Bielefeld University
Faculty of Technology

Bachelor Thesis

Implementation of Inverse Coupled Rewrite Systems

a case study

Author:
Jonas Betzendahl
Program:
Cognitive Informatics

Supervisors:
Prof. Dr. Robert Giegerich
Dr. Stefan Janssen

August 28, 2014

The end of all our exploring will be to arrive where we started and know the place for the first time.

– T. S. Eliot

Contents

1 Introduction

The aim of this thesis is to a look at the framework of **I**nverse **Co**upled **Re**write **S**ystems (short form: **ICORES**) as introduced by Giegerich and Touzet[3] and discuss for the first time implementations in the programming language `Haskell`.

ICORES were introduced as a new high-level approach of defining and tackling a variety of dynamic programming problems on all kinds of either sequential or tree-like input types. It is meant as a clear and declarative way of specifying dynamic programming problems which are often well understood in informatics but have applications far beyond that field.

In the future, given enough research and implementation effort, one could imagine a compiler for ICORES, translating problem statements in an automated fashion to functional programs in other programming languages which then can already be used in practice with little or no modification by a programmer's hand. A bold step beyond that would be a full compiler that automatically creates programs achieving excellent performance, with both multiple front- and back-ends to support a large number of platforms and fields of application.
Such a compiler would greatly reduce the need for problem-specific home-brew implementations of dynamic programming (which often turn out to be, as Giegerich an Touzet note, inefficient, hard to modify and with low levels of abstraction) and allow - for example - scientists with little programming experience but deep understanding of their problem domain to formulate only the ICORES in a highly abstracted form and have efficient executables be generated only from that.
As a result of the simplified process of problem stating and automated program generation resources (both computational and human) could be freed, re-use of software components between related problems could be increased. Also, such a compiler would create the opportunity to analyse, refine and combine solutions to problems without having to deal with implementation detail or computational theory.

This thesis provides first hints and lessons that could be used during the construction of such a compiler and discusses ICORES and their implementation.
Specifically, we will introduce the background to ICORES and the relevant theory of trees and dynamic programming involved in Chapter 2. We will also look at two ICORES from the field of sequence and tree alignment in more detail to convey a basic idea of what problem specifications as ICORES might look like. This will be followed in Chapter 3 by a discussion of the concrete implementations in `Haskell` (source code included on the enclosed CD) and a look at the results of performance tests in Chapter 4. Finally, in Chapter 5, we will draw conclusions from the previous chapters and will give an outlook towards the construction of the compiler.

2 Theoretical Foundations

2.1 Inverse Coupled Rewrite Systems

2.1.1 Definition

ICORES were introduced in 2014 by Giegerich and Touzet[3]. We will begin with a formal definition, followed by an explanation of key terms and concepts. The more advanced reader, who is already familiar with evaluation algebras and term rewriting, may skip ahead to section 2.1.6 (Coupled and Inverse Rewriting), where we will discuss the new approach this framework brings.

In the original paper[1], ICORES are defined as follows:

Definition 1. *Inverse Coupled Rewrite System. Let k be a positive natural number. An* ICORE[2] *of dimension k consists of*

- *a set V of variables*
- *a* core *signature ζ and k* satellite *signatures $\Sigma_1, \ldots, \Sigma_k$, such that $\zeta \cap (\Sigma_1 \cup \cdots \cup \Sigma_k) = \mathcal{A} \supseteq \varnothing$*
  (Function symbols of ζ are disjoint from all Σ_i, except for a possible shared alphabet.)
- *k term rewrite systems with disjoint signatures, $R_1, \ldots, R_k$, which all have the same left-hand sides in the* term algebra *$T(\zeta, V)$ (Those systems are called the* satellite rewrite systems.*)*
- *optionally a* tree grammar *$\mathcal{G}$ over the core signature ζ*
- *an* evaluation algebra *A for the core signature ζ. This includes an* objective function *Φ*

We will elaborate on all of this shortly but first for a quick overview. The general process of finding a solution to a dynamic programming problem with an ICORE is the following: efficiently find all core terms that rewrite to all k satellite inputs and then evaluate all those potential solutions according to an algebra (which can but does not have to perform optimisation).

For some examples in a proposed notation for ICORES, please refer to section 2.3.

[1] All definitions in this chapter are taken from Giegerich and Touzet[3].

[2] Even considering the obvious break of the acronym, the singular form of "ICORES" is usually referred to as "ICORE" as to enhance reading comprehension and avoid confusion.
There is also the acronym "**I** **C**an **O**nly **R**ead **E**quations" for a similar purpose.

2.1.2 Signatures & Term Algebras

Let $\mathcal{S}$ be a set of names or placeholders for data types (sorts). For every sort $\alpha \in \mathcal{S}$, there exists a set of up to countably infinite variables called V_α. The union of all V_α is simply given the name V.
Now, let F be an $\mathcal{S}$-sorted signature, that is to say, a set of function symbols, each $f \in F$ equipped with a sort declaration $\alpha_1 \times \cdots \times \alpha_n \to \alpha$. It holds that $\forall i\ \alpha_i \in \mathcal{S}$ and that $\alpha \in \mathcal{S}$. n is called the arity of f. Function symbols within F with an arity of 0 are referred to as "constant", sometimes a subset of these forms an alphabet usually denoted $\mathcal{A}$.

$T(\zeta, V)$ is representative of the set of all well-typed terms over F and V and is referred to as the *term algebra*. We furthermore use the notation $var(t)$ to refer to the set of variables used in a term $t \in T(F, V)$ and $sort(t)$ to refer to its sort. The set of ground terms of $T(F, V)$ (where $var(t) = \varnothing$) is simply denoted $T(F)$.

2.1.3 Rewrite Systems

A *term rewrite system*, call it R, is a finite set of rewrite rules. A *rewrite rule* is a pair of terms denoted $l \to r$, the arrow indicating the direction of the rewrite. It must hold that $sort(l) = sort(r)$ and $var(l) \supseteq var(r)$, meaning that the sort of the term is not changed through the rewrite and that variables are only ever consumed, never generated. Because of this, $r \to l$ usually is not a valid rewrite rule.

R is called a term rewrite system *with disjoint signatures* when F can be split into two subsets, call them ζ and Σ, so that the left-hand sides of each rewrite rule in R belongs to $T(\zeta, V)$ - called the *core signature* in this context- and the right-hand side belongs to $T(\Sigma, V)$ - the *satellite signature*. It is not unusual for core and satellite signatures to share an alphabet or function symbols (possibly even of different arity). Should this problem arise it is assumed by convention that the occurrence on the left-hand side of the rewrite rule belongs to ζ and the occurrence on the right-hand side belongs to Σ.

In some scenarios, we rewrite *modulo associativity*. This means that for any three trees or sequences A, B and C, an associative binary relation (usually denoting some kind of concatenation and furthermore usually denoted $\sim$) and a neutral element to that relation (usually denoted ε), the following holds:

$$(A \sim B) \sim C = A \sim (B \sim C)$$
$$A \sim \varepsilon = A = \varepsilon \sim A$$

This rewriting modulo associativity makes a few things in this framework either simpler or more convenient. Consider the example of strings, which can also be represented as a sequence of constant character symbols (the term "a∼s∼d∼f" being the representation of the string "asdf" etc.). Rewriting modulo associativity makes it easier to do pattern matching on expressions, the term "X∼a∼s∼d∼f∼Y" matching to any occurrence of "asdf" in a longer string and so on.

2.1.4 Tree Grammar

In some contexts, it is useful or even necessary, to restrict the set of terms that can be created. We want this to happen independently from the evaluation algebras, that is to say we explicitly want to make it impossible to generate invalid terms, rather than penalise them with a score of minus infinity for example. For this, a tree grammar (also called term grammar) can be used. Consider this tree grammar, call it $\mathcal{G}$, defined over F, as a tuple $\mathcal{G} = (F, N, S, P)$.

- F is the signature, as discussed above.
- N is a set of non-terminal symbols - to each of which is assigned a sort - such that $F \cap N = \varnothing$.
- S is the starting non-terminal symbol (also called *axiom*), with $S \in N$ and
- P is a list of *productions* of the form $X \to t$, where $X \in N, t \in T(F, N)$ and $sort(X) = sort(t)$

This gives rise to a specific language of $\mathcal{G}$, namely the language of all terms $t \in T(F)$ that can be produced from S using productions in P. We denote this language simply $L(\mathcal{G})$.

2.1.5 Evaluation Algebra

For any signature F, an *evaluation algebra*, contains for every sort symbol in F:

- an underlying data type
- a function into that type from each $f \in F$.

The value that is obtained by interpreting a core term t under an algebra A is denoted $A(t)$.

Furthermore, an algebra also contains an *objective function* (sometimes also called *choice function*) $\Phi : M \to M$ (M being the domain of multisets) over some carrier set in the respective algebra. Typically, we will either use minimisation or maximisation for Φ (when this makes sense int he context of the underlying data type), but there are a number of other possible functions that are also used frequently (e.g. identity).

The reason we want Φ to operate on multisets (that is: sets that can contain the same element more than just once) instead of sets is that we may want to make accurate statements about the size of the candidate space or evaluate all co-optimal solutions under a second algebra (e.g. "Of all the restaurants in town that offers a vegan option, we want the one closest to us."). Limiting Φ to sets would destroy information that could still be valuable later on.

Algebras can also be combined. The combination of to algebras A and B is called their *lexicographic product* $A * B$. See the paper from Giegerich an Touzet for a list of examples on this topic.

This process does not add any computing power per se (as one could always encode the properties of all algebras used into just one by hand), but it makes the development process faster and simpler by enabling re-use of components that are already known to work from different contexts.

2.1.6 Coupled & Inverse Rewriting

Mapping from concrete to abstract is always the easier way.
– Harald Ganzinger

In this section we are going to look at the most prominent innovations that the ICORE-framework brings: the notions of inverse and coupled rewriting.

First, we are going to discuss the process of *inverse* rewriting. This describes rewriting relations that map the "wrong" way, from the concrete solutions to the abstract problems rather than the other way around. This differs from the traditional way. Usually, rewrite systems supplied in this context would map from the problem to the desired solution.

Let us consider an example to make this concept more clear: think of the classical problem of finding the minimum Levenshtein distance [6]. Here, the satellite terms that serve as input are two strings. These two strings can be seen as the specific instance of the *problem*. The *solution* to that problem (the core term in our nomenclature) would be an (optimal) edit script that rewrites the one string into the other in terms of substitution, insertions and deletions.
Now, if we think of a program that solves Levenshtein, we would typically supply a rewrite relation that *starts* with the satellite structure and *produces* the core term. Rewrite rules would be formulated in terms of the satellite structure and produce elements of the core term (e.g. "If we start with changing the first letter from 'k' to 'm', that means the edit script begins with `rep`(k,m)."). The direction of the mapping would be problem $\rightarrow$ solution.

Now, when using the ICORE framework, much is the same but there is a key difference. To compute the Levenshtein distance we are still interested in the optimal edit script to get from the first input string to the second, but the rewrite relation we provide goes *the other way*. It works on (consumes) the core terms and in the process generates the satellite terms. So here, we have a mapping that works in the *opposite* direction, namely solution $\rightarrow$ problem.

This flip of direction brings the advantage that designers of ICORES only need to state how *answers* (or solutions) relate to *questions* (problems) and the compiler would take over the role of then figuring out in an automated fashion the transformation from questions to answers (which is what the end-user of the program ultimately wants).
This avoids the pitfalls of low levels of abstraction and getting tangled up too quickly and to severely in implementation detail. Furthermore, as a rule, mapping from the concrete to the abstract is usually easier since all the information is already there.

Maybe some of it needs to be forgotten, but that is relatively simple and there is not ever a need to invent (or worse: compute) any of it.
All of this in turn allows for persons with far less domain knowledge in informatics and dynamic programming to become ICORE designers, saving human and machine resources and allowing a wider range of people to benefit from the gathered knowledge of exactly these domains.

Let us now explain what is meant by *coupled* rewriting. Put simply, coupled rewriting means that a single core term is simultaneously being rewritten into multiple (specifically: k) satellite terms, using up the same function symbols in the same locations. The tricky part is coming up with a way to tell what it means to be the "same" location over several rewrites.
This process is needed to make sure that any given found core term actually rewrites validly into all the specified inputs. If this was not semantically guaranteed, the whole framework would be pointless.

The precise definition of coupled rewriting is as follows:

Definition 2. *Coupled Rewrite Relation Let c be a core term of $T(\zeta, V)$, seen as a tree. We provide each operator symbol occurring in c with its initial position in c. A* coupled rewrite *of c is a sequence of k rewrite derivations $c \to^*_{R_1} t_1, \ldots, c \to^*_{R_k} t_k$ such that there exist some number j and intermediate reducts $(t_0^1, \ldots, t_j^1), \ldots, (t_0^k, \ldots, t_j^k)$ satisfying the following:*

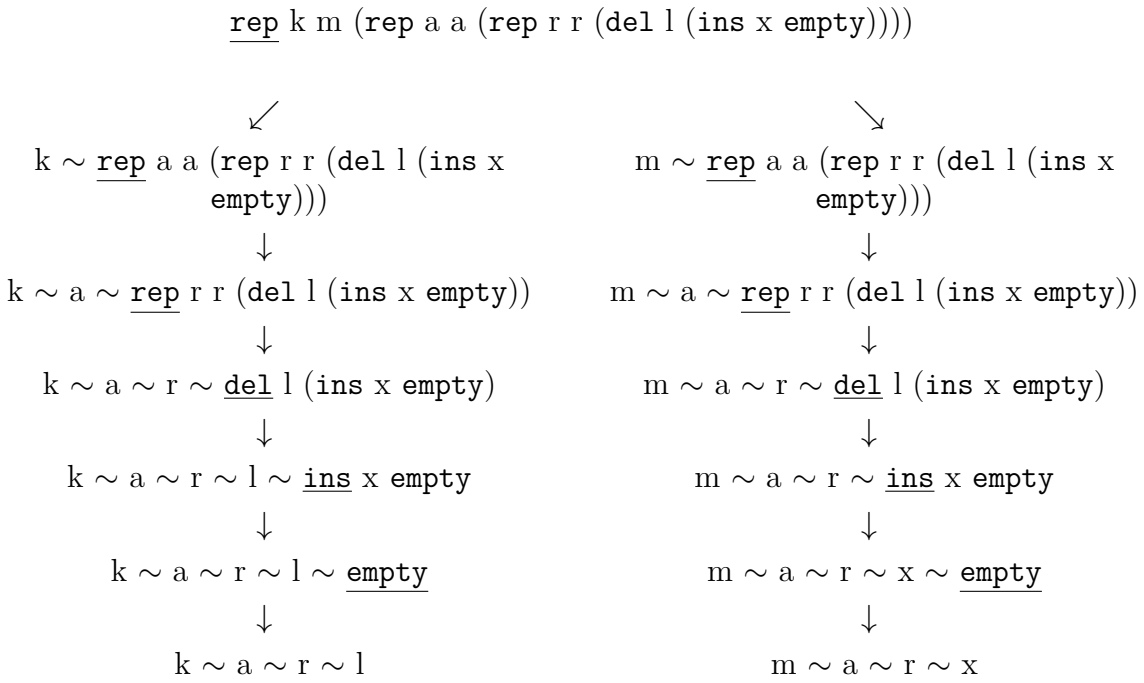
- $c = t_1^0 = \cdots = t_k^0$ *(start of a coupled derivation)*
- $t_1 = t_1^j, \ldots, t_k = t_k^j$ *(end of a coupled derivation)*
- *for each e, $0 \leq e < j$ (step in a coupled derivation), there exists a rule projection $l \Rightarrow r_1 \mid \ldots \mid r_k$ such that t_i^e rewrites modulo associativity to t_i^{e+1} with the rule $l \to r_i$ and all rewrites apply simultaneously at the same place.*

  In other words: for each i, there is a term v_i, a position p_i in v_i and a substitution σ_i such that $t_i^e =_A v_{i|p_i} = l\sigma_i, t_i^{e+1} = v_i[r\sigma_i]_{p_i}$ and all operator symbols in l have the same initial *position in c.*

Let us take a closer look at one example for clarification of the above. Consider a simple 2-dimensional ICORE solving the classical edit distance problem over the English alphabet and with the signature operators {`rep, del, ins, empty`}. Now, take a look at this core term:

$c =$ `rep 'k' 'm' (rep 'a' 'a' (rep 'r' 'r' (del 'l' (ins 'x' empty))))`

It allows for the following coupled derivation. Here, in each step the function symbol that is used up in the rewrite is underlined to make the process a little clearer.



Of course, there is more than just one such a core term that rewrites to the inputs `karl` and `marx`. The set of all of these terms can be considered the search space of an optimisation problem.

Definition 3. *ICORE Candidate Solution*
Given a k-tuple of satellite terms $\theta = (t_1, \ldots, t_k)$, the set of candidate solutions *for θ is the set of core terms c such that*

- *c is recognised by the tree grammar: $c \in L(\mathcal{G})$, and*
- *c has a coupled rewriting into all inputs: $c \Rightarrow^* t_1 \mid \ldots \mid t_k$.*

Now that we have constructed the search space of candidate solutions, we will name the process by which to identify *optimal* candidate solutions. What problem exactly an ICORE solves is ultimately up to the evaluation algebra and the objective function, because they map the search space of candidate solutions to optimal and non-optimal solutions.

Definition 4. *Solution of an ICORE*
Given an ICORE with the evaluation algebra A, its objective function Φ and a k-tuple $(t_1, \ldots, t_k)$ in $T(\Sigma_1) \times \cdots \times T(\Sigma_k)$, the multiset of optimal solutions *is given by*

$$\Phi([A(c) \mid c \in L(\mathcal{G}), c \Rightarrow^* t_1 \mid \ldots \mid t_k])$$

Another short example to make this clearer: when we again take the ICORE introduced in the example above, one possible evaluation algebra it might looks like this:

Algebra: LevenshteinScore

$\texttt{rep}(a, b, x)$	$=$ if $a == b$ then x else $x + 1$
$\texttt{del}(a, x)$	$= x + 1$
$\texttt{ins}(a, x)$	$= x + 1$
$\texttt{empty}$	$= 0$
Φ	$=$ minimum

Now, if we take the core term c from above, the application of the algebra comes out to the following:

$$\begin{aligned}
A(c) &= rep(k, m, (rep(a, a(rep(r, r(del(l, (ins(x, empty))))))))) \\
&= 1 + rep(a, a(rep(r, r(del(l, (ins(x, empty))))))) \\
&= 1 + 0 + rep(r, r(del(l, (ins(x, empty))))) \\
&= 1 + 0 + 0 + del(l, (ins(x, empty))) \\
&= 1 + 0 + 0 + 1 + ins(x, empty) \\
&= 1 + 0 + 0 + 1 + 1 \\
&= 3
\end{aligned}$$

After evaluating c under LevenshteinScore, it becomes clear that c happens to be non-optimal. The best possible score for this particular problem is 2 (four replacements, two of them without changing the letter). That means, since the objective function is minimising, that Φ would discard c in favour of a better-scoring core term.

The actual computational problem that is to be solved by ICORE-programs is quickly and efficiently spanning and evaluating the search space of core terms that rewrite to the satellite inputs. This is being done by splitting the problem into multiple sub-problems that can be evaluated separately and quickly via dynamic programming.
The task that falls on the compiler is generating an efficient implementation using the specification (with the inverse rewrite relation) only.

2.2 Trees, Forests and Subforests

The reader is assumed to have some basic familiarity with the idea of sequences, trees and forests as data structures. In this section, only the relevant additional information specific to this framework will be discussed.
For a comprehensive introduction to the topic, including all basic definitions, please refer to the cited work of Schirmer[9].

A *subtree* of a tree is one of its children with its respective offspring. It is said to be a *complete subtree* if all offspring of the original node is included. If some nodes somewhere down the tree are missing, it is instead called an arbitrary subtree.

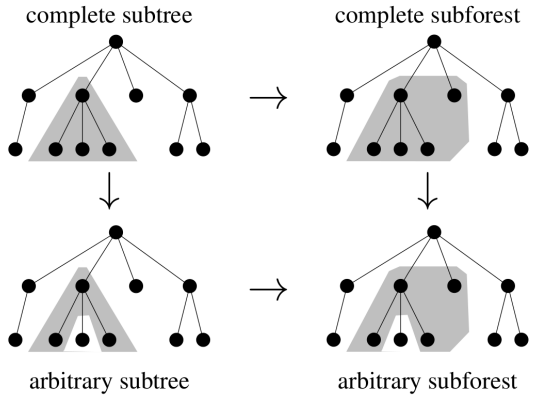


Figure 2.1: Complete and arbitrary subforests and subtrees.

Analogous to that, a *subforest* is a subsequence of the sequence of children of a particular node. It is called a *complete* oder *closed subforest* (CSF) if all offspring of all nodes in the subforest are included. Otherwise, it is referred to as an arbitrary subforest.
See figure 2.1 for a graphical illustrations of these concepts, as taken from [3].

Since the set of children of a node in a tree is again a forest of trees, it becomes clear that problems on trees and problems on forests imply each other. One cannot deal with one without *by definition* also having to deal with the other.

Schirmer also notes the following: In any forest, all CSFs can be represented by a pair of integers. The first integer is the node index, signifying the *start* of the closed subforest (we assume all nodes to be equipped with this index, usually depth-first). The second represents the *length* of the same (a length index of 0 is taken to mean an empty subforest). From this information, every CSF is uniquely identifiable.
Problems on sequences like strings can be thought of as a subset of problems on trees. Any sequence can be represented as trees in two different ways:

- a forest of leaf trees
  "asdf" = $\mathrm{a}(\varepsilon) \sim \mathrm{s}(\varepsilon) \sim \mathrm{d}(\varepsilon) \sim \mathrm{f}(\varepsilon)$
- a single unbranching tree with labelled nodes
  "asdf" = $\mathrm{a}(\mathrm{s}(\mathrm{d}(\mathrm{f}(\varepsilon))))$

These different ways of representing sequences as trees have computational implications that are quite important and which we will revisit in chapter 4.

2.2.1 Höchsmann-Mapping

Now, consider an ICORE like `TreeAlign`. It receives two input trees and to find the optimal alignment between them, all subforests of the first tree need to be compared to all subforests of the second tree. The results of all those comparisons constituting sub-problems of a dynamic programming problem and hence stored for re-use in a matrix (indexed by both pair representations of the two subforests that are to be aligned).

Höchsmann[5] was able to show that this four-dimensional matrix is sparse. This means in practice that a lot of time and space is being wasted, since there are a lot alignments being performed with empty and/or nonsensical trees. He proposes a

two-stage mapping for improved efficiency in both required memory and execution time.
The first stage is the mapping function α, which matches closed subforests to their representation as pairs of node index (i) and length (j).
As the second step, this pair is mapped to a single integer with the following mapping β, which we will refer to in the following sections as Höchsmann-mapping:

$$\beta_T(i,j) = \begin{cases} 0, & \text{if } j = 0 \\ \text{offset}_T[i] + j, & \text{otherwise} \end{cases}$$

"offset_T" in this case is a pre-computed table. The offset of a node in a tree is the number of CSFs that have a node index of less than i in their pair representation. The result is a vector perfectly representing the original tree but much more efficiently stored. Obviously, this too carries computational relevancy. And this, too, will be further discussed in chapters 3 and 4.

2.3 Example ICORES

After having explained the components and assembly of ICORES as a concept in the previous sections, here we want to concentrate on presenting two examples solving classical and well-known problems from the field of informatics to convey a more intuitive idea of how ICORES solve problems.

2.3.1 String Edit Distance with Affine Gaps

First, we are going to look at the ICORE for standard string editing, but expanded by an affine gap model, called `Affine`.
Affine gap scoring is a method motivated by gene comparison as insertions or deletions of any length can occur after an initial DNA breakage, which itself is quite rare. This is modelled through gap "openings" (for either deletions or insertions) having a relatively high cost while gap "extensions" are only charged a relatively small amount. Call the former o and the latter e and the cost of any gap of length n in this model comes out to $o + (n-1) \cdot e$

This expansion of the standard string edit model demands a change in the core signatures. Instead of just having `del` and `ins` operators, we also add operators named `open_del` and `open_ins`. These will represent gap openings while the simple operators will refer only to gap extension.
At this point there is no mechanism in place to ensure that the operators are being used in a correct fashion. To rule out the possibility of constructing core terms that are actually invalid, the tree grammar AFFI is used. AFFI restricts the candidate space in such a way that gap extension operators are only available after a gap opening operator has already been introduced, that no two identical gap opening operators are used in the same gap, etc.

This is a good example of how the tree grammars of ICORES can be used to restrict the candidate space to only *valid* solutions instead of for example penalising those invalid solutions during scoring as discussed earlier.

The evaluation algebra, AffineScore in this case, is the last piece in the puzzle. Here we assign a score to every core term considered. The precise numbers bear no deeper meaning but instead represent one of many possible scoring schemes. The most important notion here is to keep the promise made earlier about high gap opening costs and low gap extension costs (under the objective function of minimisation).

A representation of `Affine` in the notation proposed by Giegerich and Touzet can be found on page 15.

2.3.2 Classical edit distance on trees

This next ICORE describes a similar alignment problem, but with the underlying data structure being trees instead of strings. Trees are being modelled through the satellite signature TREE, the three function symbols being ε for the empty tree, f for a node and $\sim$ as tree (sibling) concatenation.
What may appear a little counter-intuitive is that in this notation, not all children of a node are directly connected to the parent. Instead, a parent is connected to a forest of siblings that are all to be treated as its children.

Like before, we have the core function symbols `rep`, `del`, `ins`, $\sim$ and `empty`. In this example, however, there is no affine aspect.
Furthermore, the evaluation algebra uses additive similarity scoring and uses a function w for weighted replacement and gap penalties γ and δ for insertions and deletions. Hence, Φ is maximisation.

A representation of `TreeAlign` in pseudo-notation can be found on page 16.

ICORES `Affine`, dimension 2

Satellite Signature: $\mathsf{SEQ} = \mathcal{A} \cup \{\sim, \varepsilon\}$ with

$$\begin{array}{lll} a & : & \to \mathcal{A}^* \quad \text{- for } a \in \mathcal{A} \text{ single letter sequence} \\ \varepsilon & : & \to \mathcal{A}^* \quad \text{- the empty sequence} \\ \sim & : \mathcal{A}^* \times \mathcal{A}^* & \to \mathcal{A}^* \quad \text{- sequence concatenation} \end{array}$$

Satellite Signature: SEQ - the second input uses the same satellite signature

Core signature: $Ali = \mathcal{A} \cup \{\texttt{rep}, \texttt{open\_del}, \texttt{del}, \texttt{open\_ins}, \texttt{ins}, \texttt{empty}\}$ with

$$\begin{array}{lll} \texttt{rep} & : \mathcal{A} \times \mathcal{A} \times Ali \to Ali & \text{- replacement} \\ \texttt{open\_del} & : \mathcal{A} \times Ali \to Ali & \text{- deletion gap opening} \\ \texttt{del} & : \mathcal{A} \times Ali \to Ali & \text{- deletion} \\ \texttt{open\_ins} & : \mathcal{A} \times Ali \to Ali & \text{- insertion gap opening} \\ \texttt{ins} & : \mathcal{A} \times Ali \to Ali & \text{- insertion} \\ \texttt{empty} & : \to Ali & \text{- empty alignment} \end{array}$$

Grammar: AFFI

$$\begin{array}{lll} X \to \texttt{rep}(a,b,X) & \mid \texttt{open\_del}(a,Y) & \mid \texttt{open\_ins}(a,Z) \mid \texttt{empty} \\ Y \to \texttt{del}(a,Y) & \mid \texttt{rep}(a,b,X) & \mid \texttt{open\_ins}(a,Z) \mid \texttt{empty} \\ Z \to \texttt{ins}(a,Z) & \mid \texttt{rep}(a,b,X) & \mid \texttt{open\_del}(a,Y) \mid \texttt{empty} \end{array}$$

Rewrite Rules:

$$\begin{array}{lll} \texttt{rep}(a,b,X) & \Rightarrow^* a \sim X & \mid b \sim X \\ \texttt{open\_del}(a,X) & \Rightarrow^* a \sim X & \mid X \\ \texttt{del}(a,X) & \Rightarrow^* a \sim X & \mid X \\ \texttt{open\_ins}(a,X) & \Rightarrow^* X & \mid a \sim X \\ \texttt{ins}(a,X) & \Rightarrow^* X & \mid a \sim X \\ \texttt{empty} & \Rightarrow^* \varepsilon & \mid \varepsilon \end{array}$$

Algebra: $\mathsf{AffineScore}$ $Ali = \mathcal{N}$

$$\begin{array}{ll} \texttt{rep}(a,b,x) & = \text{if } a == b \text{ then } x - 5 \text{ else } x + 2 \\ \texttt{open\_del}(a,x) & = x + 10 \\ \texttt{del}(a,x) & = x + 1 \\ \texttt{open\_ins}(a,x) & = x + 10 \\ \texttt{ins}(a,x) & = x + 1 \\ \texttt{empty} & = 0 \\ \Phi & = \text{minimum} \end{array}$$

ICORES `TreeAlign`, dimension 2

Satellite Signature: TREE with

ε	:		$\to Tree$	- the empty forest
f	:	$Tree$	$\to Tree$	- node labels, for $f \in F$
$\sim$	:	$Tree \times Tree$	$\to Tree$	- tree concatenation

Satellite Signature: TREE - the second input uses the same satellite signature

Core signature: TreeAl with

`rep`	:	$F \times F$	$\times Ali \to Ali$		- node label replacement
`del`	:	F	$\times Ali \to Ali$		- node deletion
`ins`	:	F	$\times Ali \to Ali$		- node insertion
`empty`	:			$\to Ali$	- empty tree alignment
$\sim$	:	$Ali \times Ali$	$\to Ali$		- subforest concatenation

Grammar: T(TreeAl)
Rewrite Rules:

$X \sim Y$	$\Rightarrow^*$	$X \sim Y$	$\mid X \sim Y$
`rep`(f, g, X)	$\Rightarrow^*$	$f(X)$	$\mid g(X)$
`del`(f, X)	$\Rightarrow^*$	$f(X)$	$\mid X$
`ins`(f, X)	$\Rightarrow^*$	X	$\mid f(X)$
`empty`	$\Rightarrow^*$	ε	$\mid \varepsilon$

Algebra: TreeSimilarity

`rep`(a, b, x)	$= x + w(f, g)$	- weighted label replacement score
`del`(a, x)	$= x - \delta$	- node deletion penalty δ
`ins`(a, x)	$= x - \gamma$	- node insertion penalty γ
`empty`	$= 0$	- empty tree similarity score
$x \sim y$	$= x + y$	- similarity score adds up
Φ	= maxmimum	- similarity scoring maximises

3 Practical Implementations

In this section we will take a look at the `Haskell` code for ICORES. This is the main novelty and primary effort in this work for there previously have been no published implementations in `Haskell` or any other language.
Although both ICORES presented in section 2.3 have been implemented, we will stick mainly to the source code of `TreeAlign`, since that one is more complex and can in all aspects but one (which we will discuss separately) be seen as a superset of `Affine`.

3.1 Input Preprocessing

In the scope of ICORES with tree-like satellite signatures, like the satellite signature TREE in the ICORE `TreeAlign` for example, it becomes necessary to do some non-trivial preprocessing. It should be noted, however, that these steps are necessary for *all* ICORE-implementations with tree-like satellite structures, not just this specific example.

3.1.1 Right-Balancing of Input Trees

The `Haskell` data structure that represents the satellite structure looks like this:

```haskell
data Sat = E                  -- empty tree
         | N Char Sat         -- node labelled by a charcter
         | (:~:) Sat Sat      -- tree concatenation
         deriving (Eq, Show) -- necessary type class derivations
```

This however, by itself, would lead to problems further down the line. During traversal for example (see section 3.1.2), there is a step that requires to find all CSFs of a given node. This task would constitute considerable computational and developmental effort if the input was left in its original format, because this format allows for a subforest continuing in entirely different branches of the data structure.

As an example, consider figure 3.1 on page 19. This is one possible input tree with seven labelled nodes. However, to single out the subforest $(3, 4)$, for example, would need a lot of effort keeping track of current and already visited nodes in the tree, be unlikely to have a simple and / or elegant implementation and possibly require mutable state.

It is because of this that before traversing and indexing the tree for $\mathcal{O}(1)$ access, all input trees are rewritten into a more readily processable format by this function:

```
-- The operator '~~' is a so-called "smart constructor" which
-- realises the satellite constructor ':~:' modulo associativity
clean :: Sat -> Sat
clean E         = E             -- identity
clean (N c E)   = N c E         -- identity
clean (N c s)   = N c (clean s) -- propagate
clean (p :~: q) = case (clean p) of
  E         -> clean q
  (N c E)   -> (N c E) ~~ (clean q)
  (N c s)   -> (N c (clean s)) ~~ (clean q)
  (s :~: t) -> (clean s) ~~ (clean (((clean t) ~~ (clean q))))
```

The result is a tree more unbalanced to the right and easier to traverse. Now, a function that returns a CSF of a given length starting from a certain node is both simple and elegant to implement. It is, however, not entirely unbalanced, which would make the tree a simple linked list, since structure-changing rewrites only happen on the concatenation function symbol (:∼:) as becomes obvious quickly looking at the implementation of `clean`. This means that parent-child relationships are always preserved in the rewrite.

The result of `clean` being applied to the tree in Figure 3.1 can be seen in Figure 3.2 on page 21.

3.1.2 Tree Traversal

In order to gain an efficient implementation, $\mathcal{O}(1)$ access to all closed subforests of the input trees is necessary. In section 2.2.1, we already discussed how to represent CSFs in a space-efficient manner. But to ultimately generate an array with the fitting indexes and elements, first of all there is a need to traverse all input trees to identify all CSFs and find their pair representation. `traverse` is a function implemented to do just that.

In the code shown below `AccessRule` refers to a pair (a 2-tuple) of both another pair of `Int`s and a `Sat`, referencing a pair representation of a subforest and the subforest encoded in the data structure respectively. `Ndown`, `Nright` and `Maxlen` all refer to pairs of `Int`s. `Ndown` and `Nright` each pair a node index with the index of their respective right-hand sibling or child. This way, we can later reconstruct which node has which "neighbours" without having to refer back to the original data structure.
The second value of a `Maxlen` indicates the maximum subforest length of the the node with the index given as the first value. Knowing this is necessary as to not build invalid (too large) subforests.

The empty tree serves as a base case for the recursion but does not generate any rules except for the trivial one: There is a subforest of length 0 at that node. The same rule is generated on a node also, plus the rule that there is also a subforest of length 1 there. Furthermore, the child of this node is registered under `Ndowns`. A concatenation symbol with either - an empty tree or another concatenation symbol - on its left branch never occurs after the tree has been cleaned. That leaves the only "interesting" case of `traverse` to be a node on the left branch. Here, both the node (`p`) and the right branch (`q`) are recursed on, before all *their* rules are combined with all new CSFs (the return value of `getAllCSFs`) and child/sibling relations that can be created with the current node at the root: the node in `p` and its child, the same node and the next one in `q` as its sibling (which trivially must have $i+1$ as node index, when i is the greatest node index in the subtree `p`). Furthermore, the maximum length of the subforest is calculated and appended to `[Maxlen]`.

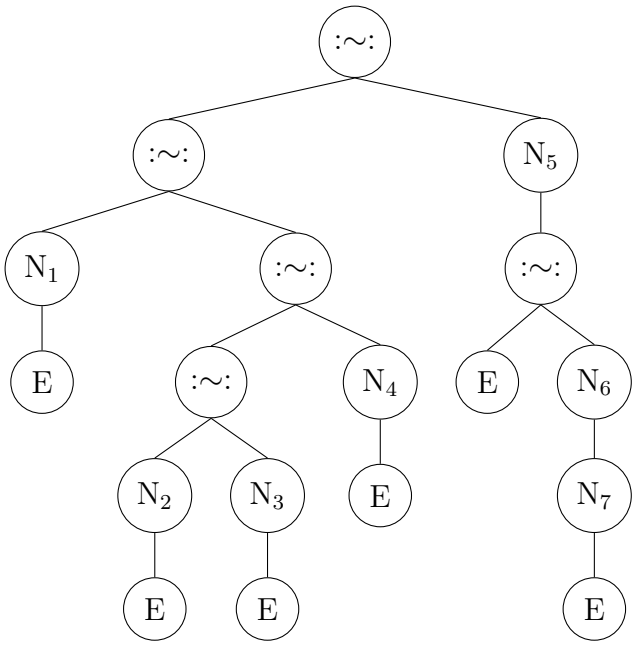


Figure 3.1: One possible input tree

```
traverse :: Sat -> (([AccessRule], Int), ([Ndown],[Nright]), [Maxlen])
traverse = (traverse' 1) . clean where -- pointfree
  traverse' :: Int -> Sat -> (([AccessRule], Int),([Ndown],[Nright]),[Maxlen])
  traverse' n E           = ((([(n,0), E)],n+1),([],[]), [(n,0)])
  traverse' n t@(N c s)   =
      let ((rules, count), (downNs, rightNs), lens) = traverse' (n+1) s
      in  ((((n,1), t) : ((n,0), E) : rules, count),
            ((n, n+1) : downNs, rightNs), (n,1) : lens)
  traverse' n t@(p :~: q) = case p of
      E         -> error "unclean satellite structure (empty)"
      (l :~: r) -> error "unclean satellite structure (concat)"
      (N c s)   -> let ((p_rules, p_count), (p_downNs, p_rightNs), plens)
                                      = traverse' (n+1) s

                       ((q_rules, q_count), (q_downNs, q_rightNs), qlens)
                                      = traverse' (p_count) q

                       (csfs, maxlen) = (getAllCSFs n t)

    in ((csfs ++ p_rules ++ q_rules, q_count), ((n, n+1):p_downNs ++ q_downNs,
        (n, (p_count)) : p_rightNs ++ q_rightNs), (n, maxlen) : plens ++ qlens)
```

This traversal function turned out the most challenging part of the implementation process and we think that it would be non-trivial to abstract from the specific data structure to any form of tree or sequential input data type.
A potential ICORE-Compiler, however, would have to do just that and generate a traversal function not unlike this one from the declaration of the satellite signature alone.

3.2 Neighbourhood Lookup

Not shown here are the definitions / implementations for `lx`, `dx` and `rx` (and `ly`, `dy` and `ry` respectively). These functions, dubbed "neighbourhood lookup functions", all have the same type signature (`dx, dy, rx, ry, lx, ly :: Int -> Int`) and are similar in implementation and functionality as well.
Neighbourhood lookup happens further into the program but requires some setting up. The `AccessRules` and neighbourhood relations gathered by `traverse` are used to construct arrays that allow to reproduce whether or not a given node had any siblings or children and if so, what their respective node indexes were. These arrays are then used in the lookup functions mentioned above, `dx` and `dy` returning children if there are any and the shorthand for the empty tree if not. Same for `lx` and `ly` and left siblings, respectively `rx` and `ry` and right siblings. The `x`s and `y`s in these function names reference on which of the two input trees they work.

3.3 Dynamic Programming Tree Alignment

In this section, we take a closer look on how the dynamic programming components of ICORES are realised and how the language features of `Haskell` can help make the job a little easier.

As mentioned earlier, arrays provide $\mathcal{O}(1)$ access to all elements contained in them, while `Haskell` lists (which are simply linked lists) would only reach performance in $\mathcal{O}(n)$ on average. Arrays in this language are constructed with the function `array` which expects two values of a type in the typeclass `Ix` as parameters - these will serve as the lower and upper bound respectively. `Ix` contains types that can be used to express a *range* of values for indexing. Typical instances are `Int` and `Char`. Furthermore, `array` takes in a list of access rules (each a pair of one index value and the corresponding subforest). The operator `(!)` is the infix array access operator for arrays.
The function `bounds` takes in an array and gives back a pair of both the lower and the upper bound of that array (meaning the indices to access the first and the last element). `indices` works in a similar manner but returns a list of *all* indices instead.

During preprocessing of the inputs, arrays are created with the help of `traverse` that represent the input tree as one-dimensional arrays, mapping the Höchsmann-indexes (see section 2.2.1) to CSFs. In the example below, these are named `xa` and `ya`. Using these, a two-dimensional table is created, exactly $length(\texttt{xa}) \times length(\texttt{ya})$

in size. In it, at position (i, j) are stored all core terms generated by aligning the CSFs at positions i and j in `xa` and `ya`.

```
-- Main array of core terms
mV = let ((lx, hx), (ly, hy)) = (bounds xa, bounds ya)
in  array ((lx,ly),(hx,hy)) [((x,y), pV x y) | x <- indices xa,y <- indices ya]
```

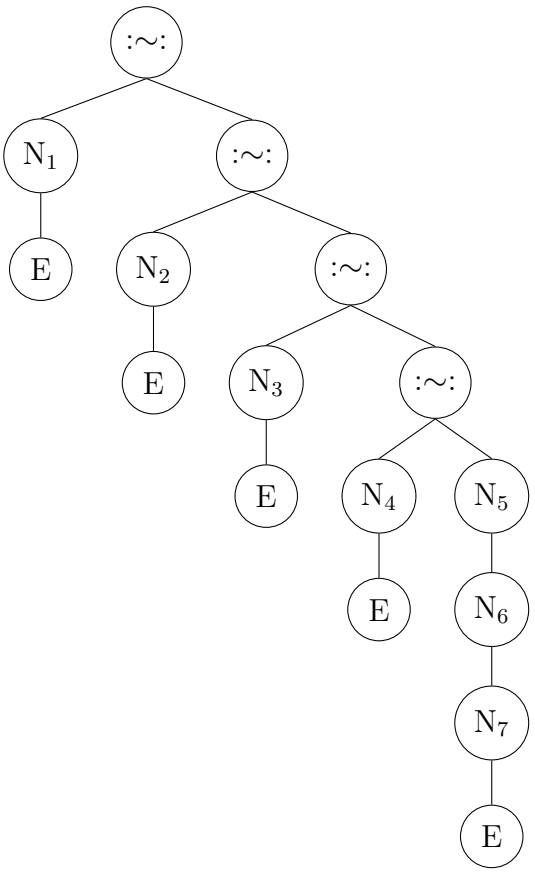


Figure 3.2: The output of `clean` given the tree in figure 3.1.

`pV` is the function that maps each cell in the table to the list of core terms that need to be generated. It takes as parameters two arguments of type `Int`, indexes in `xa` and `ya`.

`empty` does not carry special meaning but instead just references 0, or in other words, the index for the empty subforest. This serves no computational purpose but rather makes the code a little easier to comprehend for a human reader.

What makes this dynamic programming is the following: at each position in the table, only one layer of function symbols is generated and references the smaller sub-problems one layer deeper at their respective place in the table.
Hence, problems that occur multiple times are only calculated once. Afterwards, the result is stored in the table and can be accessed quickly however often is needed without the need for further computation. This is possible and correct since the problem adheres to Richard Bellman's principle of optimality[1], the fundamental definition of dynamic programming:

> "An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision."

Which symbols may be generated becomes clear from the type of trees that are to be aligned. For example, if we consider `TreeAlign`, and find that `pV` is called with

indexes that both map to empty trees, only the `empty` constructor makes sense to place.
Similarly, if the indexes map to a node and an empty tree, only deleting the node is sensible. In this case `pV` returns a list of core terms: each deleting the current node, but for every possible core term representing the child of the node. This information is retrieved from a different cell in the same table using the lookup function `dx` (see previous section).
A more complex example would be the alignment of two concatenated trees and a labelled node. Here, `pV` generates two lists of core terms: where the node is concatenated with the left element of the original concatenation and the right element is concatenated with the empty tree and vice versa.
The lists are then also concatenated and `phi`, the objective function of the algebra, is applied to the result. This last step is only necessary since out of the two lists and their disjunct subproblems may arise different scores. This possibility is not given when there is no or only one other cell in the table being referenced like in the examples before.

For all possible combinations of alignments, please refer to the code section below.

```
pV l r = case xa ! l of
 E       -> case ya ! r of
   E       -> [empty]
   (N d _) -> [oins d x | x <- mV ! (empty,(dy r))]
   (_:~:_) -> [cat x y  | x <- mV ! (empty,(ly r)),y <- mV ! (empty,(ry r))]
 (N c _) -> case ya ! r of
   E       -> [odel c x | x <- mV ! ((dx l),empty)]
   (N d _) -> [rep c d x| x <- mV ! ((dx l),(dy r))]++
              [odel c x | x <- mV ! ((dx l),r)]     ++
              [oins d x | x <- mV ! (l,(dy r))]     ...phi
   (_:~ _) -> [cat x y  | x <- mV ! (l,(ly r)),y <- mV ! (empty,(ry r))]++
              [cat x y  | x <- mV ! (empty,(ly r)),y <- mV ! (l,(ry r))]...phi
 (_:~:_) -> case ya ! r of
   E       -> [cat x y | x <- mV ! ((lx l),empty),y <- mV ! ((rx l),empty)]
   (N d _) -> [cat x y | x <- mV ! ((lx l),r),y <- mV ! ((rx l),empty)]++
              [cat x y | x <- mV ! ((lx l),empty),y <- mV ! ((rx l),r)]...phi
   (_:~:_) -> [cat x y | x <- mV ! ((lx l),(ly r)),y <- mV ! ((rx l),(ry r))]++
              [cat x y | x <- mV ! (l,(ly r)),y <- mV ! (empty, ry r))]++
              [cat x y | x <- mV ! (empty,(ly r)),y <- mV ! (l,(ry r))]++
              [cat x y | x <- mV ! ((lx l),r),y <- mV ! ((rx l),empty)]++
              [cat x y | x <- mV ! ((lx l),empty),y <- mV ! ((rx l),r)]...phi
```

The number of non-terminal symbols in the tree grammar of an ICORE corresponds to the number of tables like these needed in the implementation, since one cannot know beforehand in which "state" the optimal core term will be at any given position and hence has to check all possibilities. A change of non-terminal in the rewrite relation corresponds to a cross-table reference in the implementation.
The tree grammar for `TreeAlign` does not specify extra non-terminal symbols, so we only need one table. `Affine` however, requires three, since its tree grammar

incorporates three non-terminal symbols. This also means that there are three arrays to be constructed (each referencing cells of itself and the others) and three access functions like the one above to be written.

3.4 Haskell Language Features

Garbage collection means that the programmer does not have to worry about the end of a value's lifetime. Lazy evaluation means that she doesn't have to worry about the beginning of that lifetime, either.
– Doaitse Swierstra

While ICORES could be implemented in a wide range programming languages, we decided to use `Haskell` for this work. Haskell is a purely functional programming language with strong static types with Hindley-Milner-style [4] [7] type inference. It is also under active development and has multiple language features that are useful for implementing ICORES:

Since `Haskell` has *lazy evaluation*, we do not need to keep track of the order, in which the cells in our tables are evaluated. A value will be computed and stored when it's needed somewhere else in the program and not before then. So the task of making sure all computations are executed in the correct order is not something the programmer needs to invest energy into but is instead being taken over by the `Haskell` runtime system.
Another benefit of lazy evaluation here is that we do not have to worry about wasted resources for cells that we do not actually need computed. The second upside to "a value is only ever calculated when it is needed" is that a value is *not* being calculated when it turns out that we do not need a specific value.

One of the advantages of the ICORES framework is its high degree of modularity. Assuming compatibility, all components of an ICORE can be exchanged for others or be used in different ICORES. `Haskell`s module system allows us to capture this effect on the implementation layer, also. Independent parts of ICORES (like evaluation algebras and core terms) can be implemented in separate modules and imported if needed. This also makes code reuse feasible, clean and quick.

`Haskell` also has first-class and higher-order functions, which makes it easy to construct, combine and pass around functions (to use as Φ, for example) or even choose from many already implemented examples.

4 Results

After testing our programs for correctness against a reference implementation in a number of example cases, there have been three quantitative tests done on the final implementations of the two ICORES presented earlier. In this chapter, we discuss the results of those tests as well as the underlying theory.

4.1 Asymptotics & Theory

Tree alignment has been a topic of interest for decades. Its many applications in informatics, computational biology and a range of other subjects have lead to a profound body of research (see, for example, [9]).

Consider an alignment problem in tree-like structures as in the ICORE `TreeAlign`. A naïve observer would most likely assume that, because of the $\mathcal{O}(n)$ subforests that need to be calculated and compared with the $\mathcal{O}(m)$ subforests in the other tree (with n and m being the respective sizes of the input trees), pairwise tree alignment had an average complexity of at least $\mathcal{O}(n^2 \cdot m^2)$.

However, Denise, Herrbach and Dulucqc have found that this is not always the case[2]. They show that, even if not in all cases, in the *average case* over all possible tree forms, tree alignment can be reduced to a complexity of $\mathcal{O}(n \cdot m)$. The *worst case* is still in $\mathcal{O}(n^2 \cdot m^2)$
We shall see in the following section that these theoretical results correspond nicely to the empirical data obtained in the tests.

4.2 Runtime behaviour of TreeAlign

The first test that was executed was a simple runtime test of the final implementation of `TreeAlign`. The two input trees each varied from the empty tree up to trees with 1000 nodes and the execution time was logged. Every input tree was randomly generated and not based in any way upon previous trees (to ensure an accurate picture of the *average* case in the data). The resulting data can be found in figure 4.1.

One quickly notices a trend. Generally - even though considerable preprocessing of the input trees was necessary - with double the input only comes roughly double the execution time. We observe, that this corresponds to the theoretical prediction of the average case complexity being within $\mathcal{O}(n \cdot m)$.

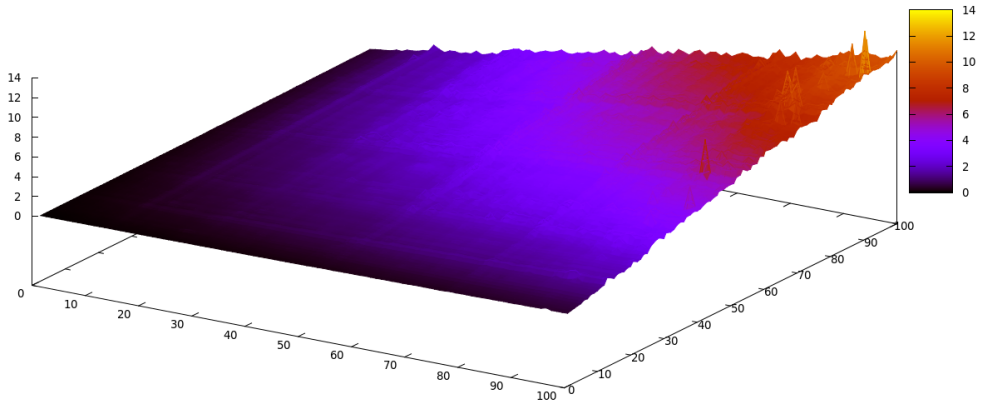


Figure 4.1: X and Z represent the size of the input trees (div 10), Y signifies the measured runtime (in seconds).

This lends confidence not only to the theoretical findings discussed earlier but also towards this specific implementation having no obvious flaws resulting in a fundamentally worse complexity. The implication for the compiler is that the implementation at hand can be used as a guideline for the code that is to be generated, which should either be at least comparable or better yet, significantly faster than this reference point.

4.3 Time-efficiency gained by Höchsmann-mapping

The next test that was done was with the exact same input trees as in the test above (in this run only varying from the empty tree to 500 nodes), this time computed twice. Once by an implementation that maps a closed subforest only to its pair representation and stores the result in a four-dimensional (sparse) table and once by the final implementation including the second stage of the Höchsmann-mapping, further reducing the pair-representation to just a single offset in an one-dimensional vector. The results are graphed in figure 4.2.

It becomes clear that there is not only a lot of used memory space saved by not using the sparse matrix approach. On top of that, there is also a great increase in efficiency concerning runtime. The speed-ups gained by the second mapping range from barely noticeable for the smallest trees up to highly impressive for some. On average, the slow programs roughly needed 4000% of the runtime of the fast program to terminate with some constellations working out to the fast program being over a hundred times faster than the slow one.

The implications are obvious. For a compiler to produce competitive programs, Höchsmann-mapping must be incorporated wherever possible. This leaves open the question if this can always be achieved. When the input types are arbitrary, it

is unclear whether a mapping into a non-sparse data structure will be known or deducible. One sensible way of addressing this problem would be for the compiler to treat the sparse data structure as a kind of "fallback". To be used if and only if there is no known method of packing the underlying data structure more densely.
This would probably also simplify implementation of the compiler since one could first concentrate all implementation effort on the general case and add more and more optimisations later on.

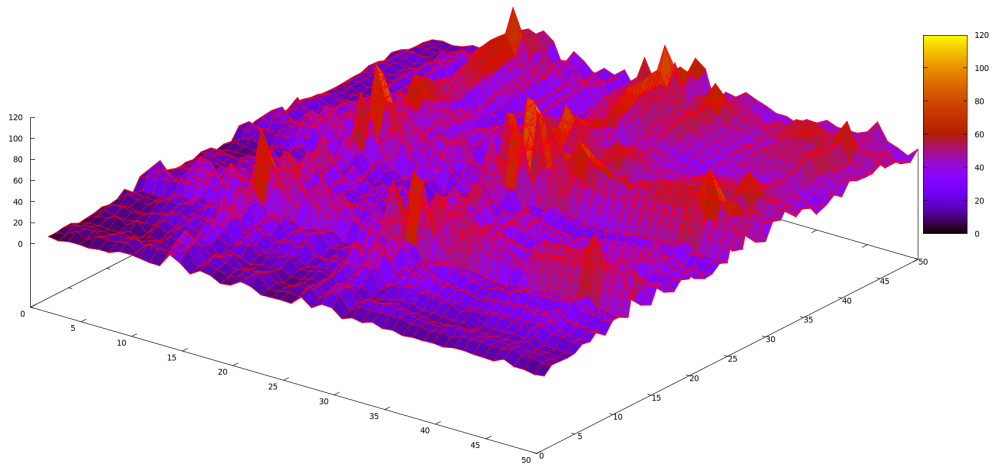


Figure 4.2: X/Z same as above. Y here represents the ratio of the runtimes of versions of TreeAlign without and with Höchsmann-mapping (in seconds).

4.4 Comparison of Strings and Sequence Trees

In section 2.2 we discussed that problems on sequences are a subclass of problems on trees since all sequences can be represented as either a single tree whose nodes only ever have one child or as a forest of leaf-trees.

One would expect that both implementations of sequences-as-trees alignment problems would be slower than the specialised implementations on strings. In the latter case, there is far fewer preprocessing needed as the problem of identifying remaining closed subforests evaporates into triviality (the remaining CSFs from a given position are all prefixes of the remaining string).
Comparing the two possibilities of representing sequences amongst each other, especially given the implementation of `traverse` from section 3.1.2, one would anticipate that vertical sequence trees can keep up with the linear complexity of sequence alignment. The traversal function only needs to make a single pass over the input tree and only needs to add obvious and trivial subforests without the need for extra computation. Additionally, one would expect that the forests of leaves needs vastly more time for processing because for each leaf in the forest all remaining subforests must be determined, raising the complexity to $\mathcal{O}(n^2)$.

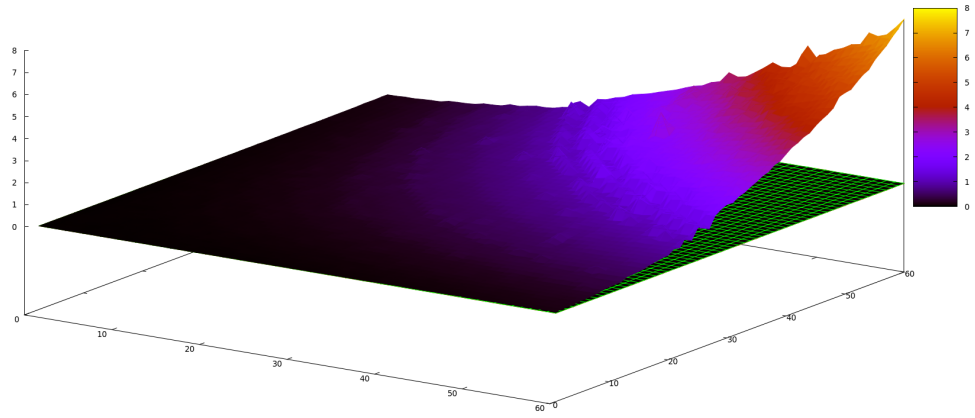


Figure 4.3: X/Z same as above. On the Y-axis, runtimes (again, in seconds) of different ICORES are plotted: Vertical sequence trees (green), simple alignment on strings (red, indistinguishable from green at this scale), and forests of leaves (purple-orange colour gradient).

The results, graphed in figure 4.3, give empirical verification to theoretical prediction. The vertical sequence trees are roughly as fast as the direct sequence alignment. As expected, there is a small overhead in preprocessing, leading to slightly slower results. This difference in speed is dwarfed, however, by the difference between vertical sequence trees and forests of leafs. The runtimes of the latter quickly diverge from those of the former, clearly showing worse-than-linear-growth.

This makes clear that an eventual compiler for ICORES should incorporate these results. It should be possible to recognise forests of leaves (and, for that matter, vertical sequence trees, too) as the simple sequences they are and automatically optimise. Like above, the tree alignment could constitute a default case to fall back on when multiple optimisations (like treating the tree as a sequence) have been proven to be non-feasible.

5 Conclusions & Outlook

In this section we highlight some of the limitations of the presented work, call for further research and implementation effort and share our conclusions on the path towards a compiler for ICORES.

5.1 Limitations

The programs created in the course of this thesis can be seen as but a first small step on the road leading to an eventual ICORES-compiler. There are many possible improvements in terms of speed, scalability and efficiency.

Unmentioned here, for example, is the topic of distributed dynamic programming, concurrency and parallelism. All programs discussed run one CPU of one machine and one CPU of one machine only. Competitive implementations of ICORES would need to be able, of course, to tap into multi-core if not multi-machine computing.

As far as empirical data-gathering is concerned, the tests performed in the context of this thesis only ever consider runtime. A more thorough investigation should also shed light on memory usage of the implementations. The runtime system of `Haskell` is well equipped for these purposes and it is quite possible that in the wake of this, more optimisation will be possible.

A further limitation of the work presented in this thesis is that all discussed programs are based on two-dimensional ICORES that solve very similar problems. There is not yet much diversity in the implemented examples from which experience can be drawn. More effort in implementing different ICORES for a wider range of problems would definitely be well placed.
For this to be possible, further research effort on the field of ICORES is also needed to find out just what is possible with ICORES and what is not and in what fields ICORES may eventually be applied that have not already been considered.

5.2 Towards a compiler for ICORES

We have seen, over the course of this work, that the framework of ICORES can be used as a basis for fairly efficient single-core programs. We highlighted the advantages of stating problems in a more declarative way and mentioned the higher level of abstraction that can be reached and the additional resources that can be freed when using ICORES. We also discussed some of the advantages that can be drawn from `Haskell`s module system, language features and type system.

On the other hand, we were also able to shed some light on possible difficulties and challenges that are likely to appear on the roadway towards an compiler for ICORES so that future adventurers on this road might be better prepared for it and equipped with the proper tools.

One promising and interesting programming technique that a potential compiler could profit from and that deserves a mention in this context are *recursion schemes*. In recent months, there has been quite a bit of activity on this topic. Especially the scheme of "Dynamorphisms" [8] deserves a second look as it promises to capture the notion of computation in ICORES quite elegantly.

All in all we are optimistic that the young research field of ICORES holds a lot of promise and look forward to further research effort in the future.

Bibliography

[1] Richard Ernest Bellmann. *Dynamic Programming*. Dover, Princeton, NJ, republished edition, 1957.

[2] Alain Denise Claire Herrbach and Serge Dulucqc. Average complexity of the jiang - wang - zhang pairwise tree alignment algorithm and of a rna secondary structure alignment algorithm. *Theoretical Computer Science*, 411(1), 2010.

[3] Robert Giegerich and Hélène Touzet. Modeling dynamic programming problems over sequences and trees with inverse coupled rewrite systems. *Algorithms*, 7(1), 2014.

[4] J. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146(1), 1969.

[5] Matthias Höchsmann. *The Tree Alignment Model: Algorithms, Implementations and Applications for the Analysis of RNA Secondary Structures*. PhD thesis, Bielefeld University, March 2005.

[6] Vladimir Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8), 1966.

[7] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science (JCSS)*, 17(1), 1978.

[8] Nicolas Wu Ralf Hinze. Histo- and dynamorphisms revisited. *WGP '13 Proceedings of the 9th ACM SIGPLAN workshop on Generic programming*, 1(1), 2013.

[9] Stefanie Schirmer. *Comparing Forests*. PhD thesis, Bielefeld University, August 2011.

Erklärung

Ich versichere, dass ich die vorliegende wissenschaftliche Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe. Die Stellen der Arbeit, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen sind, wurden unter Angabe der Quelle als Entlehnung deutlich gemacht. Das gleiche gilt auch für beigegebene Skizzen und Darstellungen. Diese Arbeit hat in gleicher oder ähnlicher Form meines Wissens nach noch keiner Prüfungsbehörde vorgelegen.

Jonas Betzendahl, August 28, 2014