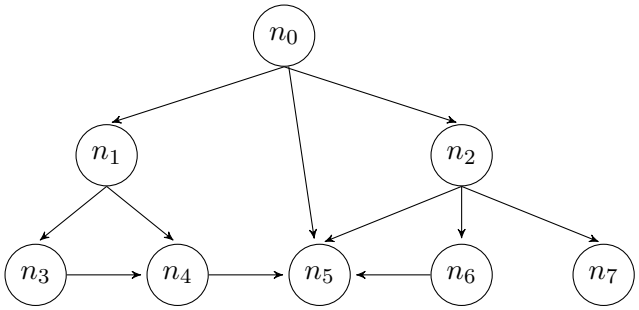
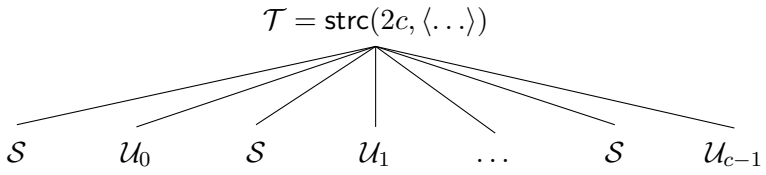
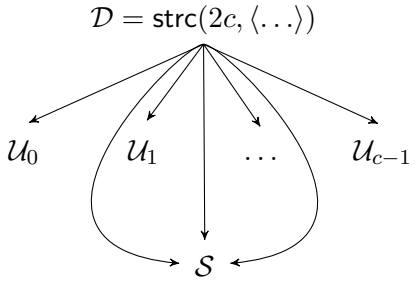


Figure 2.14: A simple DAG, depicted as a hierarchical structure. Contrary to a tree, where edges may be orientated only downwards, a hierarchically drawn DAG may also contain sideways orientated edges. It must not contain any upward edges, though.



(a) Type tree representation.



(b) Type DAG representation.

Figure 2.15: Type DAGs can be arbitrarily more concise than type trees, as this example shows. Both the type tree $\mathcal{T}$ and the type DAG $\mathcal{D}$ represent the same type map. The sub-type $\mathcal{S}$, which occurs multiple times, is folded into a single node in the DAG, thus saving the cost for all its other occurrences.

may only be orientated downwards or sideways, but never upwards. Figure 2.14 shows an example, with n_0 being the start node. A type DAG $\mathcal{D}$ with start node $\mathcal{N}$ represents a certain type map M that can be obtained by the same ordered traversal as used for type trees, namely the flattening procedure given in Algorithm 2, i.e., $M = \texttt{Flatten}(\mathcal{N}, 0)$.

Figure 2.15 gives an example of the savings that can potentially be achieved with type DAGs. Assume that the type tree $\mathcal{T}$ in Figure 2.15a is an optimal type tree for some type map M. It consists of c equal sub-types $\mathcal{S}$ interleaved with c pair-wise different

sub-types $\mathcal{U}_i$. The cost of $\mathcal{T}$ is thus

$$\text{cost}(\mathcal{T}) = c \cdot \text{cost}(\mathcal{S}) + \sum_{i=0}^{c-1} \text{cost}(\mathcal{U}_i) \quad .$$

Since the c instances of the sub-type $\mathcal{S}$ are all the same, $\mathcal{T}$ can be "folded" into a DAG. Figure 2.15b shows the resulting type DAG $\mathcal{D}$. It represents the same type map M, but is of less cost:

$$\text{cost}(\mathcal{D}) = \text{cost}(\mathcal{S}) + \sum_{i=0}^{c-1} \text{cost}(\mathcal{U}_i)$$

In other words, it saves $(c-1)$ times the cost of $\mathcal{S}$. As this example shows, compared to type tree, arbitrarily large savings are possible with type DAGs.

> Type DAG Reconstruction Problem
> *Instance*: A type map M of length n.
> *Task*: Find a least-cost (or optimal) type DAG representing M.

The definitions of the Type Reconstruction and Normalization Problems can be easily adapted for type DAGs. However, as we discuss in Chapter 7, the algorithm developed for constructing optimal type trees does not seem to be useful for the more general problem of constructing optimal type DAGs.

2.7.2 Type paths

Type maps that can be represented optimally with the help of only the `VECTOR` and `INDEXED_BLOCK` constructors are common in applications, e.g, if sub-matrices need to be communicated [Trä14, KT15]. In this case, type trees degenerate to simple paths, because the considered constructors do not take more than one sub-type. In this work, we call type representations restricted to the leaf, vec and idx constructors *type paths*. An *extended type path* may additionally consist of the idxbuc constructor.

> Type Path Reconstruction Problem
> *Instance*: A homogeneous type map M of length n.
> *Task*: Find a least-cost (or optimal) type path $\mathcal{P}$ representing M.

The Extended Type Path Reconstruction Problem is defined analogously. The Type Path Reconstruction Problem problem can be solved quite efficiently in $O(n\sqrt{n})$ time, as was shown by Träff [Trä14]. In Chapter 6, we present our algorithm for this problem, which improves the worst-case bound to $O(n \log / \log \log n)$ time.

2.7.3 Type normalization

The Type Normalization Problem, which is closely related to the Type Reconstruction Problem, is to transform a given type tree to an optimal one.

> Type Normalization Problem
> *Instance*: A type tree $\mathcal{S}$.
> *Task*: Find a least-cost (or optimal) type tree $\mathcal{T}$ representing the same type map as $\mathcal{S}$.

The problem is analogously defined for type paths and type DAGs. The Type Normalization Problem, in particular the Type Path Normalization Problem, are discussed in more detail in Section 7.2.

CHAPTER 3

State of the art and related work

Historically, many MPI implementations handled derived datatypes poorly. Trivial manual packing and unpacking code often performed significantly better than an implementation relying on MPI's derived datatype capabilities [GLS99a, RMG03]. A large amount of research since focused on more efficient datatype processing (see e.g., [THRZ99, RMG03, BGST03, MT08, KHS12, SKH13, PG15], some of which are discussed in more detail below), which evidences the need of efficient internal representations of derived datatypes.

The Type Reconstruction and Normalization Problems, both for type trees and type DAGs, appeared within this context. The Type DAG Reconstruction Problem was first stated as an interesting open problem by Mir and Träff [MT08]. A paper by Gropp et al. [GHTT11], which proposed explicit, self-consistent performance guidelines for derived datatypes, formally defines the Type Normalization Problem. It observes that enforcing these guidelines would require MPI implementations to solve this problem, which was conjectured to be NP-hard. This conjecture was substantiated in a recent paper by Träff [Trä14], who stated which type constructors are assumed to make the problem hard. They also showed that the Type Path Reconstruction Problem for type maps of length n is feasible in $O(n\sqrt{n})$ time if the set of considered constructors is restricted to `CONTIGUOUS`, `VECTOR` and `INDEXED`. Furthermore, the Type Path Normalization Problem with the same set of constructors can be solved in time proportional to the depth of the type path and the length of the longest occurring list of displacements. Their work is the most relevant for this thesis, and is discussed in more detail in Section 6.1.

To the best of our knowledge, no other approaches to construct provably cost-optimal type DAG, type tree or type path representations exist. Indeed, Gropp et al. [GHTT11] when investigating the performance of derived datatypes in several MPI implementations, found that very little type normalization is performed, even in trivial cases.

There is however a substantial amount of research using type tree representations where heuristic, local optimizations are employed, which do not provide any guarantees

on the quality of the constructed type trees. Typically, a subset of the following heuristic optimizations is applied:

1. Specializing (or promoting) constructors to less general ones: Ross et al. [RMG03] check for `STRUCT` constructors if they can be replaced with the `HINDEXED` constructor, which is assumed to be less costly. Their assumption is that this is always the case for homogeneous data layouts. However, with a flexible cost model such as the one used in this work, this is not necessarily true (refer to Section 2.6 for an example). In a similar manner, Schneider et al. [SKH13] replace with `CONTIGUOUS` any `VECTOR` constructor with equal `stride` and `blocklength` arguments. In [KHS12], Kjolstad et al. use a total of four specialization passes. After the `STRUCT` to `HINDEXED` conversion outlined above, it is checked whether further specialization to `HVECTOR`, `VECTOR` and finally `CONTIGUOUS` is possible. Details about performing such constructor specializations can be found in a technical report [KHS11] accompanying the published work [KHS12]. Prabhu and Gropp [PG15] seem to perform very similar optimizations, which unfortunately are not described in detail.

2. Compressing (or merging) constructors: Kjolstad et al. [KHS12], Schneider et al. [SKH13] and Prabhu et al. [PG15] merge contiguous sub-types into the parent type. In this way, the total amount of used constructors is reduced by exploiting the `blocklength` argument which most of the type constructors offer. An additional compression is applied by [KHS12], where two consecutive `STRUCT` or `HINDEXED` constructors are merged into one. Curiously, the `INDEXED` constructor is not considered.

3. Ross et al. [RMG03] coalesce contiguous indexed regions, i.e., multiple blocks of an `INDEXED` constructor, where the displacements are such that the blocks form a contiguous memory area.

4. Kjolstad et al. [KHS12] also perform an optimization that changes the represented type map. When a contiguous datatype is used in a send operation, the `CONTIGUOUS` constructor can be saved by integrating its `count` argument into the send operation. That is, instead of sending n elements of a contiguous type that replicates a sub-type `count` many times, n times `count` many elements of the sub-type are sent.

A different strand of research focuses on the automatic generation of derived datatypes with the aim of making their usage easier. A derived datatype can be generated either out of a C datatype definition [RP06], C++ classes [TT08] or manually written packing code [KHS12]. These approaches again perform only local heuristic optimizations, if any.

Other representations than the graph-based type DAGs and type trees are possible. Ross et al. use a representation called dataloops [RMG03], which are closely related to

type paths. Their approach does not consider the `STRUCT` constructor and therefore has to revert to considering all data as untyped bytes if the data layout contains more than one base type. Gropp et al. [GLS99a] use a representation based on a finite automaton, where some sub-types of a type tree are replaced by special, highly optimized leaf nodes. Jenkins et al. [JDB+14] developed a representation of derived datatypes that exposes fine-grained parallelism and is thus suitable for processing by GPUs. Their idea is to split arrays of displacements and sub-types, which are of arbitrary size, from the remaining constant amount of information stored with each type node.

Several authors [SWP04, WWP04] propose to improve the performance of non-contiguous data communication by facilitating advanced network features. The explicit packing of message data into a contiguous buffer can be avoided if the network hardware is capable of directly communicating strided (or even more complex) data, which enhances performance significantly.

CHAPTER 4

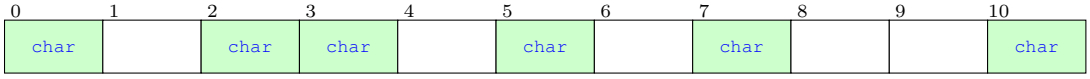
Characterizing type trees

In this chapter, we investigate the structure of optimal type trees. Some properties of optimal type trees are hinted at in the previous chapters and we formalize these observations in the following. In particular, in the discussion of MPI's type constructors in Section 2.2, we state for certain constructors that they describe the *replication* of a given type map at certain displacements. While the idx constructor uses arbitrary, possibly irregular displacement values, the vec constructor replicates into strided (or regular) locations. The idxbuc constructor is a mixture of both, while the strc constructor behaves in a fundamentally different way. To construct a type tree representation for a given type map, it has to be checked if the type map follows such a pattern of irregular or strided replications, that is, if a prefix of the type map *repeats* in a particular way. This notion is defined formally in Section 4.1.

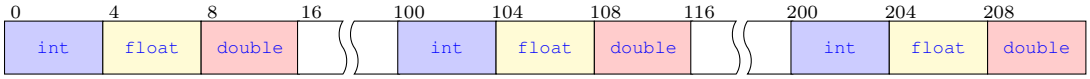
The properties shown in the following sections are crucial for efficient algorithms solving the Type Reconstruction Problem and the Type Path Reconstruction Problem, which are presented in Chapters 5 and 6 respectively. These algorithms compute optimal type trees for the special case of *aligned* type maps (see Definition 13). The structure of an optimal type tree can be characterized well for this special case and we derive several crucial properties useful for proving the correctness of our algorithms. A solution for a general type map can be derived without increasing the asymptotic time and space bounds from an optimal type tree and *partial* results constructed along the way.

This chapter contains many definitions, lemmas and corollaries and their proofs. We separate the investigation of type tree properties and the presentation of algorithms into multiple chapters for several reasons:

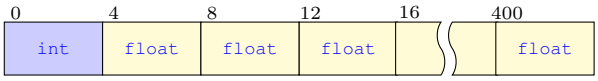
1. The properties presented in this chapter are required to solve both the Type Reconstruction Problem and the Type Path Reconstruction Problem efficiently.



(a) The prefix of length 1 is repeated, but not strided.



(b) The prefix of length 3 is repeated and strided.



(c) No repeated prefix (except for the whole type map itself).

Figure 4.1: Examples of repeated and strided prefixes.

2. This chapter provides a detailed presentation of properties of (optimal) type trees and intends to further the understanding of their nature.

3. By clearly separating these observations from how they are employed in the algorithms, this chapter may serve as a starting point for deriving other, possibly more efficient algorithms.

4.1 Repeated and strided prefixes

Recall the definition of a type map given in Section 2.1: A type map $M = (T, D)$ of length n is a sequence of base types T plus a sequence of displacements D, both of length n. The type sequence (or type signature) $T = \langle t_0, t_1, \ldots, t_{n-1} \rangle$ consists of base types t_i and the displacement sequence $D = \langle d_0, d_1, \ldots, d_{n-1} \rangle$ consists of arbitrary integer displacements d_i. A *segment* as well as a *prefix* of a type map, displacement or type sequence refer to certain parts of the whole sequence:

Definition 5 (Segment, Prefix)**.** *A* segment *of an n-element displacement sequence from index i to j is denoted by $D[i, j] = \langle d_i, d_{i+1} \ldots, d_j \rangle$, $0 \leq i \leq j < n$. A segment starting at index 0 is also called a* prefix*. We denote a prefix of length q as D_q, i.e., $D_q = D[0, q-1]$.*

These terms are defined analogously for type sequences and the definition extends to type maps in a natural way: The segment $M[i, j]$ of a type map $M = (T, D)$ of length n is defined as $M[i, j] = (T[i, j], D[i, j])$. A segment of length q starting at index 0 is again called a prefix and denoted by M_q.

If a type map M is equivalent to the replication of some prefix, the prefix is said to be *repeated* in M.

Definition 6 (Repeated prefix of a displacement sequence)**.** *A* repeated prefix *of length q in an n-element* displacement sequence *D is a prefix D_q s.t. q is a divisor of n and for all i, j, $1 \leq i < n/q$, $0 \leq j < q$ we have that*

$$D[j] - D[0] = D[iq + j] - D[iq] \quad .$$

For type sequences, the notion of a repeated prefix is slightly different:

Definition 7 (Repeated prefix of a type sequence)**.** *A* repeated prefix *of length q in an n-element* type sequence T *is a prefix T_q s.t. q is a divisor of n and for all i, $0 \leq i < n$ we have that*

$$T[i] = T[i \bmod q] \quad .$$

Definition 8 (Repeated prefix of a type map)**.** *A prefix M_q of a type map $M = (T, D)$ is a repeated prefix if both T_q and D_q are repeated prefixes in T and D respectively.*

Figure 4.1a contains an example of a repeated prefix, with the prefix of length 1 being repeated at displacements 0, 2, 3, 5, 7 and 10. A repeated prefix allows for a representation via the idx constructor. In particular, this type map can be represented as $\mathsf{idx}(6, \langle 0, 2, 3, 5, 7, 10\rangle, \mathsf{leaf}(\texttt{MPI\_CHAR}))$. The sub-type can be an arbitrarily complex type map, as the example in Figure 4.1b shows. Here, the prefix M_3 is repeated and thus the type map M can be represented as $\mathsf{idx}(3, \langle 0, 100, 200\rangle, M_3)$. Contrary to the previous two, the type map in Figure 4.1c does not contain a repeated prefix of length less than n. Nevertheless, a type tree representation exists, since the example type map can be represented as

$$\mathsf{strc}(401, \langle 0, 4, 8, \ldots, 400\rangle, \langle \mathsf{leaf}(\texttt{MPI\_INT}), \mathsf{leaf}(\texttt{MPI\_FLOAT}), \ldots\rangle \quad .$$

This is true for any type map, as the following corollary shows:

Corollary 1. *Any type map can be represented by a type tree.*

Proof. Let $M = (\langle t_0, \ldots, t_{n-1}\rangle, D)$ denote an arbitrary type map of length n. M can be represented by the type tree $\mathsf{strc}(n, D, \langle \mathsf{leaf}(t_0), \ldots, \mathsf{leaf}(t_{n-1})\rangle)$. □

It follows directly from the definitions that any homogeneous type map can be represented with one idx plus one leaf constructor, since a type map $M = (\langle t, t, \ldots\rangle, D)$ of length n can be represented as $\mathsf{idx}(n, D, \mathsf{leaf}(t))$. Equivalently, any homogeneous type map can also be represented by the idxbuc constructor as $\mathsf{idxbuc}(n, s, \langle 1, 1, \ldots\rangle, D, \mathsf{leaf}(t))$ with arbitrary bucket stride s, or with the strc constructor as $\mathsf{strc}(n, D, \langle \mathsf{leaf}(t), \mathsf{leaf}(t), \ldots\rangle)$. Note that in the considered cost model these representations may in principle be of less cost than the representation with the idx constructor, even though intuitively they are of course more redundant. For our algorithms, we cannot make any assumptions beyond the definition of the cost model given in Section 2.6 and thus have to consider these representations. For examples, however, we do not explicitly mention a possible representation with the idxbuc constructor with all bucket size values being 1, as well as a representation with the strc constructor with all sub-types being equal. They can trivially be derived for any representation with an idx constructor.

Definition 9 (Strided prefix)**.** *A* strided prefix D_q *of a displacement sequence D is a repeated prefix that additionally fulfills*

$$D[q] - D[0] = D[(i+1)q] - D[iq]$$

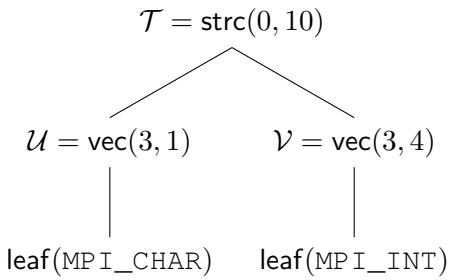


Figure 4.2: An optimal type tree representation for the type map $M = (\langle \texttt{char}, \texttt{char}, \texttt{char}, \texttt{int}, \texttt{int}, \texttt{int} \rangle, \langle 0, 1, 2, 10, 14, 18 \rangle)$. The subtree $\mathcal{U}$ represents the type map $(\langle \texttt{char}, \texttt{char}, \texttt{char} \rangle, \langle 0, 1, 2 \rangle)$, which is a prefix M. The subtree $\mathcal{V}$ represents the type map $(\langle \texttt{int}, \texttt{int}, \texttt{int} \rangle, \langle 0, 4, 8 \rangle)$. Note that this is not a segment of M, since the displacements do not match.

for all i, $0 \leq i < n/q - 1$, *where* $s = D[q] - D[0]$ *is the* stride. *A prefix* M_q *of a type map* $M = (T, D)$ *is a strided prefix if* D_q *is a strided prefix in* D *and* T_q *is a repeated prefix in* T.

Informally, a repeated prefix is a strided prefix if the repetitions occur at regular displacements. Analogously to repeated prefixes, a strided prefix additionally allows for a representation with the vec constructor. For example, the type map in Figure 4.1b can alternatively be represented as $\mathsf{vec}(3, 100, M_3)$.

Note that by definition, a type map, displacement or type sequence is always a strided prefix of itself. Any type map M can therefore be represented as $\mathsf{idx}(1, \langle 0 \rangle, M)$ or $\mathsf{vec}(1, 0, M)$. Although such representations may not be natural and, as Corollary 4 shows, are never part of an optimal type tree representation, they do exist. Defining repeated and strided prefixes in this way reduces the number of special cases that have to be dealt with in the algorithms. We typically omit this *trivially strided prefix* in examples since they need not be considered when constructing optimal type trees, as the next section shows.

4.2 Optimal type trees

In this section, we take a closer look at properties of optimal type trees. Note that type paths are a special case of type trees for which the following observations hold as well.

The fundamental observation for our algorithms is that any type map can be described by either a concatenation of the same kind of shorter type maps (and thus by the idx and idxbuc constructors and possibly also by the vec constructor) or by a concatenation of different, but shorter type maps (and thus by the strc constructor). Type maps of length 1 are trivially represented by a single leaf node.

Recall that a type tree is optimal if and only if no type tree of less cost representing the same type map exists. It is easy to see that a solution for the Type Reconstruction

PROBLEM exhibits *optimal sub-structure* [CLRS09], i.e., that the *dynamic programming principle* (or *principle of optimality*) applies. Given an optimal type tree representing a type map M, each subtree $\mathcal{S}$ must represent the corresponding type map optimally. The *corresponding type map* of a type tree is the type map represented by the type tree. For a type tree $\mathcal{S}$, the corresponding type map N is $N = \texttt{Flatten}(\mathcal{S}, 0)$. The statement requires only that *some* type map is represented optimally, i.e., the corresponding type map N is not necessarily a segment of M. Instead, the represented type map N is an *aligned* segment of M. This is illustrated in Figure 4.2 and expanded on in Section 4.3. The following lemma proves that an optimal type tree exhibits optimal sub-structure.

Corollary 2. *Given an optimal type tree $\mathcal{T}$ representing a type map M, each subtree $\mathcal{S}$ of $\mathcal{T}$ is an optimal representation of the corresponding type map.*

Proof. By contradiction: Assume that an optimal type tree $\mathcal{T}$ for the type map M contains a subtree $\mathcal{S}$, which is a non-optimal representation of the type map N. Since $\mathcal{S}$ is non-optimal, a different representation $\mathcal{U}$ of less cost exists for N. Replacing $\mathcal{S}$ with $\mathcal{U}$ in $\mathcal{T}$ reduces the cost of $\mathcal{T}$ by the cost of $\mathcal{S}$ minus the cost of $\mathcal{U}$, since the cost of $\mathcal{U}$ is strictly less than the cost of $\mathcal{S}$. This contradicts the assumption that $\mathcal{T}$ is optimal. □

Definition 10 (Structure of an optimal type tree)**.** *An optimal type tree $\mathcal{T}$ for a type map $M = (T, D)$ of length n is either*

1. $\mathcal{T} = \mathsf{leaf}(t_0)$*, a single leaf node with base type t_0 if n equals 1; or*
2. $\mathcal{T} = \mathsf{vec}(c, s, \mathcal{S})$*, where the prefix M_q of length $q = n/c$ is a strided prefix in M with stride s and $\mathcal{S}$ is an optimal type tree for M_q; or*
3. $\mathcal{T} = \mathsf{idx}(c, \langle i_0, \ldots, i_{c-1}\rangle, \mathcal{S})$*, where the prefix M_q of length $q = n/c$ is a repeated prefix in M, $\mathcal{S}$ is an optimal type tree for M_q and the displacements $i_0, \ldots, i_{c-1}$ are such that $\texttt{Flatten}(\mathcal{T}, 0) = M$; or*
4. $\mathcal{T} = \mathsf{idxbuc}(c, s, \langle b_0, \ldots, b_{c-1}\rangle, \langle i_0, \ldots, i_{c-1}\rangle, \mathcal{S})$*, where the prefix M_q of length $q = n/c$ is a repeated prefix in M, $\mathcal{S}$ is an optimal type tree for M_q and the displacements $i_0, \ldots, i_{c-1}$ together with the bucket sizes $b_0, \ldots, b_{c-1}$ and the bucket stride s are such that $\texttt{Flatten}(\mathcal{T}, 0) = M$; or*
5. $\mathcal{T} = \mathsf{strc}(c, \langle i_0, \ldots, i_{c-1}\rangle, \langle \mathcal{S}_0, \ldots, \mathcal{S}_{c-1}\rangle)$*, where the subtrees $\mathcal{S}_j$, $0 \leq j < c$, are optimal type trees for the type maps $N_j = \texttt{Flatten}(S_j, 0)$ which together with the displacements $i_0, \ldots, i_{c-1}$ are such that $\texttt{Flatten}(\mathcal{T}, 0) = M$.*

A few aspects of this characterization of optimal type trees must be pointed out.

- The stride s for a representation via a vec node is easily derived as $s = D[q] - D[0]$.
- By the definition of the vec constructor, the type map represented by the subtree $\mathcal{S}$ is replicated at displacements 0, s, 2s,... This does however not imply that $D[0]$ is equal to 0, since $\mathcal{S}$ may describe a type map where the first displacement value is different from 0. Refer to Figure 4.3 for an example.

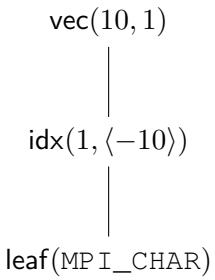
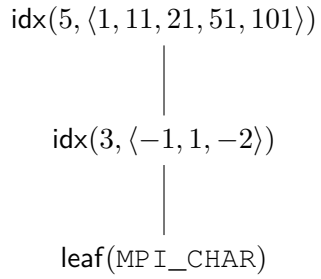
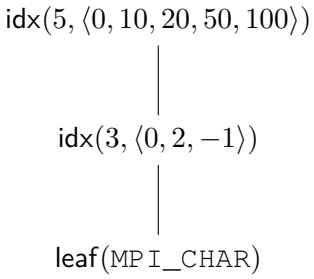


Figure 4.3: An optimal type tree for the type map $(\langle \texttt{char}, \ldots \rangle, \langle -10, -9, \ldots, -1 \rangle)$. Although the root node is a vec node, the type tree represents a type map with $D[0] \neq i$.



(a) The displacement values of the root node are not equal to $D[0]$, $D[3]$,...,$D[12]$.



(b) A more natural representation.

Figure 4.4: Two optimal type trees for the type map $(\langle \texttt{char}, \ldots \rangle, \langle 0, 2, -1, 10, 12, 9, \ldots, 100, 102, 99 \rangle)$.

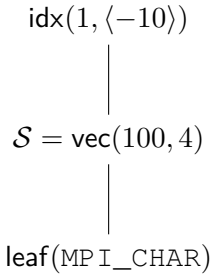


Figure 4.5: An optimal type tree for the type map $(\langle \texttt{char}, \ldots \rangle, \langle -10, -6, -2, 2, \ldots, 386 \rangle)$. The idx node with count $c = 1$ shifts the type map represented by the subtree $\mathcal{S}$, but does not affect the represented type map in any other way.

- Contrary to the stride for vec nodes, the displacements for an idx, idxbuc or strc node are not easily derivable. As the example in Figure 4.4a shows, the displacements depend on the sub-type. Infinitely many different representations with the same structure can be derived by adding some value x to the displacements of one idx node and $-x$ to the displacements of the other. Although a more natural representation exists (Figure 4.4b), both are of same cost and equally valid.

- The above characterization allows for the vec, idx, idxbuc and strc nodes to have a count of $c = 1$. This is a necessity for the irregular constructors, as Figure 4.5 exemplifies. If we required for c to be strictly larger than 1, we would not be able to exploit the regular structure of the example type map which is captured by the vec constructor. In this example, the idx node with count 1 shifts the represented type map by the value -10.

4.3 Nice type trees

In this section, we introduce a special form of a type tree, called *nice type tree*. It is more restricted in its structure than general type trees. We show that an optimal type tree of this form exists for all type maps, thereby limiting the number of type trees that need to be considered drastically. The definition of a nice type tree (Defintion 12) may seem technical, but it closely matches what one would consider a natural representation. Some technicalities are required before we can formally introduce nice type trees.

Definition 11 (Shifting node)**.** *We call an index node* $\mathsf{idx}(c, \langle i_0, \ldots\rangle, \mathcal{S})$*, an index bucket node* $\mathsf{idxbuc}(c, s, \langle \ldots\rangle\langle i_0, \ldots\rangle, \mathcal{S})$ *or a struct node* $\mathsf{strc}(c, \langle i_0, \ldots\rangle, \langle \ldots\rangle)$ *with* $i_0 \neq 0$ *a* shifting node*;* $x = i_0$ *is called the node's* shift*.*

Adding some shift x to all displacements of an irregular node $\mathcal{T}$ *shifts* the represented type map by x. For example, assume a type tree $\mathsf{idx}(c, \langle i_0, \ldots, i_{c-1}\rangle, \mathcal{S})$ representing the type map $M = (\langle t_0, \ldots, t_{n-1}\rangle, \langle d_0, \ldots, d_{n-1}\rangle)$. Then, the type tree $\mathsf{idx}(c, \langle i_0 + x, \ldots, i_{c-1} + x\rangle)$ represents the type map $M + x = (\langle t_0, \ldots, t_{n-1}\rangle, \langle d_0 + x, \ldots, d_{n-1} + x\rangle)$, i.e., the same type map shifted by the value x. If a type path contains an irregular node $\mathcal{N}$ with count 1 and displacement value i_0, $\mathcal{N}$ shifts the represented type map by i_0. That is, if a type path $\mathcal{P}$ represents a type map M, $\mathcal{P} \setminus \mathcal{N}$ represents the type map $M - i_0$. As the following corollary shows, $\mathcal{N}$ can be moved within $\mathcal{P}$ without changing the represented type map. Figure 4.6 provides an illustration.

Corollary 3. *If an irregular node* $\mathcal{N}$ *with count 1 is the child of a* vec*,* idx *or* idxbuc *node* $\mathcal{O}$*,* $\mathcal{N}$ *and* $\mathcal{O}$ *can be swapped without changing the represented type map.*

Proof. Assume that a type tree $\mathcal{T}$ contains an idx node $\mathcal{N} = \mathsf{idx}(1, \langle i_0\rangle, \mathcal{S})$ with count 1. Assume further that $\mathcal{N}$ is the child node of a vec node $\mathcal{O}$, i.e., $\mathcal{O} = \mathsf{vec}(c, s, \mathsf{idx}(1, \langle i_0\rangle, \mathcal{S}))$ with some fixed count c and stride s. The type tree rooted at $\mathcal{O}$ represents a replication of $\mathcal{S}$ at relative displacements i_0, $i_0 + s$, …, $i_0 + (c-1)s$. Swapping $\mathcal{N}$ and $\mathcal{O}$ does not change the represented type sequence. The type tree $\mathsf{idx}(1, \langle i_0\rangle, \mathsf{vec}(c, s, \mathcal{S}))$ represents a

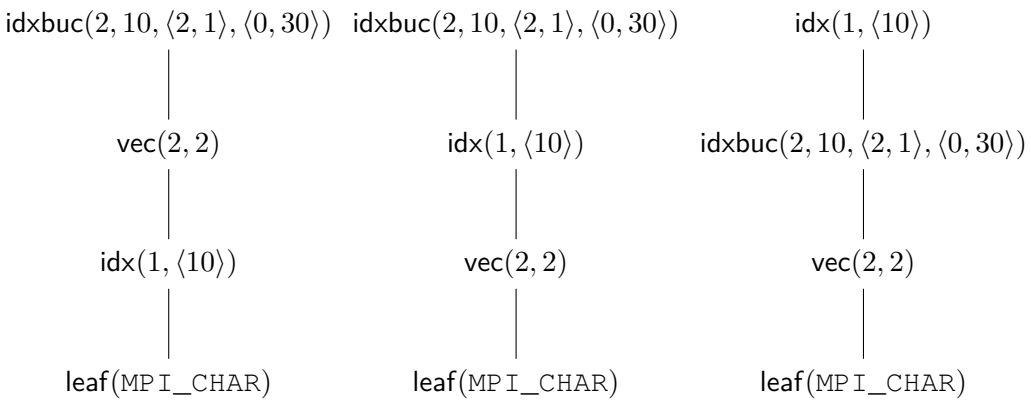


Figure 4.6: Three alternate type tree representations of same cost for the type map $(\langle \texttt{char}, \ldots \rangle, \langle 10, 12, 20, 22, 40, 42 \rangle)$.

replication of $\mathcal{S}$ at relative displacements $0 + i_0$, $s + i_0$, …, $(c-1)s + i_0$. The proof is analogous for any combination of $\mathcal{O}$ being a vec, idx or idxbuc node and $\mathcal{N}$ being an idx, idxbuc or strc node with count 1. □

Note that $\mathcal{N}$ cannot be moved across strc nodes (with count $c > 1$), as this would affect the type maps represented by the strc node's other sub-types as well, thereby changing the type map represented by the type tree.

Definition 12 (Nice type tree)**.** *A* nice type tree $\mathcal{T}$ *is a type tree that contains at most one shifting node, which, if it exists, is the first irregular node on every root to leaf path in* $\mathcal{T}$*.*

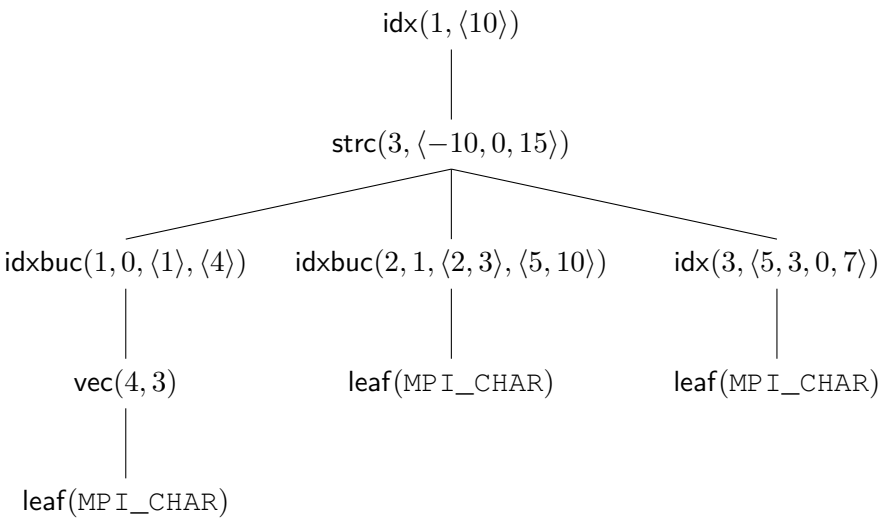
Figure 4.7 shows an example type tree and its nice type tree equivalent.

Lemma 1. *For any type tree* $\mathcal{T}$ *representing a type map* M*, a nice type tree representation* $\mathcal{T}'$ *of* M *with equal cost exists.*

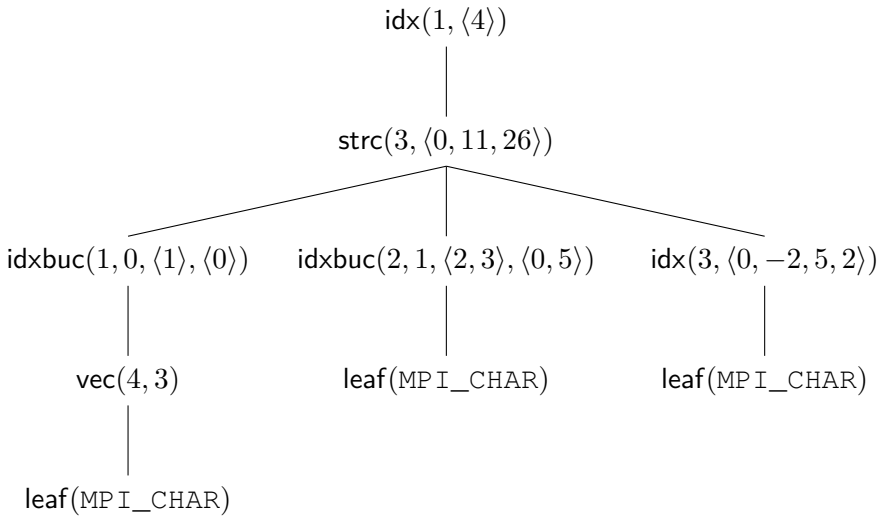
Proof. A node is *bad* if it is a shifting node and not the first irregular node on any root to leaf path in $\mathcal{T}$, i.e., if it violates the nice type tree property. If a node is not bad, it is *good*. A nice type tree $\mathcal{T}'$ representing the type map M can be constructed from the initial type tree $\mathcal{T}$ inductively as follows. Assume that a bad idx node $\mathcal{B} = \text{idx}(c, \langle i_0, \ldots, i_{c-1} \rangle, \mathcal{S})$ with shift $x = i_0$ is present in $\mathcal{T}'$ (the proof is analogous for bad idxbuc and bad strc nodes). On the path from $\mathcal{B}$ to the root node $\mathcal{R}$ there is either

1. a (good or bad) strc node $\mathcal{N}$ with no other strc, bad idx or bad idxbuc node on the path from $\mathcal{B}$ to $\mathcal{N}$; or

2. a bad idx or a bad idxbuc node $\mathcal{N}$ with no other strc, bad idx or bad idxbuc node on the path from $\mathcal{B}$ to $\mathcal{N}$; or



(a) In general, a type tree may contain many shifting nodes as well as many irregular nodes with count 1. This example type tree contains two irregular nodes with count 1 and a total of five shifting nodes.



(b) A nice type tree representation of equal cost. Note that the structure does not change, only the displacement values do. This tree contains only one shifting node, which is the first irregular node on any root to leaf path. All other nodes are aligned, i.e., their first displacement value is equal 0. The remaining displacement values have been adapted s.t. this type tree represents the same type map as the one in Figure 4.7a.

Figure 4.7: A general and a nice type tree representation for the type map ($\langle$`char`, ...$\rangle$, $\langle 4, 7, 10, 13, 15, 16, 20, 21, 22, 30, 28, 35, 32\rangle$).

3. a good idx or idxbuc node $\mathcal{N}$ s.t. there is no other idx, idxbuc or strc node on the path from $\mathcal{R}$ to $\mathcal{N}$ and cases 1 and 2 do not apply.

In the first two cases, we are interested in the bad idx, bad idxbuc or (good or bad) strc node that is closest to $\mathcal{B}$ on the path from $\mathcal{B}$ to $\mathcal{R}$; in the third case we are interested in a good idx or idxbuc node that is closest to $\mathcal{R}$. Note that for every bad node in a type tree, exactly one of the three cases applies. We show for each case how the bad node $\mathcal{B}$ can be transformed into a good one.

Case 1: On the path from $\mathcal{B}$ to $\mathcal{R}$ there is a strc node $\mathcal{N} = \mathsf{strc}(c', \langle i'_0, \ldots, i'_k, \ldots, i'_{c'-1}\rangle, \langle \mathcal{S}'_0, \ldots, \mathcal{S}'_k, \ldots \mathcal{S}'_{c-1}\rangle)$ with $\mathcal{B}$ being in the k-th subtree S'_k. We can change $\mathcal{B}$ to a non-shifting and thus good node by subtracting its shift value x from all displacements and adding x to the k-th displacement of the strc node $\mathcal{N}$. Formally, we set $\mathcal{B} = \mathsf{idx}(c, \langle 0, i_1 - x, \ldots, i_{c-1} - x\rangle, \mathcal{S})$ and $\mathcal{N} = \mathsf{strc}(c', \langle i'_0, \ldots, i'_{k-1}, i'_k + x, i'_{k+1}, \ldots, i'_{c'-1}\rangle, \langle \mathcal{S}'_0, \ldots, \mathcal{S}'_{c-1}\rangle)$.

1a: For $k = 0$, this may turn the strc node $\mathcal{N}$ from a good to a bad node and thus not decrease the total number of bad nodes in $\mathcal{T}'$. However, this can occur only finitely many times. The bad node always moves up the tree towards the root and the number of bad nodes is reduced by the following cases eventually.

1b: For $k > 0$, the status of $\mathcal{N}$ is not changed and thus we have one less bad node in $\mathcal{T}'$.

Case 2: On the path from $\mathcal{B}$ to $\mathcal{R}$ there is another bad idx node $\mathcal{N} = \mathsf{idx}(c', \langle i'_0, \ldots, i'_{c-1}, \mathcal{S}'\rangle$ (the proof is analogous for a bad idxbuc node $\mathcal{N} = \mathsf{idxbuc}(c', s', \langle b'_0, \ldots, b'_{c-1}\rangle, \langle i'_0, \ldots, i_{c-1}\rangle, \mathcal{S}')$). We change $\mathcal{B}$ to a non-shifting node by subtracting its shift value x from all displacements, as before. This is compensated by adding the same shift value to all displacements of $\mathcal{N}$, i.e., we set $\mathcal{N} = \mathsf{idx}(c', \langle i'_0 + x, \ldots, i'_{c-1} + x\rangle, \mathcal{S}')$. Since $\mathcal{N}$ was a bad node before, the total number of bad nodes in $\mathcal{T}'$ is reduced by one.

Case 3: There is no strc node and no other bad idx or idxbuc node on the path from $\mathcal{B}$ to $\mathcal{R}$. Since $\mathcal{B}$ is a bad node, there has to be at least one good idx or idxbuc node on the path from $\mathcal{B}$ to $\mathcal{R}$ (otherwise, $\mathcal{B}$ would be the first irregular node on a root to leaf path and thus not a bad node). Let $\mathcal{N} = \mathsf{idx}(c', \langle i' \ldots, i'_{c-1}\rangle, \mathcal{S}')$ be the first such node on the path from $\mathcal{R}$ to $\mathcal{B}$ (the proof is analogous for $\mathcal{N}$ being an idxbuc node). Subtracting $\mathcal{B}$'s shift x from its displacements and adding it to the displacements of $\mathcal{N}$ reduces the number of bad nodes in $\mathcal{T}'$ by one. The node $\mathcal{N}$ is the first irregular node on the path from $\mathcal{R}$ to $\mathcal{B}$ and thus the first such node on any root to leaf path. Adding the shift value x to its displacements does not turn $\mathcal{N}$ into a bad node.

A nice type tree $\mathcal{T}'$ can be obtained by applying the above transformations as long as a bad node exists in $\mathcal{T}'$. None of the transformations changes the represented type map, since a shift value subtracted from one node is always added at some other appropriate node. Only displacement values of irregular nodes are changed by these transformations, while the size of the displacement arrays stays the same and no nodes are added to or deleted from $\mathcal{T}'$. Thus, $\mathcal{T}'$ is a nice type tree representing M and of the same cost as the initial type tree $\mathcal{T}$. □

Observe that a nice type tree is not necessarily optimal. For example, the nice type tree shown in Figure 4.7b contains an idxbuc node with count 1 and $i_0 = 0$. This node is "useless" in the sense that it can be removed from the type tree without changing the represented type map. Doing so reduces the cost of the whole type tree and therefore the initial type tree, although nice, cannot be optimal. This intuition of *useless nodes* is formalized in the next corollary and we continue to prove several more properties of optimal type trees.

Corollary 4. *An optimal type tree does not contain a* vec *node with count 1 or an irregular node with count 1 and* $i_0 = 0$.

Proof. Assume a type tree $\mathcal{T} = \mathsf{vec}(1, s, \mathcal{S})$ that represents the type map M. Note that $\mathcal{S}$ represents the same type map with less cost and thus $\mathcal{T}$ cannot be an optimal representation of M. Due to the principle of optimality (Corollary 2), a vec node with count 1 can never be part of an optimal type tree representation. The proof for irregular nodes with count 1 and $i_0 = 0$ is analogous. □

Corollary 5. *Any optimal type tree* $\mathcal{T}$ *contains at most one irregular node with count 1. This node, if it exists, is a shifting node and the only irregular node in* $\mathcal{T}$.

Proof. Assume that $\mathcal{T}$ is optimal and contains more than one irregular node with count 1. The nice type tree property (Lemma 1) implies that a cost equivalent nice type tree exists which contains at most one shifting node (with arbitrary count). Any other irregular node is a non-shifting node with displacement array $\langle 0, \ldots \rangle$. Those irregular nodes with count 1 can be removed without changing the represented type map (Corollary 4). A representation of less cost exists, contradicting the premises that $\mathcal{T}$ is optimal. This proves that $\mathcal{T}$ contains at most one irregular node with count 1 and that this node is a shifting node. Call this node $\mathcal{N}$ and its shift x. It remains to show that $\mathcal{N}$ is the only irregular node in $\mathcal{T}$. Assume that $\mathcal{N}$ as well as at least one other irregular node are part of $\mathcal{T}$. Let $\mathcal{N}'$ be the first irregular node different from $\mathcal{N}$ on the path from $\mathcal{N}$ to the root node. If such a node does not exist, let $\mathcal{N}'$ be the first irregular node different from $\mathcal{N}$ on the path from $\mathcal{N}$ to a leaf node. Analogously to the proof of the nice type tree property (Lemma 1) we can set the first and only displacement value of $\mathcal{N}$ to 0 and add x to

- the displacement values of $\mathcal{N}'$ if it is an idx or idxbuc node; or
- the k-th displacement value i_k of $\mathcal{N}'$ if it is a strc node with $\mathcal{N}$ in the k-th subtree; or
- all the displacement values of $\mathcal{N}'$ if it is a strc node with $\mathcal{N}$ being an ancestor of $\mathcal{N}'$.

This does not change the represented type map but renders $\mathcal{N}$ useless. □

Corollary 6. *If an optimal type tree* $\mathcal{T}$ *representing a type map* M *contains an irregular node* $\mathcal{N}$ *with count 1,* M *can be represented by a cost-equivalent type tree* $\mathcal{T}'$ *s.t.* $\mathcal{N}$ *is the root node of* $\mathcal{T}'$.

Proof. Note that idx, idxbuc and strc nodes with count 1 all represent the same operation, i.e., for a given sub-type $\mathcal{S}$, displacement array $\langle i_0 \rangle$ and – where necessary – block lengths of $\langle 1 \rangle$, they all represent the same type map. We can therefore assume w.l.o.g. that $\mathcal{N}$ is an idx node. Due to Corollary 5, $\mathcal{N}$ is the only irregular node in $\mathcal{T}$, i.e., the only other nodes in $\mathcal{T}$ are leaf and vec nodes. In particular, $\mathcal{T}$ does not contain a strc node and we can apply Corollary 3 to move $\mathcal{N}$ to the top of the type tree. □

4.4 Optimal type trees for aligned type maps

Definition 13 (Aligned type map)**.** *A displacement sequence D is aligned if $D[0] = 0$. A type map $M = (T, D)$ is aligned if D is.*

Note that a type tree $\mathcal{T}$ not containing any shifting nodes always represents an aligned type map M, since it is not possible to generate a type map with first displacement value $D[0] \neq 0$ without a shifting node.

Corollary 7. *A nice type tree $\mathcal{T}$ for an aligned type map M does not contain any shifting nodes.*

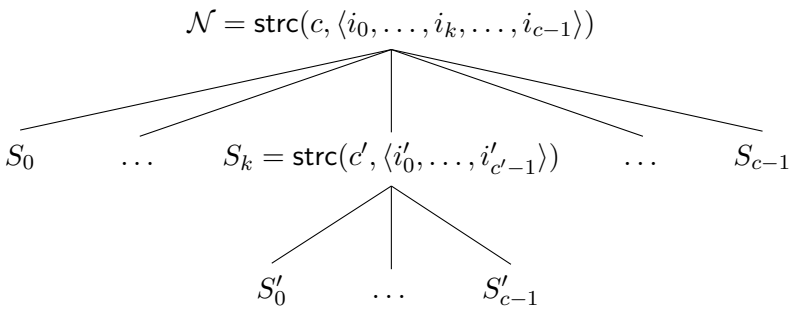
Proof. The nice type tree property (Definition 12) guarantees that $\mathcal{T}$ contains at most one shifting node, which is additionally the first irregular node on any root to leaf path. We proceed by contradiction: Assume that a shifting node $\mathcal{N} = \mathsf{strc}(c, \langle i_0, \ldots \rangle, \langle \mathcal{S}_0, \ldots, \mathcal{S}_{c-1} \rangle)$ indeed exists in $\mathcal{T}$ (the proof is analogous for idx and idxbuc nodes). Since $\mathcal{N}$ is the first irregular node on any root to leaf path, the type map N represented by $\mathcal{N}$ is a prefix of M, not a segment $M[i, j]$ with $i \neq 0$. In particular, $\mathcal{S}_0$, which does not contain any shifting nodes, represents an aligned prefix of M. Since $\mathcal{N}$ is a shifting node, it represents a prefix of M that is not aligned; a contradiction, since $\mathcal{N}$ cannot represent both an aligned and a non-aligned prefix of M. □

Given a type tree $\mathcal{T}$ without shifting nodes, it is clear that each subtree of $\mathcal{T}$ (including $\mathcal{T}$ itself) represents an aligned type map. If $\mathcal{T}$ represents the type map M, any subtree $\mathcal{S}$ represents an aligned segment of M.

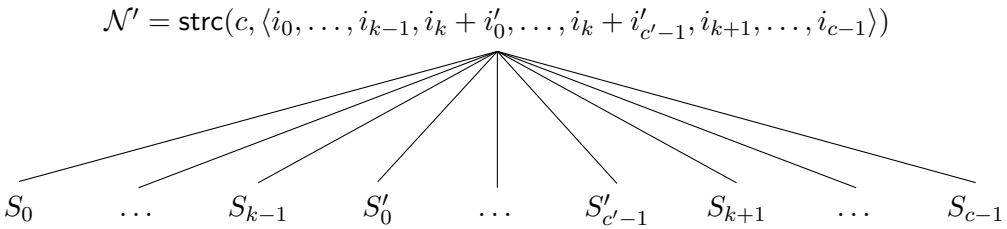
4.5 Height of an optimal type tree

In the following, we prove one more property of optimal type trees, namely that the height of an optimal type tree representation for a type map of length n is bounded by $O(\log n)$. This property is crucial for the construction of optimal type trees and optimal type paths for general type maps, i.e., the proofs of Lemma 8 (Section 5.6) and Lemma 13 (Section 6.4).

Corollary 8. *An optimal type tree $\mathcal{T}$ contains no directly nested* strc *nodes, i.e., no node of the form $\mathcal{N} = \mathsf{strc}(c, \langle i_0, \ldots, i_k, \ldots, i_{c-1} \rangle, \langle \mathcal{S}_0, \ldots, \mathcal{S}_k, \ldots, \mathcal{S}_{c-1} \rangle)$ with $S_k = \mathsf{strc}(c',$ $\langle i'_0, \ldots, i'_{c'-1} \rangle, \langle \mathcal{S}'_0, \ldots, \mathcal{S}'_{c'-1} \rangle)$ for some k, $0 \leq k < c$.*



(a) A representation with two directly nested strc nodes.



(b) A more concise representation with the two strc nodes merged into one.

Figure 4.8: Two directly nested strc nodes can always be merged into one.

Proof. The two directly nested strc nodes can be merged into a single node $\mathcal{N}' = \mathsf{strc}(c + c' - 1, \langle i_0, \ldots, i_{k-1}, i_k + i'_0, \ldots, i_k + i'_{c'-1}, i_{k+1}, \ldots, i_{c-1} \rangle, \langle \mathcal{S}_0, \ldots, \mathcal{S}_{k-1}, \mathcal{S}'_0, \ldots, \mathcal{S}'_{c'-1}, \mathcal{S}_{k+1} \ldots, \mathcal{S}_{c-1} \rangle)$. See Figure 4.8 for an illustration. The overall cost is reduced by the cost of the two nodes $\mathcal{N}$ and $\mathcal{S}_k$ and increased by the cost of $\mathcal{N}'$, i.e., by $K_{\mathsf{strc}} + cK_{\text{lookup}} + K_{\mathsf{strc}} + c'K_{\text{lookup}} - (K_{\mathsf{strc}} + (c + c' - 1)K_{\text{lookup}}) = K_{\mathsf{strc}} + K_{\text{lookup}}$. □

Lemma 2. *The height of an optimal type tree representing a type map of length n is $O(\log n)$.*

Proof. Note that a vec, idx or idxbuc node with count $c > 1$ and sub-type $\mathcal{S}$ increases the length of the represented type map by a factor of c, compared to the type map represented by $\mathcal{S}$. A strc node with count $c > 1$ on the other hand increases the length of the represented type map by only $c - 1$ in the worst case, compared to the largest type map represented by its sub-types. To see this, assume that one sub-type $\mathcal{S}$ represents a type map of length l, while the remaining $c - 1$ sub-types represent type maps of length 1 (i.e., they are simple leaf nodes). Concatenating these sub-types with a strc constructor results in a type tree representing a type map of length $l + c - 1$, i.e., a type map that is only $c - 1$ elements longer than the type map of length l represented by the sub-type $\mathcal{S}$.

Due to Corollary 5, an optimal type tree contains at most one node with count 1. Except for this node and the leaf node, at least every other node on any root to leaf path is a vec, idx or idxbuc node with count $c > 1$, since strc nodes cannot be directly nested (Corollary 8). Thus, on any root to leaf path the length of the represented type map at

least doubles with at least every other node, implying that the length of any root to leaf path in an optimal type tree is $O(\log n)$. □

CHAPTER 5

Type tree reconstruction

In this chapter, we present our algorithm that solves the Type Reconstruction Problem in polynomial time. More precisely, we prove the following theorem:

Theorem 1. *For any type map M of length n, the* Type Reconstruction Problem *can be solved in $O(n^4)$ time and $O(n^2)$ space.*

The observations regarding optimal type trees, nice type trees and aligned type maps made in the previous chapter (Sections 4.2 to 4.4) directly lead to an approach for attacking the Type Reconstruction Problem for the special case of aligned type maps. With the properties proved so far, we can give a high-level outline of our algorithm (Section 5.1). The missing parts are developed in Sections 5.2 to 5.4 and the full algorithm solving the problem for the special case of aligned type maps is given in Section 5.5. Section 5.6 shows how a solutions for general type maps can be constructed and concludes the chapter with the proof of Theorem 1.

5.1 Outline of the algorithm

We start with an outline of our algorithm for the Type Reconstruction Problem for aligned type maps. We draw from the pool of properties discussed in Chapter 4 to derive and justify an elegant and efficient algorithm. The missing parts, represented on a high level in Algorithm 3 by comments, are developed in the following sections.

Due to Lemma 1, it suffices to construct an optimal nice type tree. Furthermore, an optimal type tree for an aligned type map does not contain any shifting nodes (Corollary 7) or nodes with count 1 (Corollary 4). In other words, an optimal type tree for an aligned type map can be computed by an exhaustive search of all possible nice type tree representations that do not contain any shifting nodes or nodes with count 1. The problem exhibits optimal sub-structure (Corollary 2) and a dynamic programming algorithm can solve the problem by combining solutions of sub-problems. A *sub-problem*

Algorithm 3: A high level outline of our algorithm solving the TYPE RECONSTRUCTION PROBLEM for aligned type maps.

Input: Aligned type map $M = (T, D)$ of length n.
Output: Optimal type tree representation of M.

```
1  Function Typetree(M, n)
2      // Trivial solution for all segments of length 1
3      for i ← 0; i < n; i++ do
4          T_{i,i} ← leaf(T[i]);
5      // Construct optimal type tree for all aligned segments
           of M
6      for l ← 2; l ≤ n; l++ do
7          for i ← 0; i ≤ n − l; i++ do
8              j ← i + l − 1
9              // Let N_{i,j} be the aligned segment M[i, j]
10             // Construct for N_{i,j} the least-cost type tree...
11             // ...T_idx with an idx node as root
12             // ...T_idxbuc with an idxbuc node as root
13             // ...T_vec with a vec node as root
14             // ...T_strc with a strc node as root
15             // Optimal type tree T_{i,j} for N_{i,j} is one of T_idx, T_idxbuc,
                   T_vec or T_strc.
16     return T_{0,n−1}
```

of the TYPE RECONSTRUCTION PROBLEM for an aligned type map M of length n is to compute an optimal type tree representation for an aligned segment N of M, where N is of length strictly less than n. Given a type map $M = (T, D)$, we designate by $N_{i,j}$ the aligned segment $M[i, j]$, i.e., $N_{i,j} = M[i, j] - D[i]$. An optimal type tree representation for the sub-problem $N_{i,j}$ is designated by $\mathcal{T}_{i,j}$, which is also called a solution for the sub-problem.

Our approach is outlined in Algorithm 3 and proceeds in a bottom up manner: It iteratively constructs solutions for all sub-problems, i.e., for all aligned segments of the type map, starting with the segments of length 1. A sub-problem of length 1 is a pair of some base type t and the displacement 0. Formally, we have that $N_{i,i} = (\langle t_i \rangle, \langle 0 \rangle)$ for all i, $0 \leq i < n$. Such a type map can trivially be represented by a single leaf node $\mathsf{leaf}(t_i)$ and it is clear that this is the optimal representation. This trivial solution is constructed for each sub-problem of length 1 in lines 3 to 4 of Algorithm 3. The algorithm then proceeds to compute optimal representations for all sub-problems of length l, $l = 2, 3, \ldots, n$. Assume that a solution is known for all sub-problems of length strictly shorter than l, i.e., for all $N_{i,j}$ for which $j - i + 1 < l$, $0 \leq i \leq j < n$. A solution for a sub-problem $N_{i,i+l-1}$ is a type tree with either a vec, idx, idxbuc or strc node as the root node with the sub-types being solutions to some of the sub-problems $N_{i,j}$. For each

sub-problem, the algorithm constructs all possible nice type tree representations and picks the one of least cost. In other words, the solution is found by exhaustive search, with the search space restricted to nice type trees. The algorithm constructs up to four different representations for the sub-problem $N_{i,i+l-1}$:

- If the prefix $N_{i,i+q-1}$ of length q is repeated in $N_{i,i+l-1}$, a representation $\mathcal{T}_{\mathsf{idx}} = \mathsf{idx}(c, \langle i_0, \ldots, i_{c-1}\rangle, \mathcal{T}_{i,i+q-1})$, with $c = n/q$ is possible.
- With the same repeated prefix a representation $\mathcal{T}_{\mathsf{idxbuc}} = \mathsf{idxbuc}(c, s, \langle b_0, \ldots, b_{c-1}\rangle, \langle i_0, \ldots, i_{c-1}\rangle, \mathcal{T}_{i,i+q-1})$ is also possible.
- If the prefix of length q is strided in $N_{i,i+l-1}$, a representation $\mathcal{T}_{\mathsf{vec}} = \mathsf{vec}(c, s, \mathcal{T}_{i,i+q-1})$, with $c = n/q$ exists.
- A representation $\mathcal{T}_{\mathsf{strc}} = \mathsf{strc}(c, \langle i_0, \ldots, i_{c-1}\rangle, \langle \mathcal{T}_0, \ldots, \mathcal{T}_{c-1}\rangle)$, which always exists.

In Sections 5.2 to 5.4, we describe in detail how these representations can be constructed. Each of them is of least cost w.r.t. to all possible representations of that form. That is, no representation of $N_{i,i+l-1}$ exists that has an idx node as the root node and is of less cost than $\mathcal{T}_{\mathsf{idx}}$. Out of these four representations, the one with least cost is necessarily an optimal solution for the sub-problem $N_{i,i+l-1}$. The representation $\mathcal{T}_{\mathsf{strc}}$ is guaranteed to exist due to Corollary 1. The representations $\mathcal{T}_{\mathsf{idx}}$ and $\mathcal{T}_{\mathsf{idxbuc}}$, however, exist only if $N_{i,i+l-1}$ contains a repeated prefix. The representation $\mathcal{T}_{\mathsf{vec}}$ exists only if $N_{i,i+l-1}$ contains a strided prefix. As discussed in Section 4.1, the prefix of length l is trivially repeated and strided in any type map of length l. However, using this prefix for a representation via an idx, idxbuc, vec or strc node creates a node with count 1, which cannot exist in an optimal type tree representation. The solution for the sub-problem $N_{0,n-1}$ is the required solution for M, since $N_{0,n-1} = M$ for aligned type maps.

5.2 Representations via the vector and index constructors

In this section, we present straight-forward procedures to construct the representations $\mathcal{T}_{\mathsf{idx}}$ and $\mathcal{T}_{\mathsf{vec}}$. Formally, we prove the following lemma.

Lemma 3. *Let $M = (T, D)$ be an aligned type map of length n and assume that optimal type tree representations $\mathcal{T}_{0,j}$ are known for all sub-problems of length strictly less than n, i.e., for each sub-problem $N_{0,j}$ with $j < n - 1$. The representations $\mathcal{T}_{\mathsf{idx}}$ and $\mathcal{T}_{\mathsf{vec}}$, s.t. each of them is of least cost w.r.t. all possible representations of the respective form, can be computed in $O(n\sqrt{n})$ time.*

The algorithms computing the representations $\mathcal{T}_{\mathsf{idx}}$ and $\mathcal{T}_{\mathsf{vec}}$ employ two simple procedures (given in Algorithm 4) which check in a straight-forward way if a prefix is repeated and strided. It is easy to see that both procedures require $O(n)$ time: The inner for-loop in procedure `Repeated` executes q times for each of the n over q iterations of the outer for-loop, leading to a total of $O(n)$ executions of the nested for-loop's body. The body of the third for-loop is executed at most $n - q$ times, implying an $O(n)$ worst case

Algorithm 4: Trivial checks for repetitions and strided repetitions.

```
   Input: Type map M = (T, D) of length n; integer q s.t. q is a divisor of n.
   Output: true, if the prefix M_q is repeated in M; false otherwise.
1  Function Repeated(M, n, q)
2      for i ← q; i < n; i ← i + q do
3          for j ← 1; j < q; j ← j + 1 do
4              if D[j] − D[0] ≠ D[i + j] − D[i] then
5                  return false
6      for i ← q; i < n; i++ do
7          if T[i] ≠ T[i mod q] then
8              return false
9      return true
10
   Input: Type map M = (T, D) of length n; integer q s.t. M_q is a repeated prefix
   Output: true, if the prefix M_q is strided in M; false otherwise
11 Function Strided(M, n, q)
12     s ← D[q] − D[0]
13     for i ← q; i < n; i ← i + q do
14         if D[i] − D[i − q] ≠ s then
15             return false
16     return true
```

bound for the whole procedure. The for-loop in `Strided` performs n over q iterations, directly implying the claimed upper bound.

Furthermore, Algorithms 5 and 6 use the function `Min`. Given two type trees $\mathcal{T}$ and $\mathcal{S}$, `Min` returns the one with least cost. If either is `null`, the other is returned. Since we keep with each type node the cost of the type tree rooted at this node (as discussed in Section 2.6), `Min` can easily be implemented in constant time.

Algorithms 5 and 6 enumerate all possible representation of the desired forms and keep track of the ones with least cost. A representation with an idx node as root node can be constructed for each repeated prefix M_q of length q. Algorithm 5 simply checks for each divisor q of n whether the prefix M_q is repeated in M. If a prefix of length q is repeated in M, M can be represented with an idx node with count $c = n/q$ and the sub-type $\mathcal{T}_{0,q-1}$. Since both the prefix of length q and the type map M are aligned, we have that $N_{0,q-1} = M_q$. Thus, the displacement values are easily derived as $i_0 = D[0]$, $i_1 = D[q]$, …, $i_{c-1} = D[(c-1)q]$. If this prefix is additionally strided, a representation via a vec node with stride $s = D[q] - D[0]$ is also possible.

The foreach-loop (line 3 in both Algorithm 5 and Algorithm 6) is executed for each divisor of n, except for n itself. For $q = 1$, the sub-type is a leaf node, whereas $q = n$

Algorithm 5: Finding a least cost representation with an idx node as root node.

Input: Type map $M = (T, D)$ of length n; optimal type trees $\mathcal{T}_{0,j}$ for each sub-problem $N_{0,j}$ with $j < n - 1$.

Output: Representation $\mathcal{T}_{\mathsf{idx}}$ of M, which is of least cost w.r.t. all possible representations of this form.

```
1  Function Idx(M, n, {T_{0,j} | 0 ≤ j < n − 1})
2      T_idx ←null
3      foreach divisor q of n, q < n do
4          if Repeated(M, n, q) then
5              c ← n/q
6              for i = 0; i < c; i++ do
7                  disp[i] ← D[iq]
8              S ← idx(c, disp, T_{0,q−1})
9              S.cost ← K_idx + cK_lookup + T_{0,q−1}.cost
10             T_idx ← Min(T_idx, S)
11     return T_idx
```

Algorithm 6: Finding a least cost representation with a vec node as root node.

Input: Aligned type map $M = (T, D)$ of length n; optimal type trees $\mathcal{T}_{0,j}$ for each sub-problem $N_{0,j}$ with $j < n - 1$.

Output: Representation $\mathcal{T}_{\mathsf{vec}}$ of M, which is of least cost w.r.t. all possible representations of this form.

```
1  Function Vec(M, n, {T_{0,j} | 0 ≤ j < n − 1})
2      T_vec ←null
3      foreach divisor q of n, q < n do
4          if Repeated(M, n, q) and Strided(M, n, q) then
5              c ← n/q
6              s ← D[q] − D[0]
7              S ← vec(c, s, T_{0,q−1})
8              S.cost ← K_vec + T_{0,q−1}.cost
9              T_vec ← Min(T_vec, S)
10     return T_vec
```

would lead to nodes with count 1. Since M is aligned, we have that $D[0] = 0$ and the constructed nodes do not violate the nice type tree property. We stress that the type map must be aligned for Algorithm 6 to work correctly, whereas Algorithm 5 constructs a valid representation for general type maps.

Proof of Lemma 3. The correctness of Algorithm 5 and Algorithm 6 follows from the above observations. The body of the foreach-loop of Algorithm 5 requires $O(n)$ time: It contains one call of `Repeated` and assembling the array of displacements requires c steps. The remaining steps are each feasible in constant time. Since all these costs are additive, $O(n)$ is an upper bound for the loop-body, which is executed once for each divisor q of n (except for n itself).

A rough upper bound for the number of divisors of n is $O(\sqrt{n})$: Each divisor of n smaller than or equal to $\sqrt{n}$ is paired with a divisor larger than or equal to $\sqrt{n}$. We can count the number of divisors smaller than $\sqrt{n}$ and multiply it by two to get the total number of divisors. Since at most $\sqrt{n}$ values divide $\sqrt{n}$, the total number of divisors of n is at most $2\sqrt{n}$, which is $O(\sqrt{n})$. We note that this rough upper bound suffices for the purposes of this section. In Chapter 6, a more efficient algorithm is presented that leads to a tighter bound, which may be beneficial for implementations.

In total, Algorithm 5 performs $O(n\sqrt{n})$ steps. The same bound holds for Algorithm 6: Checking whether a prefix is repeated and strided requires $O(n)$ time. The upper bound for the number of divisors directly proves the claim. □

5.3 Representation via the index bucket constructor

For the construction of the representation $\mathcal{T}_{\mathsf{idxbuc}}$, repeated prefixes are also required. The representations $\mathcal{T}_{\mathsf{idx}}$ and $\mathcal{T}_{\mathsf{idxbuc}}$ are closely related: If a representation with an idx node as root node exists, so does a representation where the root node is an idxbuc node. Nevertheless, computing the representation $\mathcal{T}_{\mathsf{idxbuc}}$ is more involved than the construction of the $\mathcal{T}_{vct}$ and $\mathcal{T}_{\mathsf{idx}}$ representations.

Lemma 4. *Let $M = (T, D)$ be an aligned type map of length n and assume that optimal type tree representations $\mathcal{T}_{0,j}$ are known for all sub-problems of length strictly less than n, i.e., for each sub-problem $N_{0,j}$ with $j < n - 1$. The representation $\mathcal{T}_{\mathsf{idxbuc}}$ of least cost w.r.t. all possible representations of this form can be computed in $O(n\sqrt{n}\log n)$ time.*

As discussed in Section 4.1, a representation via an idxbuc node can always be derived from a representation with an idx node. In particular, if $\mathsf{idx}(c, \langle i_0, i_1, \ldots, i_{c-1}\rangle, \mathcal{T}_{0,q-1})$ represents M, so does the type tree $\mathsf{idxbuc}(c, s, \langle 1, \ldots, 1\rangle, \langle i_0, i_1, \ldots, i_{c-1}\rangle, \mathcal{T}_{0,q-1})$ with arbitrary bucket stride s. Although the latter is clearly more redundant, it is not necessarily a more costly representation of M. Using a bucket stride $s = D[q] - D[0]$ allows for the two s-strided segments $D[0, q-1]$, $D[q, 2q-1]$ to be joined into one bucket, i.e., the following representation is possible: $\mathsf{idxbuc}(c-1, s, \langle 2, \ldots, 1\rangle, \langle i_0, i_2, \ldots, i_{c-1}\rangle, \mathcal{T}_{0,q-1})$. Note that the number of buckets was reduced by one and that the cost of an idxbuc node depends on the number but not the size of the buckets. It is clear that the

Algorithm 7: Finding a least cost representation with an idxbuc node as root node.

Input: Type map $M = (T, D)$ of length n; optimal type trees $\mathcal{T}_{0,j}$ for each sub-problem $N_{0,j}$ with $j < n - 1$.
Output: Representation $\mathcal{T}_{\mathsf{idxbuc}}$ of M, which of least cost w.r.t. all possible representations of this form.

```
1  Function Idxbuc(M, n, {T_{0,j} | 0 ≤ j < n − 1})
2      T_idxbuc ← null
3      foreach divisor q of n, q < n do
4          if Repeated(M, n, q) then
5              E ← {D[iq] − D[(i − 1)q] | 1 ≤ i < n/q}
6              sort E
7              s ← most frequently occurring element in E
8              disp[0] ← D[0]
9              buckets[0] ← 1
10             j ← 0
11             for i ← 1; i < n/q; i++ do
12                 if D[iq] − D[(i − 1)q] = s then
13                     buckets[j] ← buckets[j] + 1
14                 else
15                     j ← j + 1
16                     disp[j] ← D[iq]
17                     buckets[j] ← 1
18             S ← idxbuc(j + 1, s, buckets, disp, T_{0,q−1})
19             S.cost ← K_idxbuc + 2cK_lookup + T_{0,q−1}.cost
20             T_idxbuc ← Min(T_idxbuc, S)
21     return T_idxbuc
```

least-cost representation $\mathcal{T}_{\mathsf{idxbuc}}$ joins as many segments as possible into buckets. This can be done for a given type map M and repeated prefix M_q by computing the offsets between any two consecutive segments of the displacement sequence and setting the bucket stride s to the value that occurs most frequently. Algorithm 7 does this in a straight-forward way. It generates the set E of offsets between each two consecutive segments, i.e., the set of values $D[iq] - D[(i-1)q]$ for all i, $1 \leq i < n/q$. After sorting the set E, a single scan suffices to find the most frequent value. In the next step, the bucket sizes and displacements are computed for the given bucket stride s. Two consecutive segments are joined into one bucket if their stride is equal to s. In this case, the appropriate bucket size is increased by one. Otherwise, the next displacement value is set to the segment's displacement and the next bucket size value is initialized. Note that each bucket is at least of size one and that its size is increased for every two consecutive segments with an offset of s.

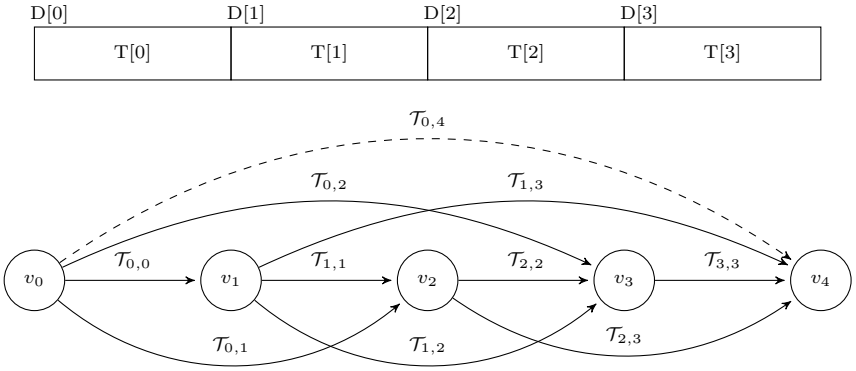


Figure 5.1: A generic type map of length 4 and its mapping to a DAG, used for finding a least-cost representation via a strc node. An edge (v_i, v_{j+1}) corresponds to the sub-problem $N_{i,j}$ and its optimal type tree representation $\mathcal{T}_{i,j}$. By construction, the edge (v_0, v_4) does not exist in the graph. It can be thought of as corresponding to the whole type map or equally, the sub-problem $N_{0,3}$), for which the desired representation can be computed by finding a shortest path in the graph.

Again, it is clear that the constructed type tree adheres to the nice type tree property. The algorithm in line 8 sets i_0 to $D[0]$, which is equal to 0 for aligned type maps.

Proof of Lemma 4. By the same argument as used in the proof of Lemma 3, the for-loop is executed $O(\sqrt{n})$ times. Sorting the set E, which is of size $O(n)$, requires $O(n \log n)$ time, while all the other steps in the loop body are trivially feasible in $O(n)$ time. In total, we have that the algorithm performs $O(n\sqrt{n} \log n)$ steps. □

5.4 Representation via the struct constructor

The algorithms for computing the representations $\mathcal{T}_{idx}$, $\mathcal{T}_{\mathsf{idxbuc}}$ and $\mathcal{T}_{\mathsf{vec}}$ are based on repeated prefixes. The strc constructor does not require a repeated prefix to be applicable and thus a fundamentally different approach is required to construct the representation $\mathcal{T}_{\mathsf{strc}}$.

Lemma 5. *Let M be an aligned type map of length n and assume that optimal type tree representations $\mathcal{T}_{i,j}$ are known for all sub-problem $N_{i,j}$ of length strictly less than n, i.e., for each sub-problem $N_{i,j}$ with $0 \leq i \leq j < n$, $j - i + 1 < n$. The representation $\mathcal{T}_{\mathsf{strc}}$ of least cost w.r.t. all possible representations of this form can be computed in $O(n^2)$ time.*

Proof. Construct a weighted, directed acyclic graph (DAG) $G = (V, E, w)$ with

$$V = \{v_0, \ldots, v_n\}$$
$$E = \{(v_i, v_j) \mid 0 \leq i < j \leq n,\ j - i < n\}$$
$$\forall (v_i, v_j) \in E:\ w(v_i, v_j) = 2K_{\text{lookup}} + \text{cost}(\mathcal{T}_{i,j-1}) \quad .$$

Algorithm 8: Finding a least cost representation with a strc node as the root node.

Input: Aligned type map $M = (T, D)$ of length n; optimal type trees $\mathcal{T}_{i,j}$ for each sub-problem $N_{i,j}$ with $0 \leq i \leq j < n$, $j - i < n - 1$.

Output: Representation $\mathcal{T}_{\mathsf{strc}}$ of M, which is of least cost w.r.t. all possible representations of this form.

Function `Struc(`M, n, $\{\mathcal{T}_{i,j} \mid 0 \leq i \leq j < n,\ j - i < n - 1\}$`)`

- `// Construct the graph` G
- $G \leftarrow (V \leftarrow \{v_0, \ldots, v_n\}, E \leftarrow \emptyset, w)$
- **for** $i \leftarrow 0$; $i < n$; $i{+}{+}$ **do**
  - **for** $j \leftarrow i$; $j < n$; $j{+}{+}$ **do**
    - $E \leftarrow E \cup (v_i, v_{j+1})$
    - $w(v_i, v_{j+1}) \leftarrow 2K_{\text{lookup}} + \mathcal{T}_{i,j}.\mathit{cost}$
- $P \leftarrow$ shortest path from v_0 to v_n in G
- Assume $P = \langle v_0, u_1, \ldots, u_k, v_n \rangle$ with $u_i \in V$, $1 \leq i \leq k < n$
- $P.\mathit{cost} = w(v_0, u_1) + w(u_1, u_2) + \cdots + w(u_k, v_n)$
- `// Map` P `to a representation via a strc node`
- $\mathit{disp} \leftarrow \langle D[v_0], D[u_1], \ldots, D[u_k] \rangle$
- $\mathit{subtypes} \leftarrow \langle \mathcal{T}_{v_0,u_1-1}, \mathcal{T}_{u_1,u_2-1}, \ldots, \mathcal{T}_{u_k,n-1} \rangle$
- $\mathcal{T}_{\mathsf{strc}} \leftarrow \mathsf{strc}(k+1, \mathit{disp}, \mathit{subtypes})$
- $\mathcal{T}_{\mathsf{strc}}.\mathit{cost} \leftarrow K_{\mathsf{strc}} + P.\mathit{cost}$
- **return** $\mathcal{T}_{\mathsf{strc}}$

In Figure 5.1 the construction is exemplified for a type map of length 4. It is used in Algorithm 8 to compute the representation $\mathcal{T}_{\mathsf{strc}}$. The intended meaning of G is as follows. The graph contains $n + 1$ vertices for a type map of length n, where a vertex v_i with $0 \leq i < n$ corresponds to the i-th base type – displacement pair (t_i, d_i) of M. The vertex v_n corresponds to the hypothetical first element after the end of M. An edge is associated with both a sub-problem and its solution. In particular, the edge (v_i, v_{j+1}) corresponds to the sub-problem $N_{i,j+1}$, with the aligned type map segment starting at index i and ending right before index $j + 1$, i.e., at index j. The weight $w(v_i, v_{j+1})$ of an edge (v_i, v_{j+1}) is equal to the cost including the associated representation $\mathcal{T}_{i,j}$ for $N_{i,j}$ as a sub-type in a strc node, i.e., to the cost of $\mathcal{T}_{i,j}$ plus the lookup costs for the displacement and sub-type arrays. By assumption, the solution $\mathcal{T}_{i,j}$ for the sub-problem $N_{i,j}$ already exists, since the length of the associated sub-problem is less than n. The edge (v_0, v_n), which is not part of the constructed graph, can be thought of as corresponding to the type tree $\mathcal{T}_{0,n-1}$, i.e., the optimal type tree representation for M that we want to compute. The number of edges in the resulting graph is

$$\binom{n+1}{2} - 1 = \frac{(n+1)n}{2} - 1 = O(n^2) \quad ,$$

since there is an edge between each node pair except for (v_0, v_n).

Next, we show a direct correspondence between paths from v_0 to v_n in the graph G and representations via the strc constructor. Informally, each edge in a path $v_0 \rightsquigarrow v_n$ in G corresponds to an aligned segment of M being used as a sub-type. Since G is a DAG by construction, any path in G is necessarily a simple path, i.e., it does not contain any loops. Additionally, because of the direct correspondence of edges and sub-problems, a path in G covers all of M: Each vertex v_i on a path, except for v_0 and v_n, has exactly one ingoing and one outgoing edge, implying the segment ending at index $i-1$ is followed by a segment starting at index i.

Let $P = \langle v_0, u_1, \ldots, u_k, v_n \rangle$ be a path in G from v_0 to v_n with $u_i \in V$ for $1 \leq i \leq k$. Note that P consists of $k+1$ edges and that any such path P consists of at least two and at most n edges: Since G does not contain the edge (v_0, v_n), a path from v_0 to v_n containing only one edge cannot exist. This implies that for any edge (v_i, v_j) in P, $j-i$ is at most $n-1$. Thus, the sub-problems represented by the edges in a path P are all of length strictly less than n. Since P cannot contain any loops, each of the $n+1$ vertices can occur at most once, implying that there are at most n edges in P.

If $P = \langle v_0, u_1, u_2, \ldots, u_k, v_n \rangle$ is a path from v_0 to v_n in G, the type tree $\mathcal{T}_{\mathsf{strc}} = \mathsf{strc}(k+1, \langle D[0], D[u_1], \ldots, D[u_k] \rangle, \langle \mathcal{T}_{0,u_1-1}, \mathcal{T}_{u_1,u_2-1}, \ldots, \mathcal{T}_{u_k,n-1} \rangle)$ is a valid representation of M. The total cost of a type tree rooted at a strc node $\mathsf{strc}(c, \langle \ldots \rangle, \langle \mathcal{S}_0, \ldots, \mathcal{S}_{c-1} \rangle)$ is equal to

$$K_{\mathsf{strc}} + 2cK_{\text{lookup}} + \sum_{i=0}^{i<c} \mathcal{S}_i.cost = K_{\mathsf{strc}} + \sum_{i=0}^{i<c} (2K_{\text{lookup}} + \mathcal{S}_i.cost) \quad .$$

Thus, the cost of the representation $\mathcal{T}_{\mathsf{strc}}$ is

$$\begin{aligned} K_{\mathsf{strc}} &+ 2cK_{\text{lookup}} + \mathcal{T}_{0,u_1-1}.cost + \mathcal{T}_{u_1,u_2-1}.cost + \cdots + \mathcal{T}_{u_k,n-1}.cost = \\ &= K_{\mathsf{strc}} + w(v_0, u_1) + w(u_1, u_2) + \cdots + w(u_k, v_n) = \\ &= K_{\mathsf{strc}} + P.cost \quad . \end{aligned}$$

Since M is an aligned type map, we have that $D[0] = 0$ and therefore the constructed node adheres to the nice type tree property. By construction, for any valid representation of M of the desired form, a corresponding path from v_0 to v_n exists in G. Since the weight of the edges is equal to the cost of including the corresponding sub-problems as sub-types, a shortest path represents a least-cost solution of the desired form.

The overall runtime of this construction as shown in Algorithm 8 is as follows. The construction of the DAG G requires $O(n^2)$ time, due to the two nested for-loops. A shortest path in G can be found in $O(|V| + |E|)$ time (Cormen et al. [CLRS09, p. 655]). The number of edges $|E|$ in G is $O(n^2)$ and a shortest path P can thus be found in $O(n^2)$ time. Given P, the desired representation can be constructed in linear time, because optimal representations for all required sub-problems are known. □

5.5 Full algorithm for aligned type maps

We can now give the full algorithm for constructing optimal type trees for aligned type maps. The algorithm completes the outline given in Algorithm 3, using the observations

and algorithms discussed in Sections 5.2 and 5.4 to fill in the missing parts.

Lemma 6. *For any aligned type map M of length n, the* Type Reconstruction Problem *can be solved in $O(n^4)$ time.*

Algorithm 9: Type tree reconstruction algorithm for aligned type maps.

Input: Aligned type map $M = (T, D)$ of length n.
Output: DAG G, with optimal type tree representations $\mathcal{T}_{i,j}$ for each sub-problem $N_{i,j}$ of M associated with edge (v_i, v_{j+1}), $0 \leq i \leq j < n$.

```
1  Function Typetree(M , n)
2      G ← (V ← {v_0, ..., v_n}, E ← ∅, w)
3      // Solutions for all sub-problems of length 1
4      for i ← 0; i < n; i++ do
5          T_{i,i} ← leaf(T[i])
6          T_{i,i}.cost ← K_leaf
7          E ← E ∪ (v_i, v_{i+1})
8          w(v_i, v_{i+1}) ← 2K_lookup + T_{i,i}.cost
9      // Find solutions for all sub-problems
10     for l ← 2; l ≤ n; l++ do
11         for i ← 0; i ≤ n − l; i++ do
12             j ← i + l − 1
13             // Get sub-problem N_{i,j}
14             N_{i,j} = (T_S ← ⟨⟩, D_S ← ⟨⟩)
15             for k ← 0; k < l; k++ do
16                 T_S[k] ← T[i + k]
17                 D_S[k] ← D[i + k] − D_S[i]
18             // Representations via vec, idx and idxbuc nodes
19             T_vec ← Vec(N_{i,j}, l, {T_{i,k} | i ≤ k < i + l}, i)
20             T_idx ← Idx(N_{i,j}, l, {T_{i,k} | i ≤ k < i + l}, i)
21             T_idxbuc ← Idxbuc(N_{i,j}, l, {T_{i,k} | i ≤ k < i + l}, i)
22             // Representation via strc node
23             T_strc ← Struc(M, G, i, j + 1)
24             // Solution for the sub-problem N_{i,j}
25             T_{i,j} ←Min(T_idx,T_idxbuc,T_vec,T_strc)
26             // Update G
27             E ← E ∪ (v_i, v_{j+1})
28             w(v_i, v_{j+1}) ← 2K_lookup + T_{i,j}.cost
29     return G // Solution for M is stored with edge (v_0, v_n)
```

The input to the dynamic programming algorithm given in Algorithm 9 is an aligned type map M of length n. The algorithm constructs an optimal type tree $\mathcal{T}_{i,j}$ for each

sub-problem $N_{i,j}$. It can be viewed as filling out a dynamic programming table, which consists of n rows and n columns. A sub-problem $N_{i,j}$ corresponds to the table entry at row i and column j, i.e., the aligned segment of M starting at index i and ending at index j. This dynamic programming table is implicit in the constructed DAG G. To compute the representations $\mathcal{T}_{\mathsf{idx}}$, $\mathcal{T}_{\mathsf{idxbuc}}$ and $\mathcal{T}_{\mathsf{vec}}$, small, technical modifications are required (Algorithm 5, Algorithm 6 and Algorithm 7 respectively). For the computation of the representation $\mathcal{T}_{\mathsf{strc}}$, we split the generation of the graph from the remaining steps of finding a shortest path and mapping it to a strc node. These modifications do not change the overall semantics of the algorithms and we we address them in detail in the following.

In the first step of Algorithm 9, the graph G is initialized with $n+1$ vertices and an empty edge set. A preprocessing step to construct optimal type tree representations for all sub-problems of length 1 follows. As discussed in Section 5.1, the optimal solution for such a sub-problem is always a single leaf node and we therefore set $\mathcal{T}_{i,i} = \mathsf{leaf}(T[i])$ for all i, $0 \leq i < n$. The cost of this representation is trivially equal to K_{leaf}. We add an edge (v_i, v_{i+1}) for each sub-problem $N_{i,i}$ to G and set its cost to $2K_{\text{lookup}} + \mathcal{T}_{i,i}.cost$, which is the cost of including $\mathcal{T}_{i,i}$ as a sub-type in a representation via a strc node.

The algorithm proceeds to construct optimal representations for longer and longer sub-problems. Note that the preprocessing step constructed solutions for all sub-problems of length 1 and that therefore Lemma 3, Lemma 4 and Lemma 5 are applicable for each sub-problem of length 2. By incrementally computing optimal representations for all sub-problems of length $l = 2, \ldots, n$, it is guaranteed that solutions for all sub-problems of length less than l are known when a solution for a sub-problem of length l is to be computed. This ensures that the three lemmas can be applied to compute an optimal representation for each sub-problem as follows. We denote the type and displacement sequence of a sub-problem $N_{i,j}$ as T_S and D_S respectively. After extracting the sub-problem (lines 14 – 17), we can use Algorithm 5, Algorithm 6 and Algorithm 7 to construct the representations $\mathcal{T}_{\mathsf{idx}}$, $\mathcal{T}_{\mathsf{vec}}$ and $\mathcal{T}_{\mathsf{idxbuc}}$ for $N_{i,j}$.

Note that these algorithms require access to optimal representations of the prefixes of the sub-problem. Since a sub-problem $N_{i,j}$ is a segment, but not necessarily a prefix of M, its prefixes start at index i of the input type map M. To account for this and avoid copying the type tree representations for prefixes of $N_{i,j}$, we pass an additional argument o representing the offset of the sub-problem within the type map M. Thus, for a sub-problem $N_{i,j}$ we use $o = i$ and replace all occurrences of $\mathcal{T}_{0,q-1}$ with $\mathcal{T}_{o,o+q-1}$ (lines 1, 8 and 9 in Algorithm 5, lines 1, 7 and 8 in Algorithm 6 as well as lines 1, 18 and 19 in Algorithm 7).

Algorithm 8, which computes the representation $\mathcal{T}_{\mathsf{strc}}$, is not directly used as a subroutine. Instead of constructing a DAG for each sub-problem, the algorithm is changed so that it can compute the representation $\mathcal{T}_{\mathsf{strc}}$ for all sub-problems using only a single, incrementally built graph. Instead of mapping the type map to a DAG, Algorithm 10 takes the DAG as input together with a start and target vertex. When computing the solution for a sub-problem $N_{i,j}$, the sub-graph induced by the vertex set $\{v_i, \ldots, v_{j+1}\}$ of G is equivalent to the graph constructed by Algorithm 8. Formally,

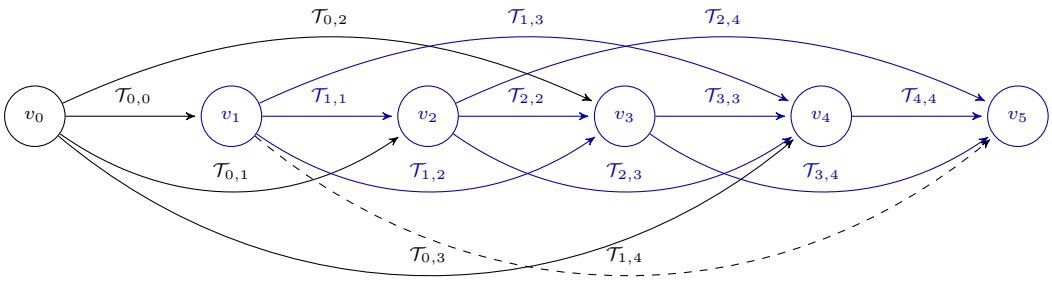


Figure 5.2: The graph G when computing the representation $\mathcal{T}_{\mathsf{strc}}$ for the sub-problem $N_{1,4}$ of a type map of length 5. Solutions for all sub-problems $N_{i,j}$ with $0 \leq i \leq j \leq 5$, $j - i \leq 3$ as well as for $N_{0,3}$ are already known and thus the corresponding edges (v_i, v_{j+1}) exist in G. To compute the representation $\mathcal{T}_{\mathsf{strc}}$ for the sub-problem $N_{1,4}$ (which corresponds to the dashed edge (v_1, v_5)), a shortest path from v_1 to v_5 has to be found in the sub-graph induced by the vertex set $\{v_1, \ldots, v_5\}$, which is highlighted in blue.

Algorithm 10: Finding least cost representations with a strc node as the root node.

Input: Aligned type map $M = (T, D)$ of length n; DAG mapping G for M; indices s, t of start and target node in G.

Output: Representation $\mathcal{T}_{\mathsf{strc}}$ for the sub-problem $N_{s,t-1}$; $\mathcal{T}_{\mathsf{strc}}$ is of least cost w.r.t. all possible representations of this form.

Function `Struc(`M, G, s, t`)`
- $P \leftarrow$ shortest path from v_s to v_t in G
- Assume $P = \langle v_s, u_1, \ldots, u_k, v_t \rangle$ with $u_i \in V$, $1 \leq i \leq k < t - s$
- $P.cost = w(v_s, u_1) + w(u_1, u_2) + \cdots + w(u_k, v_t)$
- `// Map` P `to a representation via a` strc `node`
- $disp \leftarrow \langle D[v_s], D[u_1], \ldots, D[u_k] \rangle$
- $subtypes \leftarrow \langle \mathcal{T}_{v_s, u_1 - 1}, \mathcal{T}_{u_1, u_2 - 1}, \ldots, \mathcal{T}_{u_k, n} \rangle$
- $\mathcal{T}_{\mathsf{strc}} \leftarrow \mathsf{strc}(k + 1, disp, subtypes)$
- $\mathcal{T}_{\mathsf{strc}}.cost \leftarrow K_{\mathsf{strc}} + P.cost$
- **return** $\mathcal{T}_{\mathsf{strc}}$

given a graph $G = (V, E)$ and a subset $V' \subset V$, the sub-graph *induced* by V' is the graph $G' = (V', E')$, where $E' = \{(u, v \in E \mid u, v \in V')\}$ (see, e.g., [CLRS09, p. 1171]). By construction, when computing the desired representation for a sub-problem $N_{i,j}$ of length $l = j - i + 1$, G contains edges that correspond to optimal type trees for all sub-problems of length less than l. That is, for each edge (v_i, v_{j+1}) with $j - i < l$, an optimal type tree representation $\mathcal{T}_{i,j}$ for the corresponding sub-problem $N_{i,j}$ exists. We do not pass these type trees explicitly as for the previous algorithms, since they are implicitly passed with G. The graph G may additionally contain some edges representing sub-problems of length l. However, those edges can never be part of the sub-graph induced by the vertex set $\{v_i, \ldots, v_{j+1}\}$. In other words, a shortest path from v_i to v_{j+1} in G is equivalent to the shortest path computed by Algorithm 8 in the sense that it leads to the exact same representation of the sub-problem $N_{i,j}$. Figure 5.2 illustrates an example graph G and the induced sub-graph required for finding the representation $\mathcal{T}_{\mathsf{strc}}$ for a sub-problem.

Once $\mathcal{T}_{\mathsf{idx}}$, $\mathcal{T}_{\mathsf{vec}}$, $\mathcal{T}_{\mathsf{idxbuc}}$ and $\mathcal{T}_{\mathsf{strc}}$, which are each representations of the sub-problem $N_{i,j}$ and are of least cost w.r.t. all possible representations of the respective form, have been computed, the optimal representation for this sub-problem is simply the one with minimal cost. Finally, the edge (v_i, v_{j+1}), corresponding to the newly found solution $\mathcal{T}_{i,j}$ for the sub-problem $N_{i,j}$, is added to G, ensuring that $\mathcal{T}_{i,j}$ can be used as a sub-type for other, longer sub-problems. The weight of this edge is equal to the cost of including this representation as a sub-type, i.e., $(v_i, v_{j+1}) = 2K_{\text{lookup}} + \mathcal{T}_{i,j}.cost$.

Proof of Lemma 6. The first two steps of Algorithm 9, namely the initialization of the DAG G and the construction of solutions for each sub-problem of length 1 can trivially be done in $O(n)$ time. The algorithm then computes an optimal representation for each of the $O(n^2)$ sub-problems, i.e., the body of the nested for-loops is executed $O(n^2)$ times. Extracting the sub-problem $N_{i,j}$ is again trivially feasible in $O(l)$ times, where $l = j - i + 1$. Note that the length of the sub-problem is l, and not n. Computing the representations $\mathcal{T}_{\mathsf{idx}}$, $\mathcal{T}_{\mathsf{vec}}$ and $\mathcal{T}_{\mathsf{idxbuc}}$ with the help of Algorithm 5, Algorithm 6 and Algorithm 7 requires at most $O(l^2)$ time for each, as shown in Lemma 3 and Lemma 4. The steps for finding $\mathcal{T}_{\mathsf{strc}}$ are exactly the same as in Algorithm 8 minus the construction of the graph, and thus a shortest path for a sub-problem $N_{i,j}$ can be found and mapped to a representation via a strc node in $O(l^2)$ time. One execution of the loop body requires $O(l^2)$ time, which is $O(n^2)$ since $l \leq n$. Since the loop body is executed $O(n^2)$ times, we find that Algorithm 9 requires $O(n^4)$ time to compute an optimal type tree representation for an aligned type map M of length n. □

Algorithm 9 computes a shortest path for each of the $O(n^2)$ sub-problems. Alternatively, a shortest path for all sub-problems of length l can be computed by solving the all-pairs shortest path (APSP) problem once. Although this approach does not improve the asymptotic bound, we mention it for the sake of completeness.

Johnson's algorithm [CLRS09, p. 700] solves APSP in $O(|V|^2 \log |V| + |V||E|)$ time, and we can apply it in each iteration of the outer-for loop. Note that only the solutions for node pairs (v_i, v_{i+l}), i.e., the sub-problems of length l, are required. For pairs (v_i, v_j) with $j - i < l$, an optimal solution and thus a shortest path is already known. For pairs

(v_i, v_j) with $j - i > l$, the optimal solutions for the sub-problems of length l need to be considered too, which is done in the next iteration of the loop. We can bound the number of edges in the graph for each execution of the APSP algorithm with $O(n^2)$. Thus, finding a shortest path for all the $n - l + 1$ sub-problems of length l requires $O(n^3)$ time, implying that $O(n^4)$ time is required in total for the computation of all the required shortest paths.

5.5.1 Required space

To meet the space bound stated in Theorem 1, Algorithm 9 as presented in the previous section has to be amended slightly. We apply a standard trick often used in dynamic programming algorithms to reduce the space bound by a factor of n, proving the following lemma.

Lemma 7. *For any aligned type map M of length n, an optimal type tree representation can be found using $O(n^2)$ space.*

Algorithm 9 constructs a graph with $O(n)$ vertices and $O(n^2)$ edges, where a basic tree $\mathcal{T}_{i,j}$, representing the solution for the sub-problem $N_{i,j}$, is associated with each edge (v_i, v_{j+1}). For each edge (v_i, v_{j+1}) it suffices to store the root node of the associated basic tree $\mathcal{T}_{i,j}$ plus pointers to its child nodes, which are already stored with the respective edges. Note that the sub-problem $N_{i,j}$ is only generated temporarily but not stored explicitly with the generated graph. To meet the desired space bound, only a constant amount of space may be used by each edge and associated type tree. This is trivially true for leaf and vec nodes: Apart from one word indicating the node's kind and the cost of the type tree rooted at that node, for a leaf node only the count c needs to be stored. For vec nodes, an additional two integer values and one pointer to the child node are required.

However, idx, idxbuc and strc nodes may require $\Omega(n)$ space in the worst case, e.g., if $\mathsf{strc}(n, D, \langle \mathsf{leaf}(t_0), \mathsf{leaf}(t_1), \ldots \rangle$ is the optimal representation of $M = (\langle t_0, t_1 \ldots \rangle, D)$. We avoid this issue by storing for each node only the information required to reconstruct the full solution once the algorithm has terminated. A node for which only this partial information is stored is called a *node fragment*. To meet the space bound, no lists of displacements, bucket sizes or sub-types can be stored. In particular, we store

- for idx nodes the count c and a pointer to the sub-type $\mathcal{S}$;
- for idxbuc nodes the count c, the bucket stride s, a pointer to the sub-type $\mathcal{S}$ and the length q of the sub-problem represented by $\mathcal{T}$;
- for strc nodes the count c.

Note that the sub-type $\mathcal{S}$ for idx nodes could alternatively be derived easily. Due to its association with an edge (v_i, v_{j+1}), we know that the idx node is the root node of the type tree $\mathcal{T}_{i,j}$, which represents the sub-problem $N_{i,j}$. The sub-problem is of length $l = j - i + 1$ and therefore $\mathcal{S} = \mathcal{T}_{i,i+q-1}$, with $q = l/c$. Given the input type map

Algorithm 11: Constructing a full type tree out of a type tree containing node fragments.

Input: Type tree $\mathcal{T}_{i,j}$ representing the sub-problem $N_{i,j}$ of the type map $M = (T, D)$; $\mathcal{T}_{i,j}$ may contain node fragments. DAG G as constructed by `Typetree`.

Output: Full type tree representation.

```
1  Function Reconstruct(T_{i,j}, M, G)
2      if T_{i,j}.kind = leaf then
3          return
4      l ← j − i + 1
5      for k ← 0; k < l; k++ do
6          D_S[k] ← D[i + k] − D[i]
7      c ← T_{i,j}.count
8      if T_{i,j}.kind = idx then
9          q ← l/c
10         T_{i,j}.disp ← ⟨D_S[0], D_S[q], ..., D_S[(c − 1)q]⟩
11         Reconstruct(T_{i,j}.subtype, M, G)
12     if T_{i,j}.kind = idxbuc then
13         disp[0] ← D[0]
14         buckets[0] ← 1
15         h ← 0
16         s ← T_{i,j}.stride
17         q ← T_{i,j}.q
18         for k ← 1; k < l/q; k++ do
19             if D_S[kq] − D_S[(k − 1)q] = s then
20                 buckets[h] ← buckets[h] + 1
21             else
22                 h ← h + 1
23                 disp[h] ← D_S[kq]
24                 buckets[h] ← 1
25         T_{i,j}.buckets ← buckets
26         T_{i,j}.disp ← disp
27         Reconstruct(T_{i,j}.subtype, M, G)
28     if T_{i,j}.kind = strc then
29         Remove edge (v_i, v_{j+1}) from G
30         T_{i,j} ← Struc(M, G, i, j+1)
31         for k ← 0; k < T_{i,j}.count, k++ do
32             Reconstruct(T_{i,j}.subtype[k], M, G)
```

$M = (T, D)$, the displacement sequence D_S for the sub-problem $N_{i,j}$ can easily be derived as $D_S = \langle D[i] - D[i], D[i+1] - D[i], \ldots, D[j] - D[i] \rangle$. The array of displacements for an idx node can thus be recomputed as $\langle D_S[0], D_S[q], \ldots, D_S[c-1)q] \rangle$. For an idxbuc node, we additionally store the length q of the sub-problem represented by the sub-type $\mathcal{S}$. Note that contrary to idx nodes, the value q cannot be easily derived for idxbuc nodes since it is not equal necessarily equal to l over c. The count c defines the number of buckets, while the number of replications of the repeated prefix is equal to

$$\sum_{i=0}^{c-1} buckets[i] \quad .$$

The displacement and bucket size arrays can be recomputed by the same steps as used in Algorithm 7, lines 8 – 17.

The parameters of a strc node associated with an edge (v_i, v_{j+1}) can be reconstructed by again computing the shortest path from node v_i to v_{j+1} and mapping it to a strc node as done by Algorithm 10.

Proof of Lemma 7. Note that this reconstruction step as shown in Algorithm 11 does not change the asymptotic runtime bound, since all costs are additive to the $O(n^4)$ time bound of Algorithm 9. The required space for each edge and associated node is $O(1)$. There are $O(n^2)$ edges and $O(n)$ vertices in G and the weight function requires $O(n^2)$ space to be stored. It is trivial to see that the remaining data, such as the computed shortest path, the extracted sub-problems and sequences of displacements and datatypes can easily be handled in a total of $O(n^2)$ memory. The claimed upper bound of $O(n^2)$ space follows. □

5.6 General type maps

In this section, we show how an optimal type tree for general, non-aligned type maps can be constructed.

Lemma 8. *Given optimal nice type tree representations $\mathcal{T}_{i,j}$ for all sub-problems $N_{i,j}$ of a type map $M = (T, D)$ of length n, an optimal type tree $\mathcal{T}$ representing M can be computed in $O(n^2)$ time and $O(n^2)$ space.*

Proof. By Lemma 1, a cost-equivalent nice type tree representation $\tilde{\mathcal{T}}$ of M exists for any optimal type tree representation $\mathcal{T}$ of M. It therefore suffices to find an optimal nice type tree $\tilde{\mathcal{T}}$ for M. By assumption, an optimal nice type tree representation $\mathcal{T}_{0,n-1}$ for the aligned type map exists. We distinguish the following two cases to construct $\tilde{\mathcal{T}}$:

Case 1: If $\mathcal{T}$ contains an irregular node, find the first such node on any root to leaf path in $\mathcal{T}_{0,n-1}$. Adding the type map's shift $x = D[0]$ to the displacements of this node generates the desired solution. In particular, if the top-most irregular node in $\mathcal{T}$ is an idx node $\mathcal{N} = \mathsf{idx}(c, \langle i_0, \ldots, i_{c-1} \rangle, \mathcal{S})$, set $\tilde{\mathcal{N}} = \mathsf{idx}(c, \langle i_0 + x, \ldots, i_{c-1} + x \rangle, \mathcal{S})$. The construction is analogous for idxbuc and strc nodes. Note that $\tilde{\mathcal{T}}$ represents M and is

of the same cost as $\mathcal{T}$. Thus, $\tilde{\mathcal{T}}$ is an optimal representation of M. Finding the first irregular node in $\mathcal{T}$ requires $O(\log n)$ steps, since the height of the tree is limited by the same upper bound (Lemma 2). The list of displacements of $\mathcal{N}$ contains at most n elements and therefore this construction is feasible in $O(n)$ time.

Case 2: If no irregular node exists in $\mathcal{T}$, it follows from the nice type tree property (Lemma 1) and Corollary 6 that the optimal representation is either

- $\tilde{\mathcal{T}} = \mathsf{idx}(c, \langle \ldots \rangle, \mathcal{T}_{0,n/c-1})$, for some divisor c of n; or
- $\tilde{\mathcal{T}} = \mathsf{idxbuc}(c, s, \langle \ldots \rangle, \langle \ldots \rangle, \mathcal{T}_{0,n/c-1})$, for some divisor c of n; or
- $\tilde{\mathcal{T}} = \mathsf{strc}(c, \langle \ldots \rangle, \langle \mathcal{T}_{0,j_1}, \mathcal{T}_{j_1,j_2}, \ldots, \mathcal{T}_{j_{c-1},n-1} \rangle)$, for some c, $1 \leq c \leq n$ and sub-types $\mathcal{T}_{j_i,j_{i+1}}$ with $0 \leq j_i < n$ for all i.

Note that both of the trivial representations with $c = 1$ and $c = n$ need to be considered for each node. Such a node with count 1 does not violate the nice type tree property, since it is the only such node in the constructed type tree and also the first irregular node on any root to leaf path. Although any irregular node with count 1 has the same effect (it only shifts the represented type map), we cannot assume that an idx node with count 1 is cheaper than an idxbuc or strc node with count 1. A representation of least cost among all those given by the above characterization can be found by exhaustive search. For the idx and idxbuc nodes, this can be done the same way as in Section 5.2 and Section 5.3, i.e., by checking for each divisor c of n whether the prefix M_q with $q = n/c$ is repeated in M. The algorithms need to be modified slightly to cover all divisors of n, including n itself. This modification does not change the asymptotic time and space bounds. It is easy see that these two algorithms compute a correct result also for non-aligned type maps (this is not true for Algorithm 6, which however is not required for this step). Thus, Lemma 3 and Lemma 4 are applicable and we conclude that this step requires $O(n^2)$ time. For the strc node, we can apply Lemma 5 with a small modification: The generated graph has to contain also the edge (v_0, v_n) with a cost of $2K_{\text{lookup}} + \mathcal{T}_{0,n-1}.\mathit{cost}$. Lemma 5 still holds with this modification, implying that this step too is feasible in $O(n^2)$ time. Overall, the construction requires only $O(n^2)$ time since the solutions $\mathcal{T}_{i,j}$ for the sub-problems $N_{i,j}$ are already known. The space bound is $O(n^2)$ by the same arguments as used in the proof of Lemma 7. □

We can now prove that the Type Reconstruction Problem for general type maps can be solved in $O(n^4)$ time and $O(n^2)$ space, as claimed in Theorem 1.

Proof of Theorem 1. For a general, non-aligned type map $M = (T, D)$, an optimal type tree $\mathcal{T}$ is constructed by the following three steps:

1. Let M' be the aligned type map M and let x be M's shift, i.e., $x = D[0]$ and $M' = M - x$.
2. Use Algorithm 9 to compute optimal type tree representations $\mathcal{T}_{i,j}$ for all sub-problems $N_{i,j}$ of M'. This includes the optimal type tree representation $\mathcal{T} = \mathcal{T}_{0,n-1}$

or the aligned type map $M' = N_{0,n-1}$. Note that the type trees $\mathcal{T}_{i,j}$ may contain node fragments, for which some missing arguments remain to be recomputed.

3. To construct an optimal type tree for the original type map M, we distinguish two cases:

   - If $\mathcal{T}$ contains at least one irregular node, compute the complete solution out of the fragment as detailed in Section 5.5.1. Find the first irregular node on any root to leaf path and add the type map's shift x to all displacement values, i.e., apply Case 1 of Lemma 8.
   - Otherwise, apply Case 2 of Lemma 8 to construct a fragment of an optimal solution before recomputing the complete solution as shown in Section 5.5.1.

   Normalizing a type map is trivially feasible in linear time and requires only a constant amount of additional space. Thus, the whole construction is clearly dominated by the $O(n^4)$ time required by Algorithm 9. The space bound follows from the $O(n^2)$ space required by both Algorithm 9 and the last step of generating the solution for a general type map (Lemma 8).

□

CHAPTER 6

Type path reconstruction

In this chapter, we present our algorithm for the Type Path Reconstruction Problem. For this special case of the Type Reconstruction Problem, we restrict ourselves to the leaf, vec and idx constructors. Our algorithm solves the problem in $O(n \log / \log \log n)$ time, which is a significant improvement over the previously best known bound of $O(n\sqrt{n})$ by Träff [Trä14]. We detail their approach in Section 6.1, before setting out to prove the following lemma.

Theorem 2. *Given a homogeneous type map M of length n, the* Type Path Reconstruction Problem *can be solved in $O(n \log n / \log \log n)$ time and $O(n)$ space.*

The structure of an optimal type path is analogous to the structure of an optimal type tree as given in Definition 10, minus the strc and idxbuc nodes.

Definition 14 (Structure of an optimal type path)**.** *An optimal type path $\mathcal{P}$ for a homogeneous type map $M = (t, D)$ of length n is either*

1. *$\mathcal{P} = \mathsf{leaf}(t)$, a single leaf node with base type t if n equals 1; or*
2. *$\mathcal{P} = \mathsf{vec}(c, s, \mathcal{Q})$, where the prefix M_q of length $q = n/c$ is a strided prefix in M with stride s and $\mathcal{Q}$ is an optimal type path for M_q; or*
3. *$\mathcal{P} = \mathsf{idx}(c, \langle i_0, \ldots, i_{c-1} \rangle, \mathcal{Q})$, where the prefix M_q of length $q = n/c$ is a repeated prefix in M, $\mathcal{Q}$ is an optimal type path for M_q and the displacements $i_0, \ldots, i_{c-1}$ are such that $\texttt{Flatten}(\mathcal{P}, 0) = M$.*

A type path contains exactly one leaf node and can thus only represent homogeneous type maps, i.e., type maps that consist of a single base type. To reduce notational overhead and since all base types are equal, we denote homogeneous type maps by $M = (t, D)$, i.e., with a single base type t instead of a sequence of base types. Thus, a prefix M_q of M is a repeated prefix if and only if the prefix D_q of the displacement sequence D is a repeated prefix, i.e., the type signature need not be checked.

Table 6.1: In the type map $M = (\texttt{char}, D)$, with the first 16 values of D as shown in the second table row, the prefixes of length 2 and 8 are repeated. The prefix of length 4 is not repeated in M, since $D[6] - D[5] = 3 \neq D[6 \bmod 4] - D[5 \bmod 4] = D[2] - D[1] = 2$.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$D[i]$	0	1	3	4	8	9	12	13	20	21	23	24	28	29	32	33
$D[i] - D[i-1]$		1	2	1	4	1	3	1	87	1	2	1	4	1	3	1

Since any type path is a type tree, all properties shown in Chapter 4 also hold for type paths. In particular, the problem exhibits optimal sub-structure. Equivalently to nice type trees, for each optimal type path a nice type path of equal cost exists. For aligned type maps, an optimal type path does not contain any nodes with count 1 and at most one shifting node. Our algorithm for the Type Path Reconstruction Problem is similar to the algorithm solving the Type Reconstruction Problem given in Chapter 5. Note that all constructors considered for the Type Path Reconstruction Problem problem require a repeated prefix to exist, which is used as the sub-type. Contrary to the Type Reconstruction Problem, a segment that is not a prefix of the input type map can never be used as a sub-type in a type path. In order to achieve the drastically lower runtime bound, some new ideas for computing repeated prefixes are required. The key idea is to find all repeated and strided prefixes in a preprocessing step, so that the dynamic programming algorithm performing the actual reconstruction can be implemented more efficiently. The straight-forward approach to find all repeated prefixes is to check for each divisor q of n whether the prefix of length q is repeated in D. While the check for a given divisor can trivially be done in linear time (see Algorithm 4), the number of divisors $d(n)$ can be somewhat large. As shown in Landau [Lan09, pages 219–222],

$$d(n) = e^{\Theta\left(\frac{\log n}{\log \log n}\right)} \quad .$$

Note that this bound is sharp. Although this approach directly leads to an $O\left(nn^{\frac{1}{\log \log n}}\right)$ (i.e., a sub-quadratic) time procedure for computing all repeated prefixes, it does not seem to allow for an $O(n \log n)$ time algorithm.

Somewhat counter-intuitively, the existence of a repeated prefix of length c does not imply that the prefixes of length kc or c/k, with $k \in \mathbb{N}$, are also repeated. Table 6.1 gives an example type map where the prefixes of length 2 and 8 are repeated, but the prefix of length 4 is not. Figure 6.1 shows a type path representation for the example type map, which directly shows that the prefixes of length 2 and 8 are repeated. This observation makes it clear that computing all repeated prefixes efficiently, i.e., faster than the straight-forward approach outlined above, is not a trivial task.

We present in Section 6.2 a more efficient algorithm that finds all repeated prefixes of a homogeneous type map in $O(n \log n / \log \log n)$ time. This is a crucial building block for our algorithm solving the Type Path Reconstruction Problem for the special

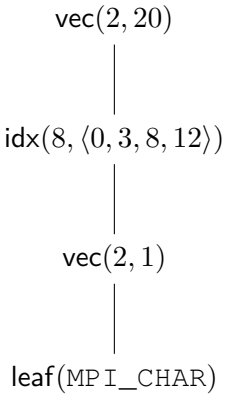


Figure 6.1: A type tree representing the type map $M = (\texttt{char}, \langle 0, 1, 3, 4, 8, 9, 12, 13, 20, 21, 23, 24, 28, 29, 32, 33\rangle)$. As Table 6.1 shows, the prefixes of length 2 and 8 are repeated, but the prefix of length 4 is not.

case of aligned type maps, which is detailed in Section 6.3. Section 6.4 shows how to derive a solution for a general type map.

With our approach it is furthermore possible to also include the idxbuc constructor. Such a type path is called an *extended type path*, which in addition to Definition 14 may be of the form

4. $\mathcal{P} = \mathsf{idxbuc}(c, s, \langle b_0, \ldots, b_{c-1}\rangle, \langle i_0, \ldots, i_{c-1}\rangle, \mathcal{Q})$, where the prefix M_q of length $q = n/c$ is a repeated prefix in M, $\mathcal{Q}$ is an optimal extended type path for M_q and the displacements $i_0, \ldots, i_{c-1}$ together with the bucket lengths $b_0, \ldots, b_{c-1}$ and the stride s are such that $\texttt{Flatten}(\mathcal{P}, 0) = M$.

This constructor can be included in the algorithm in a straight-forward manner as shown in Section 6.5. However, this increases the time bound to $O(n^2 \log^2 n)$. Although this may not be feasible in practice, it demonstrates that our algorithm may easily be extended to consider further constructors. We strongly suspect that more efficient approaches for the idxbuc constructor exist.

6.1 Träff's type path reconstruction and normalization algorithm

Träff [Trä14] recently presented an algorithm that solves the Type Path Reconstruction Problem in $O(n\sqrt{n})$ time. They use this algorithm as a basis to solve the Type Path Normalization Problem in $O(dl\sqrt{l} + (d \log l)^3)$ time, where d is the height (or depth) of the type path and l is the length of the longest occurring displacement sequence. Their approach is based on a nice property stating that a cost-optimal type path can always be constructed out of a so-called *canonical type path* by purely local optimizations.

The formal model used in their work is basically the same as the one detailed in Chapter 2, with a few minor differences.

- Consideration is restricted to homogeneous type maps, since a type path cannot represent a type map consisting of more than one basic datatype.
- The remaining basic datatype is discarded, and data layouts are interpreted as a sequence of bytes. They thus assume that the input for the Type Path Reconstruction Problem is a displacement sequence without any type information.
- A leaf node, denoted by $\mathsf{con}(c)$, with count c describes a sequence of c *contiguous* displacements $0, 1, \ldots, c-1$. Since no type information is kept, this node is also used to represent basic datatypes whose representations span multiple bytes (as is usual for basic datatypes such as, e.g., `int` and `float`).
- The assumed cost model is the same as in this work, but different, changing cost values are adopted for examples.

The following *structural lemma* contains the key observation for their algorithm. We paraphrase the lemma to match the definitions and conventions of this work.

Lemma 9. *Let D be a displacement sequence that is not regularly strided, and let D_q be the shortest repeated prefix of length q, $q \geq 2$. For any longer prefix $D_{q'}$ of length q', $q' > q$ that is also repeated in D, it holds that q' is a multiple of q.*

The reader is referred to [Trä14] for a formal proof of this lemma. Intuitively, Lemma 9 shows that for displacement sequences that do not contain a strided prefix it suffices to find the shortest repeated prefix. The length of all other repeated prefixes is a multiple of the length q of the shortest repeated prefix D_q and they can be discovered by recursively analyzing D_q. This lemma is the basis for an algorithm constructing a *canonical type path* for the given displacement sequence. A canonical type path is not necessarily a least-cost representation, but has the following useful properties.

Definition 15 (Canonical type path)**.** *A* canonical type path *for a displacement sequence of length n is a type path from which any other type path representation of D can be obtained by repeatedly applying the following two modifications:*

- *Combine two successive nodes.*
- *Split a* vec *node with count c into two* vec *nodes with counts c_0 and c_1, where $c = c_0 c_1$.*

By the same arguments that we use to bound the height of an optimal type tree (Lemma 2), the height of a canonical type path is at most $\lceil \log_2 n \rceil$. Care must be taken to cover the two modifications for all possible combinations of considered constructors. A dynamic programming algorithm is used to perform local improvements according to the two possible modifications outlined above. The paper considers only the idx, vec and

con constructors, which already lead to ten different cases. It is not clear if and how the canonical type path property can be extended to include further constructors. A curious feature of the algorithm is that in this step, global optimality is achieved by purely local optimizations.

These ideas to solve the Type Path Reconstruction Problem directly lead to a non-trivial approach for the Type Path Normalization Problem. For a given type path that is to be normalized, the algorithm performs the following sequence of steps.

1. Convert idx to vec nodes wherever possible, i.e., where the prefix represented by the sub-type is regularly strided. Note that this operation is similar to the heuristic optimization of specializing a constructor where possible (see Chapter 3).
2. Combine nested vec nodes where possible.
3. For each remaining idx node, apply type reconstruction on the represented displacement sequence.
4. Perform local optimizations as outlined above.

To show that the resulting normalized type path is indeed optimal, it suffices to show that the normalization procedure constructs the same type path that results from solving the Type Path Reconstruction Problem on the flattened input type tree.

6.2 Efficient computation of repeated prefixes

We now present our algorithm for efficiently computing all repeated prefixes, proving the following lemma.

Lemma 10. *For any displacement sequence D of length n, all repeated prefixes can be found in $O(n \log n / \log \log n)$ time and $O(n)$ space.*

In a homogeneous type map $M = (t, D)$, a prefix M_q is repeated if and only if the prefix D_q is repeated in the displacement sequence D, i.e., the base type is of no significance. In this section, we therefore restrict consideration to displacement sequences.

For a given divisor q of n, we define the set R_q as

$$R_q = \{p \geq 1 \mid p \text{ divides } q \ \wedge \ D_p \text{ is a repeated prefix in } D\} \quad .$$

The set R is defined as

$$R = \bigcup_{q|n} R_q \quad .$$

We give an example to illustrate this rather technical notion and to highlight the difference between a set R_q and the set R: Assume a displacement sequence D of length $n = 24$ where, in addition to the trivially repeated prefixes of length 1 and 24, the prefixes of length 6, 8 and 12 are repeated. For $q = 12$, we find that $R_{12} = \{1, 6, 12\}$, i.e., R_{12} contains all repeated prefixes whose length p is a divisor of 12. The repeated prefix of

length 8 is not in R_{12}, since 8 is not a divisor of 12. This prefix is however contained in the set R_8 and we we have that $R = \{1, 6, 8, 12, 24\}$ for this example. Each repeated prefix is contained in at least one set R_q and thus in the set R. For a repeated prefix D_p of length p, p is in the set R_p, since p is necessarily a divisor of n (otherwise, D_p cannot be a repeated prefix) and p trivially divides p.

Our approach for computing the set of all repeated prefixes for a given displacement sequence is detailed in Algorithm 14. It uses Algorithm 13, which computes the set R_q for a given displacement sequence and divisor q.

Lemma 11. *Let D be an arbitrary displacement sequence of length n and q be an arbitrary divisor of n. All prefixes repeated in D that are of length p, where p is a divisor of q (i.e., the set R_q), can be found in linear time.*

To find the whole set R of repeated prefixes, we need to ensure that all possible divisors of n are covered. This is done by applying the above lemma for those divisors of n that contain all but one of its distinct prime factors. More formally, given the prime factor decomposition $n = f_1^{a_1} f_2^{a_2} \dots f_k^{a_k}$ of n, let $q_i = n/f_i$ for $1 \leq i \leq k$ and apply Lemma 11 for each q_i. The correctness of this approach and the claimed time and space bounds are shown in the remainder of this section. While the algorithms presented in this section are quite short and straight-forward to implement, the proofs are somewhat technical and require careful attention to detail.

The key idea for Algorithm 13 is that once some repeated prefix has been found, some (but not all) shorter repeated prefixes can be found easily by recursively analyzing the already found repeated prefix. Assume that a prefix D_q is repeated in a displacement sequence D of length n. For each divisor p of q, it suffices to check if the prefix D_p is repeated in D_q to determine whether the prefix D_p is repeated in D. In other words, the segment $D[q, n-1]$ does not have to be checked explicitly.

Corollary 9. *Assume an n-element displacement sequence D s.t. the prefix D_q, $1 \leq q \leq n$, is a repeated prefix. If the prefix D_p, for $1 \leq p \leq q$, is repeated in the prefix D_q, D_p is also repeated in D.*

Proof. The definition of repeated prefixes (Definition 6) can equivalently be stated as

$$q \mid n \wedge\ \forall i\, \forall j\ 1 \leq i < n/q,\ 1 \leq j < q : D[j] - D[j-1] = D[iq + j] - D[iq + j - 1] \quad .$$

Taking $S[i] = D[i] - D[i-1]$, this is equivalent to

$$\forall i, 1 \leq i < n :\ i \bmod q = 0 \vee S[i \bmod q] = S[i] \quad .$$

Since D_q is repeated in D and D_p is in turn repeated in D_q, we have that

1. $\forall i,\ 1 \leq i < n$: $i \bmod q = 0 \vee S[i \bmod q] = S[i]$ and
2. $\forall j,\ 1 \leq j < q$: $j \bmod p = 0 \vee S[j \bmod p] = S[j]$.

Since q divides n and p divides q, we get that

$$\forall i, 1 \leq i < n : \; i \mod p = 0 \vee S[i \bmod p] = S[i] \quad ,$$

i.e., the prefix of length p is repeated in D. □

To compute R_q, Algorithm 13 employs Algorithm 12 to find a repeated prefix that quickly leads to as many additional repeated prefixes as possible. To make this notation precise, the following definitions are required.

Definition 16 (Postfix of a displacement sequence)**.** *Given a displacement sequence D of length n, a* postfix starting at index q *denotes the segment $D[q, n-1]$.*

Definition 17 (Potentially repeated prefix)**.** *Given a displacement sequence D of length n and a non-trivial divisor q of n, we say that a prefix D_p is* potentially repeated *in D w.r.t. q if the prefix D_p is repeated in the postfix $D[q, n-1]$, i.e., D_p being repeated in the prefix D_q implies that D_p is also a repeated prefix in D.*

Algorithm 12: Finding the longest potentially repeated prefix in the postfix starting at index q.

Input: Displacement sequence D of length n; non-trivial divisor q of n.
Output: Largest p s.t. p divides q and the prefix of length p is potentially repeated in D w.r.t. q.

```
1 Function LongestInPostFix(D, n, q)
2     for i ← 1; i < n/q; i++ do
3         for j ← 1; j < q; j++ do
4             if D[j] − D[0] ≠ D[iq + j] − D[iq] then
5                 D′ ← D[iq + j, n − 1]
6                 n′ ← n − iq − j
7                 q′ ← gcd(q, j)
8                 return LongestInPostFix(D′, n′, q′)
9     return q
```

Given a displacement sequence D of length n and a divisor q of n, Algorithm 12 finds the longest *potentially* repeated prefix D_p, where p is a divisor of q. The algorithm checks for every segment $D[iq, (i+1)q-1]$, $1 \leq i < n/q - 1$, whether the prefix of length q is repeated. If this is the case, the prefix D_q is a repeated prefix in the postfix $D[q, n-1]$ and therefore also the longest potentially repeated prefix w.r.t. q. Since D_q is trivially repeated in the segment $D[0, q-1]$, the potentially repeated prefix D_q is a repeated prefix in D.

If the condition for repeated prefixes is not met at index j of the i-th segment $D[iq, (i+1)q-1]$, the procedure proceeds recursively to find the longest prefix of length p repeated in the postfix $D[iq + j, n-1]$, where p is a divisor of $\gcd(q, j)$. The condition

Algorithm 13: Computing the set R_q for a displacement sequence D of length n and a non-trivial divisor q of n.

Input: Displacement sequence D of length n; non-trivial divisor q of n; the distinct prime *factors* of q in ascending order.
Output: r_q is set to **true** if the prefix of length q is repeated in D.

```
1  Function AllRRPForDivisor(D, n, q, factors)
2      if q == 1 then
3          r_1 ← true
4          return
5      q_l ←LongestInPostFix(D, n, q)
6      if q_l == q then
7          r_q ← true
8          n' ← q
9          f ← smallest prime factor of q
10         q' ← q/f
11         Remove one occurrence of f from factors
12     AllRRPForDivisor(D, n', q', factors)
```

being met up to index $iq + j - 1$ implies that the prefix D_q is repeated in the segment $D[q, iq+j-1]$. Once the recursive call returns with the length p of the longest potentially repeated prefix in the segment $D[iq + j, n - 1]$, it suffices to check whether the prefix D_p is repeated in the prefix D_q to determine whether or not D_p is repeated in the segment $D[q, iq + j - 1]$ (Corollary 9). Since D_p is repeated in the postfix starting at index $iq + j$, D_p is potentially repeated in the postfix starting at index q. This extends to divisors o of p: If D_p is potentially repeated in the postfix $D[q, n - 1]$ and the prefix D_o is repeated in D_q (and therefore also in the prefix D_p), D_o is repeated in D.

The condition not being met at index $iq+j$ means that $D[iq+j]-D[iq] \neq D[j]-D[0]$, implying that the length p of the longest repeated prefix in D w.r.t q is at most j. Since we are only interested in those prefixes of length p where $p \mid q$, we find that $p \leq \gcd(q, j)$. Since the prefix of length 1 is a repeated prefix in any displacement sequence and 1 is always a divisor of q, the procedure is guaranteed to find a longest repeated prefix.

Algorithm 13 uses Algorithm 12 to efficiently compute the set R_q, i.e., all repeated prefixes of length p where p divides q. If the prefix D_q is potentially repeated in the postfix $D[q, n - 1]$, it is trivially also a repeated prefix in D. The algorithm proceeds by recursively computing all repeated prefixes in R_q. Note that the longest possible, non-trivial repeated prefix in R_q is of length less than or equal to q over the smallest prime factor of q. Due to Corollary 9, any prefix repeated in D_q is also repeated in the displacement sequence D. If the longest potentially repeated prefix is of length $q' < q$, we need to check whether the prefix of length q' is repeated in the prefix of length q. This is done by recursively computing all repeated prefixes of the prefix $D[0, q - 1]$. The prime factors of q (with multiplicity) are passed to Algorithm 13 in the initial call and

the algorithm keeps track of the prime factorization of the parameter q.

Proof of Lemma 11. Algorithm 12 requires $O(n)$ time, due to the recursive call proceeding with the postfix $D[iq + j, n - 1]$ in case the condition for repeated prefixes is not met for an element at index $iq + j$. Note that when called with parameter q for a displacement sequence of length n, the procedure does not do the check for any element in the prefix $D[0, q - 1]$. We use this fact to account for any subsequent calls of Algorithm 12 that happen in Algorithm 13: The algorithm recurses on the prefix of length (at most) q, for which it calls Algorithm 12 with a displacement sequence of length q. Therefore, during an execution of Algorithm 13, Algorithm 12 performs the check on line 4 of Algorithm 12 for each element at most once. The smallest prime factor of q is stored as the first element of *factors*. If *factors* is implemented as an ordered list, the smallest prime factor of q can be retrieved and removed in constant time. □

Given a displacement sequence D of length n, Algorithm 14 sets $r_q =$ **true** if and only if the prefix D_q is repeated in D. The algorithm first factorizes n, the length of the input sequence. The list *all* is used to store all prime factors of n in ascending order, while *distinct* keeps track of its unique prime factors. Given the prime factor decomposition of n, $n = f_1^{a_1} f_2^{a_2} \dots f_k^{a_k}$, *all* contains a_1 times the prime factor f_1, followed by a_2 occurrences of f_2 and so on. The list *distinct* contains each of $p_1, p_2, \dots, p_k$ exactly once, in arbitrary order. We make no claim that the given approach for factorizing n is particularly efficient. However, as the analysis will show, its asymptotic runtime behavior suffices for our purposes. The algorithm then proceeds to find all repeated prefixes by applying Lemma 11 once for each distinct prime factor f_i of n. It executes Algorithm 13 with parameter $q = n/f_i$ for each f_i, $1 \leq i \leq k$, to compute the set R_q. The prime factor decomposition of q can easily be obtained by removing one occurrence of p_i from the prime factor decomposition of n. It is passed to Algorithm 13, which requires that the first element may be accessed and removed in constant time. Therefore, *all* and *factors* are assumed to be implemented as ordered linked lists.

Proof of Lemma 10. The prime factorization as outlined in Algorithm 14 is feasible in linear time w.r.t. the size of the input for the Type Reconstruction Problem, which is $O(n)$. We do not claim that factorizing an integer n is possible in linear time w.r.t. the size of the binary representation of n, and therefore the previous statement does not contradict the fact that prime factorization is not known to be solvable in polynomial time. More efficient approaches can be found in [Knu81, p. 379-417]. Note that any number is a composite of at most $\log_2(n)$ prime factors. The worst case is n being a power of 2 and thus the size of both *all* and *distinct* is $O(\log n)$.

Algorithm 14 applies Lemma 11 (implemented as Algorithm 13) for each $q_i = n/f_i$, where the f_i are the distinct prime factors. Any non-trivial divisor q of n is equal to n/f, where f is a divisor of n that contains at least 1 of n's prime factors. Since Algorithm 13 with parameter q finds all repeated prefixes of length q' where q' divides q, calling it for all $q_i = n/f_i$ ensures that all divisors of n are covered.

As shown in the proof of Lemma 11, Algorithm 13 requires $O(n)$ steps on displacement sequences of length n. The number of executions is bounded by the number of distinct

Algorithm 14: Finding all repeated prefixes.

Input: Displacement sequence D of length n.
Output: $r_q = \textbf{true}$ if and only if the prefix of length q is repeated in D.

```
1  Function AllRepeatedPrefixes(D, n)
2      // prime factorization of n
3      f ← 2
4      d ← 0
5      n' ← n
6      all ← null
7      distinct ← null
8      while f ≤ n' do
9          if n' mod f = 0 then
10             if f ≠ d then
11                 d ← f
12                 distinct.add(d)
13             all.add(f)
14             n' ← n'/f
15         else
16             f++
17     // Initialization
18     for i ← 1; i ≤ n; i++ do
19         r_i ← false
20     // Find all repeated prefixes
21     for i ← 1; i < |distinct|; i++ do
22         factors ← all
23         Remove one occurrence of distinct[i] from factors
24         AllRRPForDivisor(D, n, n/distinct[i], factors)
25     // Trivially repeated prefix
26     r_n ← true
```

prime factors of n. As shown by Robin [Rob83], the number of distinct prime factors of n is upper-bounded by $O(\log n / \log\log n)$, implying the claimed runtime bound. Apart from the two linked lists of size $O(\log n)$, the algorithm requires $O(n)$ variables r_i. The total required space is therefore $O(n)$. □

6.3 Full algorithm for aligned type maps

In this section, we present our algorithm that solves the Type Path Reconstruction Problem in $O(n \log n / \log\log n)$ time for the special case of aligned homogeneous type maps. Some of the key ideas are the same as for our algorithm solving the Type Reconstruction Problem (see Chapter 5). Since any type path is a type tree, the properties discussed in Chapter 4 hold for them as well. In particular, a *nice type path* of equal cost exists for any optimal type path. Furthermore, the problem exhibits optimal sub-structure. For the Type Path Reconstruction Problem, the sub-problems are the prefixes of the aligned type map. We denote the sub-problem of length q as N_q, i.e., $N_q = (t, D[0, q-1])$ for $1 \leq q \leq n$. An optimal type path for the sub-problem N_q is denoted by $\mathcal{P}_q$. We can reduce the required space by the same trick as used in Section 5.5.1, namely that of reconstructing a full solution out of solution fragments once the algorithm terminates. A solution for a general type map can be constructed out of the solutions for all prefixes of a homogeneous type map. This is similar to the approach in Section 5.6, but can be done much more efficiently for type paths. In addition to these ideas, we use two preprocessing steps to efficiently compute repeated and strided prefixes. Once this information is available, the algorithm can construct an optimal type path in near-linear time.

Algorithm 15 proceeds in four steps. In the first, the algorithm constructs the trivially optimal representation for the sub-problem N_1. For an aligned type map, the prefix of length 1 is the type map $(t, \langle 0 \rangle)$ and therefore the optimal representation is via a leaf node. The algorithm thus directly sets $\mathcal{T}_1 = \mathsf{leaf}(t)$.

In the second step, all repeated prefixes of D are computed with the help of Algorithm 14. As discussed in the previous section, a prefix M_q of a homogeneous type map $M = (t, D)$ is a repeated prefix if and only if the prefix D_q is repeated in the displacement sequence D. If the prefix D_p is repeated in D_q, its type path representation $\mathcal{P}_p$ can be used as a sub-type for the representation $\mathcal{P}_q$.

In the third step, the algorithm computes for each repeated prefix D_p the longest prefix D_l in which D_p is repeated, i.e.,

$$\max_{l, p \leq l \leq n} (D_p \text{ is strided in } D_l) \quad .$$

Note that if the prefix D_p is strided in D_l, it is also strided in any prefix D_q with q s.t. p divides q and $q \leq l$. If the repeated prefix D_p is strided in some prefix D_l, any repeated prefix D_q with $p \leq q \leq l$ and $p \mid q$ can be represented via a vec node, using $\mathcal{P}_p$ as the sub-type.

In the fourth step, the algorithm iteratively constructs optimal type path representations for longer and longer prefixes, using the already computed optimal representations

Algorithm 15: Construction of optimal type path representations for aligned homogeneous type maps.

Input: Aligned homogeneous type map $M = (t, D)$ of length n.
Output: Optimal type path representation for M.

Function `Typepath(`M`, `n`)`
  `// Step 1: Trivial optimal representation for` N_1
  $\mathcal{T}_1 = \mathsf{leaf}(t)$
  `// Step 2: Find all repeated prefixes`
  `AllRepeatedPrefixes(`D`, `n`)`
  `// Step 3: Find strided prefixes`
  for $p \leftarrow 1$; $p < n$; $p{+}{+}$ **do**
    if $r_p \neq$ **true then**
      continue
    $s_p \leftarrow p$
    $s \leftarrow D[p] - D[0]$
    while $s_p + p < n$ *and* $D[s_p + p] - D[s_p] = s$ **do**
      $s_p \leftarrow s_p + p$
  `// Step 4: Construct optimal representation for each repeated prefix`
  foreach $q \in \{i \mid r_i =$ **true**, $2 \leq i \leq n\}$ **do**
    $c_{best} \leftarrow \infty$
    foreach $p \in \{i \mid r_i =$ **true**, $1 \leq i < q\}$ **do**
      if $q \leq s_p \wedge p$ *divides* q **then**
        $c_{\mathsf{vec}} \leftarrow K_{\mathsf{vec}} + \mathcal{T}_p$
        if $c_{\mathsf{vec}} < c_{best}$ **then**
          $s \leftarrow D[p] - D[0]$
          $\mathcal{T}_q \leftarrow \mathsf{vec}(q/p, s, \mathcal{T}_p)$
          $c_{best} \leftarrow c_{\mathsf{vec}}$
      $c_{\mathsf{idx}} \leftarrow K_{\mathsf{idx}} + (q/p)K_{\text{lookup}} + \mathcal{T}_p.cost$
      if $c_{\mathsf{idx}} < c_{best}$ **then**
        `//` $disp \leftarrow \langle D[0], D[p], \ldots, D[(q/p-1)p] \rangle$
        $\mathcal{T}_q \leftarrow \mathsf{idx}(q/p, disp, \mathcal{T}_p)$
        $c_{best} \leftarrow c_{\mathsf{idx}}$
  return $\mathcal{T}_n$

for shorter prefixes as sub-types. To do this efficiently, each node stores the cost of the type path rooted at this node. When a new type path $\mathcal{P}_q$ for the prefix D_q is constructed out of a prefix D_p, the cost of $\mathcal{P}_q$ can be computed in constant time, by adding the cost of the constructed root node to the cost of the sub-type $\mathcal{P}_p$. The information gathered by the first three steps allows for an efficient construction of an optimal representation for a prefix of length q out of already computed optimal representations for repeated prefixes of length less than q. As discussed at the begin of this chapter, the optimal type path representation for a repeated prefix of length q, $1 < q \leq n$, is either an idx or vec node with an optimal representation of a shorter repeated prefix as the sub-type and appropriate parameters. The least cost representation is found by enumerating all possible representations of the prefix of length q, as done in the body of the nested for-loop. If two prefixes D_p and D_q, $p < q$ are both repeated in D, it is clear that D_p is also repeated in D_q. A representation of D_q via an idx node of the form $\mathcal{P}_q = \mathsf{idx}(q/p, \langle D[0], D[p], \ldots \rangle, \mathcal{P}_p)$ is possible. The prefix of length 1 is a repeated prefix in any homogeneous type map and thus the algorithm is guaranteed to find a valid representation for all repeated prefixes. If the prefix D_p is a strided repetition in the prefix D_q, D_q can alternatively be represented via a vec node as $\mathcal{P}_q = \mathsf{vec}(q/p, D[p] - D[0], \mathcal{P}_p)$. Note that r_n was set to **true** by Algorithm 14, since the prefix of length n is by definition a repeated prefix. Thus, an optimal type path representation for the full displacement sequence is constructed in the last iteration.

The first step implicitly assumes constant update times for the variables r_i, which can be guaranteed by storing them in an n-element array. In the fourth step however we cannot afford to scan an array of size n to find those r_i that were set to **true**, since that would lead to an $O(n^2)$ time bound for the two nested loops. This can be avoided by copying only the required information to a new, smaller array that contains only the values i for which $r_i =$ **true**.

Lemma 12. *Given an aligned homogeneous type map M of length n, Algorithm 15 constructs an optimal type path representation in $O(n \log n / \log \log n)$ time and $O(n)$ space.*

Proof. As shown in Section 6.2, all the repeated prefixes of D can be computed using $O(n \log n / \log \log n)$ time and $O(n)$ space. The second step requires $O(n/q)$ time for each repeated prefix of length q. The *divisor summatory function*, i.e., the function $\sum_{q|n} q$, computes the sum of all divisors q of n. Due to Gronwall [Gro13], the result of this summation can be bounded by

$$\sum_{q|n} q = O(n \log n / \log \log n) \quad .$$

Using this upper bound, the time required by the second step can be bounded as

$$\sum_{q \text{ s.t. } r_q = \textbf{true}} n/q \leq \sum_{q|n} n/q = \sum_{q|n} q = O(n \log n / \log \log n) \quad .$$

The body of the inner for-loop of the fourth step requires only constant time due to the information gathered in the previous steps and the trick of not computing the array

of displacements for idx nodes as outlined above. The number of repeated prefixes of a displacement sequence D_q is at most equal to the number of divisors of q. Using the same rough upper bound as in the proof of Lemma 3, we can bound the number of divisors of q with $O(\sqrt{q})$. Thus, the body of the inner loop is executed $O(\sqrt{q})$ times for each q. The remainder of the outer for-loop's body trivially requires constant time. The outer loop is executed at most once for each divisor q of n and thus $O(\sqrt{n})$ times. Since $q \leq n$, we have that $O(\sqrt{q})$ is $O(\sqrt{n})$ and the body of the nested for-loop is executed $O(\sqrt{n}\sqrt{n}) = O(n)$ times.

We use the same trick as in Section 5.5.1 to prove the claimed space bound. We do not need to store the displacement array with each constructed idx node, since it can easily be reconstructed once the optimal type path has been found. In particular, if the root node of an optimal type path for a sub-problem N_q is an idx node, only the kind of the node, the count c and the cost of the type path $\mathcal{P}_q$ rooted at this node have to be stored. The full node can easily be derived as $\mathsf{idx}(c, \langle D[0], D[p], \ldots, D[(c-1)p]\rangle, \mathcal{P}_{q/c})$ with $p = q/c$.

Note that the information kept for an idx node fragment requires only constant space. For vec nodes, no reconstruction is required since all information can trivially fit into a constant amount of space. Optimal type paths are computed only for the repeated prefixes of the type map M, the number of which is upper bounded by $O(n)$. This implies that the constructed type paths for all sub-problems require only $O(n)$ space. The algorithm additionally requires $O(n)$ variables r_i and s_i. The claimed space bound follows directly. □

6.4 General type maps

Lemma 13. *Given optimal nice type path representations $\mathcal{P}_q$ for all sub-problems N_q of a homogeneous type map $M = (t, D)$ of length n, an optimal type path $\mathcal{P}$ representing M can be computed in $O(n)$ time.*

Proof. By Lemma 1, a cost-equivalent nice type path representation $\tilde{\mathcal{P}}$ of M exists for any optimal type path representation $\mathcal{P}$ of M. By assumption, an optimal nice type path representation $\mathcal{P}_n$ for the sub-problem N_n, i.e, the aligned type map M, exists. To construct an optimal representation of M, find the top most index node $\mathcal{N} = \mathsf{idx}(c, \langle i_0, \ldots, i_{c-1}\rangle, \mathcal{Q})$ in $\mathcal{P}_n$ and add the displacement sequence's shift $x = D[0]$ to all its displacements, i.e., $\mathcal{N} = \mathsf{idx}(c, \langle i_0 + s, \ldots, i_{c-1} + x\rangle, \mathcal{Q})$. Note that $\mathcal{P}$ has the same cost as $\mathcal{P}_n$ and is therefore optimal. Traversing the whole type path requires only $O(\log n)$ time (Lemma 2) and the list of displacements in an index node is at most of length n. This step is therefore feasible in $O(n)$ time.

If such an index node does not exist, it follows directly from Lemma 1 that the optimal representation of M is of the form $\mathcal{P} = \mathsf{idx}(n/q, \langle \ldots \rangle, \mathcal{P}_q)$ for some divisor q of n, including the trivial divisors 1 and n. Since solutions for all sub-problems N_q with $q < n$ already exist, an optimal type path for M can be found in linear time by enumerating (the

cost of) all possible representations of this form. The approach is analogous to the one used in Algorithm 15, except that also the prefix of length n needs to be considered. □

Proof of Theorem 2. The Type Path Reconstruction Problem for a homogeneous type map M of length n can be solved by computing an optimal type path for the aligned type map with the help of Algorithm 15 and applying the post-processing step outlined in Lemma 13. The claimed time and space bounds follow directly from Lemma 12 and Lemma 13. □

6.5 Including the index bucket constructor

In this section, we extend the algorithm for the Type Path Reconstruction Problem (Algorithm 15) to also consider the idxbuc constructor. Formally, the following problem is solved.

> Extended Type Path Reconstruction Problem
> *Instance*: A homogeneous type map M of length n.
> *Task*: Find a least-cost (or optimal) extended type path $\mathcal{P}$ representing M.

Algorithm 16: Constructing an optimal type path for the prefix M_q with an idxbuc node as root node and the repeated prefix M_p as sub-type.

```
1  E ← {D[ip] − D[(i − 1)p] | 1 ≤ i < p/q}
2  sort E
3  s ← most frequently occurring element in E
4  o ← number of occurrences of s in E
5  c_idxbuc ← K_idxbuc + 2oK_lookup + P_p.cost
6  if c_idxbuc < c_best then
7      disp[0] ← D[0]
8      buckets[0] ← 1
9      j ← 0
10     for i ← 1; i < q/p; i++ do
11         if D[ip] − D[(i − 1)p] = s then
12             buckets[j] ← buckets[j] + 1
13         else
14             j ← j + 1
15             disp[j] ← D[ip]
16             buckets[j] ← 1
17     P_q ← idxbuc(j + 1, s, buckets, disp, P_p)
18     c_best ← c_idxbuc
```

The approach is very similar to the one used for computing the representation $\mathcal{T}_{\mathsf{idxbuc}}$ for the construction of an optimal type tree (Section 5.3). The slightly adapted procedure is shown in Algorithm 16, which can directly be plugged into the type path reconstruction algorithm (Algorithm 15) by including it in the body of the inner-most foreach-loop, i.e., after line 28. The correctness of this approach is immediate from the observations made in Section 5.3 and Section 6.3. We however make no claim that this approach is particularly efficient and expect that the asymptotic runtime can be reduced significantly with the help of a clever preprocessing step similar to the idx and vec constructors (steps two and three in Algorithm 15). The main purpose of this section is to demonstrate that our algorithm can be extended to consider further constructors based on repeated prefixes.

Lemma 14. *The* Extended Type Path Reconstruction Problem *can be solved in* $O(n^2 \log^2 n)$ *time and* $O(n)$ *space.*

Proof. The asymptotic runtime of Algorithm 16 is dominated by the cost of sorting q over p elements on line 2. The remaining steps can all easily be implemented in linear time. The inner foreach-loop of Algorithm 15 is executed for each divisor p of a given q, whereas the outer foreach-loop is executed for each divisor q of n. The number of elements to be sorted in one iteration of Algorithm 16 is q/p. We can apply Gronwall's [Gro13] upper bound for the divisor summatory function to bound the number of elements to be sorted in one iteration of the outer loop as

$$\sum_{p|q} q/p \leq \sum_{p|q} p = O(q \log q / \log \log q) \quad .$$

Thus, $O(q \log q / \log \log q)$ elements have to be sorted to compute the optimal type path $\mathcal{P}_q$ for the prefix M_q, assuming that optimal representations $\mathcal{P}_p$ are already known for all prefixes M_p repeated in M_q. We find that in total the number of items that need to be sorted is

$$\sum_{q|n} O(q \log q \log \log q) \leq \sum_{q|n} O(q \log q) \leq \sum_{q=1}^{n} O(q \log q) \leq O(n^2 \log n) \quad .$$

Using any standard sorting algorithm that sorts n elements in $O(n)$ time, this is feasible in $O(n^2 \log^2 n)$ time, since

$$O(n^2 \log n \log(n^2 \log n)) = O(n^2 \log n (2 \log n + \log \log n)) = O(n^2 \log^2 n) \quad .$$

The space bound follows directly from the fact that the set E and the lists *disp* and *buckets* each contain at most n elements. □

CHAPTER 7

Problem variants

In this chapter, we discuss variants of the Type Reconstruction Problem and their potential advantages. In particular, we discuss the Type DAG Reconstruction Problem in Section 7.1, the Type Normalization Problem in Section 7.2 and the inclusion of additional constructors in Section 7.3. For most of these problems, only straight-forward solutions are known. However, we strongly suspect that more efficient solutions exist and this chapter can be seen as a collection of possible directions for future work.

7.1 Type DAG reconstruction

Recall the definition of type DAGs and the Type DAG Reconstruction Problem from Section 2.7.1. Contrary to the less general type tree structure, the nodes of a type DAG may have multiple predecessor nodes (or incoming edges), whereas a node in a type tree may have at most one parent. Basic concepts, such as the flattening procedure for type trees (Algorithm 2 in Section 2.7) can easily be adapted for type DAGs. As mentioned in Section 2.7.1, type DAGs may lead to arbitrarily more concise representations and some of the related work discussed in Chapter 3 may implicitly work with a DAG-like structure.

However, the principle of optimality (or dynamic programming principle) does not hold for type DAGs, as we show by example in the following. We first define the notation of a *sub-structure* for a type DAG. The definition is a bit more involved than for type trees, where a sub-structure is easily defined as a subtree. Intuitively, in a DAG the sub-structure attached to a node $\mathcal{N}$ consists of all the nodes reachable from $\mathcal{N}$ and all the edges between these nodes. Figure 7.1 illustrates this notation.

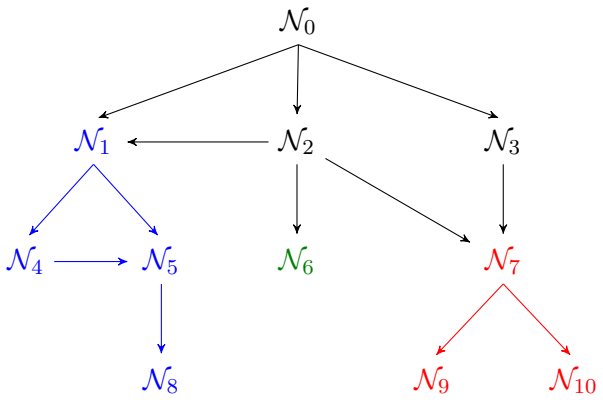
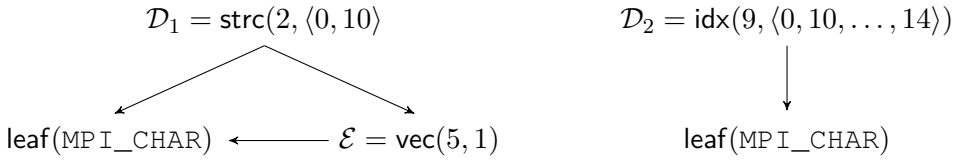


Figure 7.1: Examples of a sub-structures in a type DAG. The sub-structure attached at the node $\mathcal{N}_1$ consists of all the blue nodes and edges. The sub-structure attached at node $\mathcal{N}_6$ is highlighted in green and the sub-structure attached to $\mathcal{N}_7$ in red.



(a) Cost: 17.

(b) Cost: 14.

Figure 7.2: Two type DAG representations for the type map $M = (\langle \texttt{char}, ... \rangle, \langle 0, 10, 11, \ldots, 14 \rangle)$.

Definition 18. *Given a type DAG $\mathcal{D} = (V, E)$ with the node set $V = \{\mathcal{N}_0, \ldots, \mathcal{N}_{n-1}\}$ and the edge set E, the sub-structure $\mathcal{E} = (V'E')$ attached at node $\mathcal{O} \in V$ is defined as*

$$V' = \{\mathcal{N}_i \in V \mid \mathcal{N}_i \text{ is reachable from } \mathcal{O}\}$$
$$E' = \{(\mathcal{N}_i, \mathcal{N}_j) \in E \mid \mathcal{N}_i \in V' \wedge \mathcal{N}_j \in V'\}$$

A node $\mathcal{N}_i$ is reachable from a node $\mathcal{O}$ if there exists a path from $\mathcal{O}$ to $\mathcal{N}_i$.

We now give an example to show that a type DAG does not adhere to the principle of optimality. To keep the example simple, we do not consider idxbuc nodes in this section. One may assume that the cost K_{idxbuc} of an idxbuc node is very large, e.g., $K_{\mathsf{idxbuc}} = \infty$, so that a representation via an idxbuc node is never optimal.

Lemma 15. *An optimal type DAG may contain sub-structures that do not represent the corresponding type map optimally.*

Proof. Assume a type map that consists of 6 char values, 5 of which are stored consecutively, e.g., a type map $M = (\langle \texttt{char}, ... \rangle, \langle 0, 10, 11, \ldots, 14 \rangle)$. Figure 7.2 gives two representations for the type map M. The representation $\mathcal{D}_1$ in Figure 7.2a has a cost

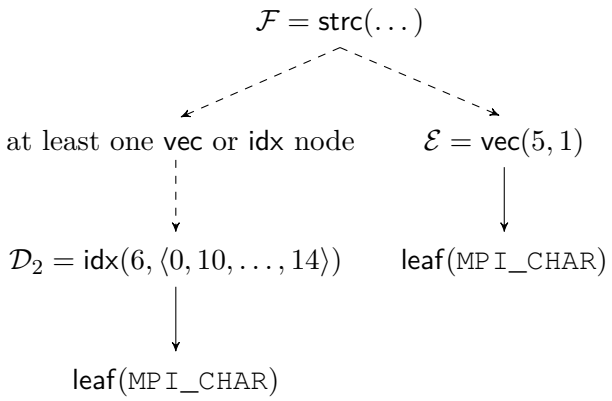


Figure 7.3: A type DAG using optimal representations $\mathcal{D}_2$ and $\mathcal{E}$ for the sub-problems M and N.

of $K_{\mathsf{strc}} + 2K_{\text{lookup}} + K_{\mathsf{leaf}} + K_{\mathsf{vec}} = 17$, while the representation $\mathcal{D}_2$ in Figure 7.2b has a smaller cost of $K_{\mathsf{idx}} + 6K_{\text{lookup}} + K_{\mathsf{leaf}} = 14$. The representation $\mathcal{D}_2$ is optimal for the type map M. The representation $\mathcal{D}_1$ contains the sub-structure $\mathcal{E}$, which is an optimal representation for the sub-problem $N_{1,5}$ of M, i.e., the aligned type map $N = (\langle \texttt{char}, \dots\rangle, \langle 0, 1, \dots, 4\rangle)$.

We construct a larger type DAG $\mathcal{F}$, shown in Figure 7.3, that contains the type DAGs $\mathcal{D}_2$ and $\mathcal{E}$ as sub-structures. The dashed lines indicate the general structure of $\mathcal{F}$. We require $\mathcal{F}$ to follow these four properties but do not care about its concrete form:

- The sub-structures $\mathcal{D}_1$ and $\mathcal{E}$ are joined in $\mathcal{F}$ by a strc node.
- There is at least one idx or vec node between the two nodes $\mathcal{D}_2$ and $\mathcal{F}$. This is to ensure that the sub-structure representing the sub-problem M cannot be directly merged into a strc node, even if the start node of the representation is a strc node.
- The type DAG $\mathcal{D}_2$ occurs only once in $\mathcal{F}$.
- The type DAG $\mathcal{D}_1$ does not occur in $\mathcal{F}$.

Assume that type DAGs adhere to the principle of optimality. We contradict this assumption in the following by showing that a type DAG of less cost can be constructed by replacing the optimal sub-structure $\mathcal{D}_2$ with the non-optimal sub-structure $\mathcal{D}_1$. Since the representations $\mathcal{D}_2$ and $\mathcal{E}$ are optimal for the respective sub-problems M and N, they may in principle occur in an optimal type DAG of the form outlined in Figure 7.3 (they do not necessarily occur, since other representations that do not make use of these two sub-problems may exist.) The total cost of the sub-structures $\mathcal{D}_2$ and $\mathcal{E}$ is $K_{\mathsf{idx}} + 6K_{\text{lookup}} + K_{\mathsf{leaf}} + K_{\mathsf{vec}} + K_{\mathsf{leaf}} = 22$.

However, we can reduce the cost of this type DAG by replacing the representation $\mathcal{D}_2$ with the (initially) more costly representation $\mathcal{D}_1$ of the same sub-problem M. This allows to represent the second occurrence of the sub-structure $\mathcal{E}$ with a single edge

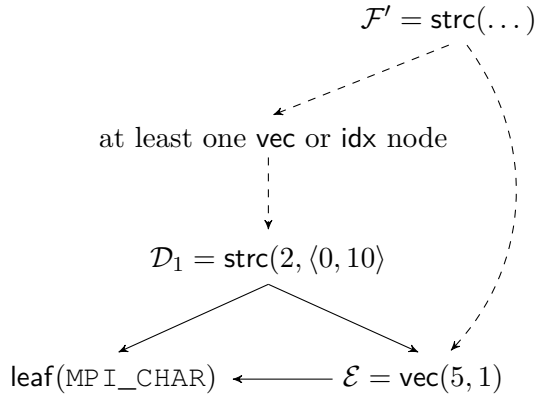


Figure 7.4: Using the non-optimal representation $\mathcal{D}_1$ for the sub-problem M reduces the overall cost of the DAG $\mathcal{F}$, since in $\mathcal{F}'$ parts of $\mathcal{D}_1$ can be reused to represent the sub-problem N.

to its first occurrence, as shown in Figure 7.4. In the modified type DAG $\mathcal{F}'$, the cost of the representations $\mathcal{D}_1$ and $\mathcal{E}$ for the sub-problems M and N is reduced to $K_{\mathsf{strc}} + 2K_{\text{lookup}} + K_{\mathsf{vec}} + K_{\mathsf{leaf}} = 17$. Thus, the representation $\mathcal{F}$, although using an optimal representation for sub-problem M, is more costly than the representation $\mathcal{F}'$, which uses a non-optimal representation for the same sub-problem. This contradicts the assumptions that type DAGs exhibit optimal sub-structure, i.e., the dynamic programming principle does not apply. □

The dynamic programming principle is one of the key ideas for our algorithm for the Type Reconstruction Problem. Further observations, like the nice type tree property (Lemma 1) and the construction of optimal type trees for non-aligned type maps (Lemma 8), build on this principle. It seems that our ideas for the Type Reconstruction Problem are not (easily) adaptable for the Type DAG Reconstruction Problem and it remains an open question whether or not this problem can be solved in polynomial time. Due to the potentially large benefits of type DAGs in comparison to type trees, this problem is certainly relevant. Further efforts are required to determine the computational hardness of the Type DAG Reconstruction Problem. Thus, heuristic approaches that target type DAGs specifically are a further topic of interest.

7.2 Type normalization

Recall the definition of the Type Normalization Problem form Section 2.7. Given a type tree $\mathcal{T}$, the task is to compute an optimal representation for the type map represented by $\mathcal{T}$.

A straight-forward solution for the Type Normalization Problem is to flatten the given type tree $\mathcal{T}$ and perform type reconstruction on the resulting type map. The same strategy works for type paths and type DAGs.

(a) Cost: 17. (b) Cost: 14.

Figure 7.5: Two type path representations for a type map $M = (\langle \texttt{char}, \ldots \rangle, \langle 0, s, 2s, \ldots \rangle)$ of arbitrary length n. For Träff's type path normalization algorithm [Trä14], the representation $\mathcal{P}_1$ given in Figure 7.5a constitutes the best case, whereas the representation $\mathcal{P}_2$ from Figure 7.5b constitutes the worst case. A type path of height d with the longest occurring displacement sequence being of length l is normalized in $O(dl\sqrt{l} + (d \log l)^3)$ time. The representation $\mathcal{P}_1$ does not contain any displacement sequence and the depth is constant w.r.t. n. It can thus be normalized in constant time. The algorithm, however, requires $O(n\sqrt{n})$ time to normalize the representation $\mathcal{P}_2$, since the longest occurring displacement sequence is of length n. Our approach, which is to flatten the input type path and perform type path reconstruction on the resulting type map, requires $O(n \log n / \log \log n)$ time in both cases. Flattening the type paths $\mathcal{P}_1$ and $\mathcal{P}_2$ can trivially be done in $O(n)$ time. Our approach is thus asymptotically faster for the representation $\mathcal{P}_2$, but slower for the representation $\mathcal{P}_1$.

So far, we have not given runtime bounds for the flattening procedure (Algorithm 2). It is in general not possible to give an asymptotic upper bound w.r.t. the length n of the represented type map, since a (non-optimal) type tree representation may contain arbitrarily many nodes (refer to Section 2.5). For the following, we assume that the type trees are such that flattening can be performed in $O(n \log n / \log \log n)$ time, i.e., that the flattening step does not dominate the complexity of the outlined approach for type path normalization. This assumption is reasonable since any type map $M = (T, D)$ of length n can be represented as $\mathsf{strc}(n, D, T)$, which can be flattened in $O(n)$ time.

To the best of our knowledge, previous attempts to solve the Type Normalization Problem exist only for type paths consisting solely of vec, idx and leaf nodes [Trä14]. Apart from heuristic approaches (see Chapter 3 for details), we were not able to find solutions for normalizing type trees or type DAGs. Thus, the straight-forward approach using our algorithm for the Type Reconstruction Problem is currently the most efficient for type trees and no algorithm for normalizing type DAGs exists as of yet.

The straight-forward approach outlined above requires to explicitly generate the full type map $M = (T, D)$ represented by $\mathcal{T}$. If M is of length n, $O(n)$ memory is required to store M. However, a type tree (type path, type DAG) representation may be arbitrarily more concise than the represented type map. For example, the simple type path $\mathcal{P}_1 = \mathsf{vec}(n, s, \mathsf{leaf}(\texttt{MPI\_CHAR}))$ can easily be stored in a constant amount of memory, but the type map requires $O(n)$ memory, i.e., M may be arbitrarily larger than $\mathcal{T}$. This cost has to be paid even if the given representation was already optimal. Träff [Trä14] presented an algorithm that performs normalization on type paths in time

proportional to the height d of the type path and the length l of the longest occurring displacement sequence in any node of the input or output type tree. In particular, the algorithm requires $O(dl\sqrt{l} + (d \log l)^3)$ time. Type paths that consist only of vec and leaf nodes (e.g., $\mathcal{P}_1$) are normalized in $O(d^3)$ time since no displacement arrays are present. If the given type path contains no (or only a constant number of) useless nodes, i.e., vec and idx nodes with count 1, this is equivalent to $O(\log^3 n)$, since the height of such a path is $O(\log n)$ (Lemma 2). The worst case for this algorithm occurs when a node in the given type path contains an array of displacements of size n, e.g., for $\mathcal{P}_2 = \mathsf{idx}(n, D, \mathsf{leaf}(\texttt{MPI\_CHAR}))$. In this case, the algorithm requires $O(n\sqrt{n})$ time, whereas flattening and constructing an optimal type path with our algorithm presented in Chapter 6 requires $O(n \log n / \log \log n)$ time. In other words, there are relevant cases where the algorithm by Träff is asymptotically slower than ours. However, our approach always hits the worst case.

We note that Träff's algorithm can be improved by using our ideas for computing repeated prefixes efficiently (Section 6.2). Their current approach performs straight-forward checks to see if a given prefix is repeated. Using our efficient method to pre-compute all repeated prefixes, the worst case bound of Träff's algorithm can be improved to match our bound. However, their approach does not seem to allow for an (easy) integration of further constructors, e.g., the idxbuc constructor. As discussed in Section 6.1, their algorithm is based on the canonical path property. It is not clear how this property can be extended to cover further constructors. The dynamic programming step of combining and splitting nodes, which already consists of ten cases if only the leaf, idx and vec nodes are considered, will be a lot more involved for every additional constructor.

Unfortunately, so far we were not able to find a way to utilize the ideas presented in Chapters 4 to 6 for an algorithm that performs type normalization more efficiently than the straight-forward approach outlined above.

7.3 Additional constructors

Several new type constructors have been proposed in addition to the ones defined by the MPI standard. The proposals usually stem from a desire to make the usage of derived datatypes easier and more transparent. Furthermore, they typically capture occurring data layouts more efficiently than the available type constructors. We provide a brief overview of the proposed constructors and argue informally about their inclusion in our algorithms for the Type Reconstruction Problem and the Type Path Reconstruction Problem.

- A *bounded vector* [TR14] is a slight generalization of the `VECTOR` constructor. Recall that the `VECTOR` constructor replicates a given sub-type `blocklength` many times into each of the `count` many blocks. The idea is to give the total number of replications of the sub-type instead of the number of blocks. Thus, the type map resulting from the standard `VECTOR` constructor is cut off when a

certain number of replications is reached, i.e., the last block may contain less than `blocklength` elements. Träff et al. [TRH14] show that this constructor is useful in the implementation of the classic Bruck algorithm [BHK+97] for the all-to-all collective communication operation (`MPI_ALLTOALL` [MPI15, p. 168]).

- A *circular vector* [TR14] further generalizes the bounded vector constructor. The basic idea is to wrap the type map generated by a bounded vector. After a certain number of replications of the sub-type, the remaining replications are at smaller displacements, i.e., the type map "wraps around" a given bound. This constructor too is useful for implementing the Bruck algorithm [TRH14].

- The *bucket type* [TR14] is the natural complement to the `INDEXED_BLOCK` constructor. Instead of an array of displacements, it takes an array of block lengths as argument. The basic idea is that a memory area is divided in contiguous blocks (or buckets) of a fixed length, called the bucket size. The number of elements in each bucket is given by the corresponding bucket length value. This constructor is also useful for the Bruck algorithm [TRH14], in particular for irregular all-to-all communication (`MPI_ALTOOALLV` [MPI15, p. 170]).

- The *triangular type* [TLRH15] describes a type map of regularly changing blocks. The blocks change in two dimensions: The i-th block contains `blockincrement` many replications more than its predecessor block. Its stride is equal to the initial `stride` plus the `strideincrement` value, i.e., the i-th block starts at displacement $i(\texttt{stride} + \texttt{strideincrement})$. This constructor is useful for the implementation of stencil computations [TLRH15].

The bounded and circular vector constructors can easily be expressed with existing MPI datatype constructors, although for the latter it is necessary to take care of multiple tedious special cases. Type maps specified by the bucket type constructor can trivially be represented by the built-in `INDEXED` constructor. However, since the displacements of the blocks are all multiples of the bucket size, listing them explicitly is redundant and incurs a large memory overhead. Instead of one array (the actual number of elements in each bucket) and a scalar value (the bucket size), two arrays (displacements and block lengths) are required to specify the same type map with the `INDEXED` constructor. The triangular type constructor can represent some of the type maps occurring in stencil computations with constant cost, whereas a representation with the `INDEXED` constructor would require space proportional to the number of blocks.

Träff and Rougier [TR14] observe that the bounded vector constructor in particular was beneficial for the overall performance. They argue that an integration into the MPI datatype engine will lead to further performance improvements. To handle the new constructors efficiently, they need to be considered for type normalization and reconstruction. We argue informally that the proposed constructors can be integrated into our algorithm for the Type Reconstruction Problem. To include one of the proposed constructors in Algorithm 9, one has to check whether the constructor can represent a given type map with a given sub-type. Recall that the key challenge for this

algorithm was the inclusion of the strc constructor, which is the only constructor taking more than one sub-type. As with the vec, idx and idxbuc constructors, straight-forward approaches for the new constructors suffice as long as they require $O(n^2)$ time to perform one such check. Since the proposed constructors all follow the basic idea of the idxbuc or vec constructors to replicate a sub-type into (semi-) regular locations, this should be possible. For a formal proof, the properties given in Chapter 4 need to be extended to also cover the new constructors. Since all of the proposed constructors take only one sub-type, key characteristics such as the nice type tree property (Lemma 1) or the bound on the height of a type tree (Lemma 2) should still hold.

Constructors that rely on repeated prefixes can be incorporated into our algorithm for the Type Path Reconstruction Problem (Algorithm 15) similarly to the idxbuc constructor (see Section 6.5). This includes the triangular and bucket types as well as the bounded and vector types. Note that for the last two, only the sub-type but not the block is required to be a repeated prefix. As for the idxbuc constructor, further ideas are required to reduce the asymptotic runtime of the resulting algorithm. The $O(n \log n / \log \log n)$ time bound may be hard to achieve, since the proposed constructors take more parameters and thus more possible representations of a type map need to be checked. However, we strongly believe that the basic algorithm for handling only the vec and idx constructors provides a solid basis for this endeavor.

CHAPTER 8

Conclusion

In this concluding chapter, the main results of this master's thesis are summarized briefly. We furthermore put our results into context with related work and state possible directions for future research. We point out that the previous chapter, which discusses several interesting variations of the Type Reconstruction Problem, contains multiple open problems as well as a detailed comparison of our result for the Type Path Reconstruction Problem with the best previously known solution.

8.1 Results of this work

This master's thesis explored the Type Reconstruction Problem, that is, the problem of constructing provably optimal *type tree* representations for arbitrarily complex, heterogeneous and non-contiguous data layouts with the help of MPI's derived datatype constructors.

The master thesis starts with an intuitive introduction to the problem. In particular, we give an overview of MPI and its derived datatypes mechanism to motivate and put into context the Type Reconstruction Problem. The introductory chapter is followed by a detailed and precise presentation of MPI's derived datatype functionality, before the formal model for data layouts and type trees as well as the assumed cost model are introduced. This is followed by a formal definition of the following problems:

- The Type Reconstruction Problem, which asks for a least-cost type tree representation of a given data layout.
- The Type Path Reconstruction Problem, which is a special case of the former problem where type trees are restricted to type paths.
- The Type DAG Reconstruction Problem, for which the structure of type trees is generalized to type DAGs.

- The Type Normalization Problem, which requires to construct a least-cost type tree representation out of a given type tree.

A careful and detailed analysis of the structure of (optimal) type trees is provided in Chapter 4. The properties proved in this part of the master's thesis are crucial for the algorithms developed in the following two chapters and may provide a good starting point for future, possibly more efficient algorithms. We deem this chapter most important for a thorough understanding of type trees and the reconstruction and normalization problems.

The following chapter presents in detail our algorithm that solves the Type Reconstruction Problem in polynomial time, i.e., in $O(n^4)$ time and $O(n^2)$ space, where n is the size of the given data layout. This refutes the conjectured NP-hardness of the problem [GHTT11, Trä14] and to the best of our knowledge is the first successful attempt to solve the problem to optimality. The result was submitted to the IPDPS conference [GKST] and a preliminary version is available online [GKST15].

Our second main contribution is an algorithm solving the special case of the Type Path Reconstruction Problem in asymptotically less time than previous approaches. In particular, our algorithm requires $O(n \log / \log \log n)$ time and $O(n)$ space to construct an optimal type path for a data layout of length n. The previously best known solution requires $O(n\sqrt{n})$ time. Our approach is furthermore able to integrate additional type constructors. Although this increases the asymptotic bound significantly, our algorithm is the first to allow for a relatively easy extension to further constructors. This result was published at this year's EuroMPI conference [KT15].

As a minor contribution, we show in Chapter 7 that the principle of optimality, which is crucial for our type tree and type path reconstruction algorithms, does not hold when the structure of the representation is generalized to a DAG. The chapter furthermore contains a discussion about closely related, interesting problem variations and the applicability of our results to the Type Normalization Problem.

8.2 Comparison to related work

A detailed comparison of our solution for the Type Path Reconstruction Problem and the resulting straight-forward approach to the Type Path Normalization Problem is already given in Section 7.2. The heuristic optimization techniques outlined in Chapter 3 are performed implicitly by our algorithms as far as they are applicable in our model, assuming that the chosen cost values reflect the assumed advantages of these optimizations.

- If a data layout can be represented with less cost with a more specialized constructor, the algorithm will choose the latter. Heuristic optimizations typically assume that a representation with the `STRUCT` constructor is the most costly. Optimizations are performed by specializing along the following hierarchy (see, e.g., [RMG03, SKH13, KHS12]):

  - `STRUCT` → `HINDEXED`. Ross et al. [RMG03] assume that this specialization is always possible for homogeneous data layouts without decreasing the quality of the solution. As shown by example in Section 2.6, this is not necessarily true. Our algorithm for the Type Reconstruction Problem treats this case correctly, while the algorithm for the Type Path Reconstruction Problem does not consider the `STRUCT` constructor by definition, i.e., it implicitly performs this "optimization".

  - `HINDEXED` → `HVECTOR` → `VECTOR` → `CONTIGUOUS`. Recall that the MPI standard often defines two versions of the same constructor, with one measuring displacements and strides in multiples of the extent of the base type while the other uses bytes as the base unit. In our model, we measure all these arguments in bytes and therefore both versions are modeled with a single constructor. Furthermore, the `CONTIGUOUS` constructor is modeled as a vec constructor with count 1. Apart from that, the same optimizations are performed if the cost model is chosen s.t. the cost of a vec constructor is less than the cost of an *idx* constructor etc.

- Kjolstad et al. [KHS12] merge two consecutive `STRUCT` constructors into a single one. As shown in Lemma 8, an optimal type tree cannot contain two directly nested strc nodes and thus our algorithm is guaranteed to not produce such a representation.

- Schneider et al. [SKH13] and Kjolstad et al. [KHS12] furthermore merge contiguous sub-types into the parent type with the help of the `blocklength` argument that most MPI type constructors take. In our simplified model, only the idxbuc constructor still takes this argument, while vec, strc and idx, contrary to their MPI counterparts, do not. This optimization is thus not applicable in our model.

- Contiguous indexed regions are coalesced by Ross et al. [RMG03] This is done by our algorithms only if it reduces the cost of the generated representation. It may very well be the case that a representation of less cost is possible if parts of a contiguous region are represented by multiple constructors, e.g., if each part can be represented efficiently with the vec constructor.

- Kjolstad et al. [KHS12] save the cost of the `CONTIGUOUS` constructor if it is the top-most constructor in a derived datatype. In particular, instead of communicating one element of a contiguous datatype with count c and a certain sub-type, the operation is changed to communicate c elements of the sub-type. Although this results in a semantically equivalent operation, the represented type map is not the same. Such an optimization is thus out of the scope of datatype reconstruction and normalization, but can easily be performed in addition.

8.3 Open issues and future work

Due to its $O(n^4)$ runtime, our algorithm for the Type Reconstruction Problem may not be all that helpful in practice, except for small n. To improve this asymptotic bound, the step of constructing the least-cost representation $\mathcal{T}_{\mathsf{strc}}$ in Algorithm 9 has to be improved. We currently bound the runtime of this step as $O(n^2)$ for all sub-problems of length l, $l \leq n$. This is clearly a very rough upper bound which may be improved significantly by a careful investigation and analysis of the algorithm. This step is dominated by the $O(n^2)$ time required to find a shortest path in a DAG with $O(n)$ nodes and $O(n^2)$ edges. As discussed at the end of Section 5.5, we can subsume the computations of a shortest path for each of the $n - l + 1$ sub-problems of length l by one instance of an all pairs shortest path problem. This, however, does not reduce the asymptotic runtime. Small changes to the current algorithm and/or a more careful analysis may reduce this bound significantly.

Our algorithm for the Type Path Reconstruction Problem runs in near-linear time, which we deem acceptable for practical purposes. However, the bound increases to $O(n^2 \log^2 n)$ time when also the idxbuc constructor is considered. Our approach for the inclusion of this constructor is straight-forward and we expect that this bound may be improved drastically.

Several directions for future research are outlined in Chapter 7, where we discuss variations of the Type Reconstruction Problem. In particular, the following questions require further investigation:

- Does there exist a polynomial-time algorithm for the Type DAG Reconstruction Problem or is this problem NP-hard?
- Can our ideas be used for a type tree (or type path) normalization algorithm that is more efficient than performing type reconstruction on the flattened type tree?
- As discussed in Section 7.3, additional constructors not currently defined by the MPI standard should be easy to integrate into our algorithm for the Type Reconstruction Problem. However, a formal proof of correctness has to include the additional constructors in the proofs of all the properties discussed in Chapter 4, which requires a non-trivial effort.
- Can the additional constructors be included in our algorithm for the Type Path Reconstruction Problem, without increasing its asymptotic runtime (too much)?

The open questions outlined so far are of a purely algorithmic nature. For a practically useful type reconstruction algorithm, the following more practical issues need to be investigated:

- The cost model adapted for this work is very general. While it is simple enough for a formal analysis and flexible enough to reasonable model costs in different settings, it is not detailed enough to capture the costs as they occur in practice. The current

cost model for example does not consider memory hierarchies, which often play a crucial role for high performance applications.

- An experimental evaluation of our algorithm for the Type Path Reconstruction Problem is required to determine its applicability in practice. This requires an integration of the algorithm in an MPI implementation, for which our proof of concept implementation may be a helpful starting point.

Bibliography

[BGST03] Surendra Byna, William D. Gropp, Xian-He Sun, and Rajeev Thakur. Improving the performance of MPI derived datatypes by optimizing memory-access cost. In *2003 IEEE International Conference on Cluster Computing (CLUSTER 2003), 1-4 December 2003, Kowloon, Hong Kong, China*, pages 412–419. IEEE Computer Society, 2003.

[BHK+97] Jehoshua Bruck, Ching-Tien Ho, Shlomo Kipnis, Eli Upfal, and Derrick Weathersby. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Trans. Parallel Distrib. Syst.*, 8(11):1143–1156, 1997.

[BT11] Enes Bajrovic and Jesper Larsson Träff. Using MPI derived datatypes in numerical libraries. In Cotronis et al. [CDND11], pages 29–38.

[CDND11] Yiannis Cotronis, Anthony Danalis, Dimitrios S. Nikolopoulos, and Jack Dongarra, editors. *Recent Advances in the Message Passing Interface - 18th European MPI Users' Group Meeting, EuroMPI 2011, Santorini, Greece, September 18-21, 2011. Proceedings*, volume 6960 of *Lecture Notes in Computer Science*. Springer, 2011.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[DDG+15] Jack Dongarra, Alexandre Denisand, Brice Goglin, Emmanuel Jeannot, and Guillaume Mercier, editors. *22nd European MPI Users' Group Meeting, EuroMPI '15, Bordeaux, France - September 21 - 23, 2015*. ACM, 2015.

[DIH14] Jack Dongarra, Yutaka Ishikawa, and Atsushi Hori, editors. *21st European MPI Users' Group Meeting, EuroMPI/ASIA '14, Kyoto, Japan - September 09 - 12, 2014*. ACM, 2014.

[Fly72] Michael Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 100(9):948–960, 1972.

[GHTT11] William Gropp, Torsten Hoefler, Rajeev Thakur, and Jesper Larsson Träff. Performance expectations and guidelines for MPI derived datatypes. In Cotronis et al. [CDND11], pages 150–159.

[GKR+08] William D. Gropp, Dries Kimpe, Robert B. Ross, Rajeev Thakur, and Jesper Larsson Träff. Self-consistent MPI-IO performance requirements and expectations. In Lastovetsky et al. [LKD08], pages 167–176.

[GKST] Robert Ganian, Martin Kalany, Stefan Szeider, and Jesper Larsson Träff. Polynomial-time construction of optimal mpi derived datatype trees. *Submitted to IPDPS*.

[GKST15] Robert Ganian, Martin Kalany, Stefan Szeider, and Jesper Larsson Träff. Polynomial-time construction of optimal tree-structured communication data layout descriptions. *CoRR*, abs/1506.09100, 2015.

[GLS99a] William Gropp, Ewing Lusk, and Deborah Swider. Improving the performance of MPI derived datatypes. In *Third MPI Developer's and User's Conference (MPIDC'99)*, pages 25–30, 1999.

[GLS99b] William D. Gropp, Ewing L. Lusk, and Anthony Skjellum. *Using MPI - portable parallel programming with the message-parsing interface. Second edition*. MIT Press, 1999.

[Gro13] Thomas Hakon Gronwall. Some asymptotic expressions in the theory of numbers. *Transactions of the American Mathematical Society*, 14(1):113–122, 1913.

[HG10] Torsten Hoefler and Steven Gottlieb. Parallel zero-copy algorithms for fast fourier transform and conjugate gradient using MPI datatypes. In Keller et al. [KGRD10], pages 132–141.

[IEE08] IEEE Computer Society. IEEE Standard for Floating-Point Arithmetic. IEEE Standard 754-2008, August 2008.

[JDB+14] John Jenkins, James Dinan, Pavan Balaji, Tom Peterka, Nagiza F. Samatova, and Rajeev Thakur. Processing MPI derived datatypes on noncontiguous GPU-resident data. *IEEE Trans. Parallel Distrib. Syst.*, 25(10):2627–2637, 2014.

[KGR10] Dries Kimpe, David Goodell, and Robert B. Ross. MPI datatype marshalling: A case study in datatype equivalence. In Keller et al. [KGRD10], pages 82–91.

[KGRD10] Rainer Keller, Edgar Gabriel, Michael M. Resch, and Jack Dongarra, editors. *Recent Advances in the Message Passing Interface - 17th European MPI Users' Group Meeting, EuroMPI 2010, Stuttgart, Germany, September 12-15, 2010. Proceedings*, volume 6305 of *Lecture Notes in Computer Science*. Springer, 2010.

[KHS11] Fredrik Kjolstad, Torsten Hoefler, and Marc Snir. A transformation to convert packing code to compact datatypes for efficient zero-copy data transfer.

Technical report, University of Illinois at Urbana-Champain, 2011. Retrieved from https://www.ideals.illinois.edu/handle/2142/26452, last visited on 06/29/2015.

[KHS12] Fredrik Kjolstad, Torsten Hoefler, and Marc Snir. Automatic datatype generation and optimization. In J. Ramanujam and P. Sadayappan, editors, *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012*, pages 327–328. ACM, 2012.

[Knu81] Donald E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition.* Addison-Wesley, 1981.

[KT15] Martin Kalany and Jesper Larsson Träff. Efficent, Optimal MPI Datatype Reconstruction for Vector and Index Types. In Dongarra et al. [DDG+15].

[Lan09] E Landau. *Handbuch der Lehre von der Verteilung der Primzahlen, Band 1.* Teubner, 1909.

[LKD08] Alexey L. Lastovetsky, M. Tahar Kechadi, and Jack Dongarra, editors. *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 15th European PVM/MPI Users' Group Meeting, Dublin, Ireland, September 7-10, 2008. Proceedings*, volume 5205 of *Lecture Notes in Computer Science*. Springer, 2008.

[LWPS04] Qingda Lu, Jiesheng Wu, Dhabaleswar K. Panda, and P. Sadayappan. Applying MPI derived datatypes to the NAS benchmarks: A case study. In *33rd International Conference on Parallel Processing Workshops (ICPP 2004 Workshops), 15-18 August 2004, Montreal, Quebec, Canada*, pages 538–545. IEEE Computer Society, 2004.

[MPI15] MPI Forum. MPI: A Message-Passing Interface Standard. Version 3.1, June 2015.

[MT08] Faisal Ghias Mir and Jesper Larsson Träff. Constructing MPI input-output datatypes for efficient transpacking. In Lastovetsky et al. [LKD08], pages 141–150.

[PG15] Tarun Prabhu and William Gropp. DAME: A Runtime-Compiled Engine for Derived Datatypes. In Dongarra et al. [DDG+15].

[RMG03] Robert B. Ross, Neill Miller, and William Gropp. Implementing fast and reusable datatype processing. In Jack Dongarra, Domenico Laforenza, and Salvatore Orlando, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface,10th European PVM/MPI Users' Group Meeting, Venice, Italy, September 29 - October 2, 2003, Proceedings*, volume 2840 of *Lecture Notes in Computer Science*, pages 404–413. Springer, 2003.

[Rob83] Guy Robin. Estimation de la fonction de tchebychef θ sur le k-ième nombre premier et grandes valeurs de la fonction ω (n) nombre de diviseurs premiers de n. *Acta Arithmetica*, 42(4):367–389, 1983.

[RP06] Eric Renault and Christian Parrot. MPI pre-processor: Generating MPI derived datatypes from C datatypes automatically. In *2006 International Conference on Parallel Processing Workshops (ICPP Workshops 2006), 14-18 August 2006, Columbus, Ohio, USA*, pages 248–256. IEEE Computer Society, 2006.

[SGH12] Timo Schneider, Robert Gerstenberger, and Torsten Hoefler. Micro-applications for communication data access patterns and MPI datatypes. In Jesper Larsson Träff, Siegfried Benkner, and Jack J. Dongarra, editors, *Recent Advances in the Message Passing Interface - 19th European MPI Users' Group Meeting, EuroMPI 2012, Vienna, Austria, September 23-26, 2012. Proceedings*, volume 7490 of *Lecture Notes in Computer Science*, pages 121–131. Springer, 2012.

[SKH13] Timo Schneider, Fredrik Kjolstad, and Torsten Hoefler. MPI datatype processing using runtime compilation. In Jack Dongarra, Javier García Blas, and Jesús Carretero, editors, *20th European MPI Users's Group Meeting, EuroMPI '13, Madrid, Spain - September 15 - 18, 2013*, pages 19–24. ACM, 2013.

[SWP04] Gopalakrishnan Santhanaraman, Dhabaleswar Wu, and Dhabaleswar K. Panda. Zero-copy MPI derived datatype communication over infiniband. In Dieter Kranzlmüller, Péter Kacsuk, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, September 19-22, 2004, Proceedings*, volume 3241 of *Lecture Notes in Computer Science*, pages 47–56. Springer, 2004.

[TGT10] Jesper Larsson Träff, William D. Gropp, and Rajeev Thakur. Self-consistent MPI performance guidelines. *IEEE Trans. Parallel Distrib. Syst.*, 21(5):698–709, 2010.

[THRZ99] Jesper Larsson Träff, Rolf Hempel, Hubert Ritzdorf, and Falk Zimmermann. Flattening on the fly: Efficient handling of MPI derived datatypes. In Jack Dongarra, Emilio Luque, and Tomàs Margalef, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 6th European PVM/MPI Users' Group Meeting, Barcelona, Spain, September 26-29, 1999, Proceedings*, volume 1697 of *Lecture Notes in Computer Science*, pages 109–116. Springer, 1999.

[TLRH15] Jesper Larsson Träff, Felix Lübbe, Antoine Rougier, and Sascha Hunold. Isomorphic, sparse MPI-like collective communication operations for parallel

stencil computations. In *22nd European MPI Users' Group Meeting, EuroMP '15, Bordeaux, France - September 21 - 24, 2015*, 2015.

[TR14] Jesper Larsson Träff and Antoine Rougier. Zero-copy, hierarchical gather is not possible with MPI datatypes and collectives. In Dongarra et al. [DIH14], page 39.

[Trä09] Jesper Larsson Träff. *Aspects of the efficient implementation of the message passing interface (MPI)*. Shaker, 2009.

[Trä14] Jesper Larsson Träff. Optimal MPI datatype normalization for vector and index-block types. In Dongarra et al. [DIH14], page 33.

[TRH14] Jesper Larsson Träff, Antoine Rougier, and Sascha Hunold. Implementing a classic: zero-copy all-to-all communication with mpi datatypes. In Arndt Bode, Michael Gerndt, Per Stenström, Lawrence Rauchwerger, Barton P. Miller, and Martin Schulz, editors, *2014 International Conference on Supercomputing, ICS'14, Muenchen, Germany, June 10-13, 2014*, pages 135–144. ACM, 2014.

[TT08] Wesley Tansey and Eli Tilevich. Efficient automated marshaling of C++ data structures for MPI applications. In *22nd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, Miami, Florida USA, April 14-18, 2008*, pages 1–12. IEEE, 2008.

[WWP04] Jiesheng Wu, Pete Wyckoff, and Dhabaleswar K. Panda. High performance implementation of MPI derived datatype communication over infiniband. In *18th International Parallel and Distributed Processing Symposium (IPDPS 2004), CD-ROM / Abstracts Proceedings, 26-30 April 2004, Santa Fe, New Mexico, USA*. IEEE Computer Society, 2004.

[zar] *z/Architecture: Principles of Operation*. `http://publibz.boulder.ibm.com/epubs/pdf/dz9zr001.pdf` accessed 07/09/2015.