Eindhoven University of Technology

MASTER

A taxonomy of Lempel-Ziv compression algorithms

Kouwenberg, J.J.C.

Award date:
2003

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Technische Universiteit Eindhoven
Department of Mathematics and Computer Science

A Taxonomy of Lempel-Ziv Compression Algorithms

by
ing. J.J.C. Kouwenberg

Supervisors:
dr.ir. C. Hemerik
drs. ing. H.P.J. van Geldrop-van Eijk

Eindhoven, August 2003

Preface

This thesis has been written to obtain the Master's degree at the Eindhoven University of Technology (TU/e) in The Netherlands. It completes the graduation phase of the condensed version[1] of the Computer Science Education. The normal version has a duration of five years. The duration of the condensed version is about two and a half to three years. To be admitted to the condensed version, a Bachelors degree in Computer Science –or equivalent– is required. In my case this is a Bachelors degree in Electrical Engineering, gained at the University of Professional Education in 's-Hertogenbosch.

This thesis is the result of the final graduation project, which has the duration of one academic year. The final graduation project is a continuation of a project initiated by members of the Software Construction group (SoC). SoC is a part of the Computer Science group which is a part of the department of Mathematics and Computer Science at the TU/e. The project belongs to the primary research subproject within SoC, namely TAxonomy-BAsed Software COnstruction (TABASCO). The aim of TABASCO is to create order in a specific problem area. In this case it is the area of Lempel-Ziv compression algorithms.

There are two people who I want to thank especially, namely my supervisors Kees Hemerik and Rik van Geldrop-van Eijk. They gave many helpful comments. One of the things they did was acting as "guard-dogs" for the clarity and simplicity of the text. Take for example a large formal proof. With some other definitions the proof reduced considerable. They also suggested many alternative – simpler – explanations of things.

Furthermore, I want to thank the people who gave their comments on the drafts of this thesis.

As last I want to thank all the other people who gave any kind of support during this final graduation project.

[1] "verkorte opleiding" or "VKO" in Dutch

Contents

Preface i

Summary vii

1 Introduction 1
- 1.1 Compression algorithms in general . . . 1
- 1.2 Lempel-Ziv compression algorithms . . . 2
  - 1.2.1 Description . . . 2
  - 1.2.2 Advantages . . . 4
  - 1.2.3 Applications . . . 4
- 1.3 Problem statement . . . 4
- 1.4 Document structure . . . 6

2 Preliminaries 7
- 2.1 Elementary concepts and notations . . . 7
- 2.2 Additional concepts and notations . . . 12

3 Constructing and presenting taxonomies 15
- 3.1 Constructing taxonomies . . . 15
- 3.2 Presenting taxonomies . . . 16
  - 3.2.1 Presentation in general . . . 16
  - 3.2.2 Presenting a visual summary of a taxonomy . . . 16
  - 3.2.3 Presenting an algorithm from the taxonomy . . . 16

4 Taxonomy 23
- 4.1 Overview . . . 23
- 4.2 Root specification . . . 23
  - 4.2.1 Initial root specification . . . 25
    - 4.2.1.1 Introducing lossless . . . 25
    - 4.2.1.2 Introducing sequential input processing . . . 26
    - 4.2.1.3 Introducing textual substitution . . . 26
  - 4.2.2 Adapted initial root specification . . . 28
  - 4.2.3 Final root specification . . . 29

4.3 Specializing the root . . . 33
4.3.1 Defining the ProperPrefix function h_{pp} . . . 33
4.3.2 Introducing environment $\mathcal{E}$ . . . 36
4.3.2.1 $f_{\mathcal{R}}$ and $g_{\mathcal{R}}$ . . . 36
4.3.2.2 $f_{\mathcal{R}}$ and $g_{\mathcal{R}}$ related to environment $\mathcal{E}$ . . . 37
4.3.2.3 Defining environment $\mathcal{E}$ . . . 37
4.3.2.4 Using environment $\mathcal{E}$ . . . 38
4.4 Dividing the specialized root in subgroups . . . 41
4.4.1 Subgroup $\mathcal{K} = \mathcal{R}$ . . . 42
4.4.1.1 Introducing $\mathcal{K} = \mathcal{R}$ . . . 42
4.4.1.2 Selecting η (ℓr) . . . 44
4.4.1.3 $\mathcal{E} = \mathcal{D}_{vu}$ and definition of δ_{init} . . . 47
4.4.1.4 Altering the dictionary . . . 47
4.4.1.4.1 $\eta + 1$-addition . . . 48
4.4.1.4.2 (previous η)$+\!\!+\eta$-addition . . . 49
4.4.1.4.3 (previous η)$+\!\!+$(all prefixes η)-addition . . . 50
4.4.1.4.4 "Y"-addition . . . 50
4.4.2 Subgroup $\mathcal{K} = \mathcal{R} \times \Sigma$ . . . 52
4.4.2.1 Introducing $\mathcal{K} = \mathcal{R} \times \Sigma$ . . . 52
4.4.2.2 Selecting η (ℓr) . . . 55
4.4.2.3 Implementing $\mathcal{E}$ . . . 56
4.4.2.3.1 $\mathcal{E} = \mathcal{D}_{ul}$ . . . 56
4.4.2.3.2 $\mathcal{E} = \mathcal{B}$ . . . 57
4.4.3 Subgroup $\mathcal{K} = \mathcal{R} + \Sigma$ . . . 59
4.4.3.1 Introducing $\mathcal{K} = \mathcal{R} + \Sigma$ . . . 59
4.4.3.2 Selecting η (ℓr and $> J$) . . . 62
4.4.3.3 $\mathcal{E} = \mathcal{B}_{lim}$, definition of δ_{init} and addition . . . 65
4.5 Concrete data types for environment $\mathcal{E}$ . . . 67
4.5.1 Dictionary . . . 67
4.5.1.1 Unlimited dictionary . . . 68
4.5.1.2 Limited dictionary . . . 69
4.5.1.2.1 Standard limited dictionary . . . 69
4.5.1.2.2 Virtually unlimited dictionary . . . 69
4.5.2 Search buffer . . . 71
4.5.2.1 Unlimited search buffer . . . 72
4.5.2.1.1 Standard unlimited search buffer . . . 72
4.5.2.1.2 Unlimited search buffer with lookahead . . . 73
4.5.2.2 Limited search buffer . . . 74

5 Conclusions and directions for further work 75
5.1 Conclusions . . . 75
5.2 Directions for further work . . . 77

Bibliography **78**

A Inventory of Lempel-Ziv variants **83**

B List of selected specifications **87**

C Backgrounds of Ziv and Lempel **97**

C.1 Jacob Ziv 97

C.2 Abraham Lempel 98

Summary

This thesis is about the construction of a taxonomy of a family of Lempel-Ziv data compression algorithms. We shall explain each of these terms in turn.

Data compression is used to reduce the size of data whilst maintaining (most of) the information content. Data compression has many diverse applications.

A particular method of data compression has been introduced by Abraham Lempel and Jacob Ziv in [ZL77] and [ZL78]. Some important characteristics of their method are, that it is lossless (i.e. the compressed data can be reconstructed exactly) and universal (i.e. applicable to any kind of data). Some well-known applications of the Lempel-Ziv method are WinZip and `.gif` image compression.

Since the original publications [ZL77, ZL78] by Lempel and Ziv, many variants of their methods have been published and implemented (see Appendix A for an overview), partly to improve the quality of data compression, but partly also to get around patent restrictions by making the new algorithm "sufficiently different". As a result, one can now speak of family of Lempel-Ziv data compression algorithms, which differ in many aspects, but which also have some important characteristics in common. As the various algorithms have been described in very different styles, at different levels of abstraction, and using different nomenclature, it is difficult to get an overview or understanding of the field, the more so since none of these descriptions contain formal proofs of correctness properties.

This thesis attempts to remedy the situation by constructing a taxonomy of the field. In general, a taxonomy is a classification of a collection of elements, based on their properties. Here we use the term in a more technical sense. An algorithm taxonomy is a directed acyclic graph, where each vertex corresponds to an algorithm and each edge corresponds to a discriminator (i.e. an essential detail of an algorithm). The root path of an algorithm node contains all its essential discriminators. This makes it easy to compare algorithms. Correctness of an algorithm follows from the (trivial) correctness of the root algorithm and the fact that each discriminator is added in a correctness preserving way. Some good examples of taxonomies in this sense - and in fact the main sources of inspiration for the present work - are the taxonomies constructed by Jonkers [Jon82] and Watson [Wat95].

This thesis is structured as follows: Chapters 1, 2 and 3 are introductory in nature, dealing with Lempel-Ziv data compression, mathematical preliminaries, and taxonomy construction respectively. Chapter 4 is the core of this thesis: it presents the entire taxonomy and has a substructure that closely matches that of the taxonomy itself. Chapter 5

contains conclusions and directions for future work. Appendix A is an inventory of Lempel-Ziv variants known from the literature. Appendix B contains selected descriptions of the main vertices of the taxonomy. Appendix C provides some personal background of Jacob Ziv and Abraham Lempel.

Chapter 1

Introduction

This chapter first gives a small introduction to compression algorithms in general. Secondly, it gives more specific information about the Lempel-Ziv compression algorithms, including a general textual description. Thirdly, it gives a motivation for making a taxonomy of the Lempel-Ziv compression algorithms. The main structure of the remaining chapters will be presented last.

1.1 Compression algorithms in general

Compression algorithms are all about reducing the size of data. If data has to be transferred –to another place or in time–, then it is desired that the transfer uses as few resources as possible. When data has to be transferred to another place, then the resource is bandwidth, if it has to be transferred in time, then the resource is storage space.

There are many compression algorithms, all with their own specific properties. A brief description of some properties is given next.

- The *compression ratio* is one property. It relates the size of the uncompressed data to size of the compressed data. There are many possibilities to express this ratio. One possibility is to express the size of the compressed data as a percentage of the size of the uncompressed data.

- The compress and decompress algorithm both have their own *time efficiency*. One algorithm may process its input very fast, another may process it slow. The time efficiency of the compress and decompress algorithm are not related. There can for example exist an algorithm that needs much time to compress and needs not much time to decompress.

- The *lossless* and *lossy* properties express to what extent the original data can be reconstructed from the compressed data. Lossless indicates that the reconstructed data is exactly the original. Lossy indicates that the reconstructed data is not exactly the original.

- *Non-universal* and *universal* indicate which kinds of data can be compressed.

  A non-universal algorithm can not compress all kinds of data. It needs some kind of *a priori* knowledge of the data to compress the data. This a priori knowledge is based on a single kind of data. Compression can only be achieved for that single kind of data.

  A universal algorithm can compress all kinds of data. It adapts to the data and it does not need *a priori* knowledge of the data. There are two kinds of universal algorithms, namely the semi-adaptive and the adaptive. A semi-adaptive algorithm first collects knowledge about the data. This knowledge will be used for compression, just like a priori knowledge is. An adaptive algorithm adapts on the fly, while the data is being compressed.

1.2 Lempel-Ziv compression algorithms

First, this section provides a general textual description of the Lempel-Ziv compression algorithms. Next, some reasons will be given why these algorithms are used. Finally, there is an indication of the applications that use these algorithms.

1.2.1 Description

Lempel-Ziv[1] compression algorithms have been named after two persons, Jacob Ziv and Abraham Lempel. They developed the algorithms[ZL77, ZL78] that are the base of all the Lempel-Ziv compression algorithms. Appendix C gives a general background of them.

All Lempel-Ziv algorithms have the following properties

- They are *lossless*.
- They use universal *textual substitution* with only references to the left.
- They process the input *sequentially*.

These properties will be clarified next.

- *Lossless compression* –also named exact or textual compression– can reconstruct the original data exactly. Lossless compression can only achieve compression if the data contains redundancy. If redundancy has been removed, then the size of the data reduces. Consequently, lossless compression is the process of removing redundancy. If more redundancy has to be removed, then more effort has to be made to do so. If the data has no redundancy, then no redundancy can be removed, regardless of

[1] Ziv-Lempel is actually a better name, because Jacob Ziv is mentioned first in the articles that Ziv and Lempel wrote together[ZL77, ZL78]. But this will not be used, because Lempel-Ziv is being used almost everywhere.

how much effort will be made. There is always a trade-off between the amount of redundancy to be removed and the effort to be made.

Lossy compression is the opposite of lossless compression. It is not able to reconstruct the original data exactly, but the reconstructed data is an approximation of the original data. Lossy compression can always achieve compression, even if the data contains no redundancy. This is possible because approximation is used. In theory, everything can be compressed in one single bit with lossy compression. This is possible if it is allowed that the reconstructed data is an extreme approximation of the original data. In practice it is not desirable to use this kind of extreme approximation. Lossy compression is for example applied to digital represented analogue data, which is already an approximation of the original analogue data. Examples of lossy compression are JPEG(pictures) and MPEG(video).

- *Textual substitution* is a method to represent data more compact. It replaces a part of data with a *reference*. The reference refers to another part of the data, which is the *referent*.

  Take for example lossless lossy. This can be represented more compact as lossless [reference to loss]y or [reference to loss]less lossy. In both cases the referent is loss.

  There are different kinds of textual substitution [SS82]. Each kind has its own properties. Two properties are mentioned here explicit.

  - A reference can refer to the left or to the right. If only left references are used, then the referent will occur before the reference occurs. If only right references are used, then the referent will occur after the reference occurs. In the above example the first solution uses a reference to the left. The second uses a reference to the right.
  - Additional references can be used or not. An additional reference does not refer to the data itself, but to some additional predetermined data. Additional references can be seen as references to the left, but also as references to the right.

    As an example assume that ode is an additional predetermined referent. encoder_decoder can be compressed as enc[reference to ode]r_dec[reference to ode]r or even enc[reference to ode]r_de[reference to coder].

  Lempel-Ziv variants only use references to the left and additional references. Because references to the left are used, textual substitution adapts to the symbols on the left. This means that it is universal.

- *Sequential input processing* means that the input is processed once from front to end. It is achieved by repeatedly removing a prefix of the remaining input. Each prefix that is removed results in one output symbol. In this way the output is constructed in such a way that the first n symbols from the input correspond with the first m symbols from the output.

1.2.2 Advantages

Lempel-Ziv compression algorithms have several advantages.

One advantage is that the algorithms are universal. In other words, the algorithms can be used for any kind of data. Other algorithms exist that can not be used on any kind of data.

The good speed/compression ratio is another advantage of Lempel-Ziv variants. They provide ”good” compression in a ”reasonable amount of time”. There are algorithms that can achieve better compression, but they require more time.

A third advantage is that some variants only require a small amount of memory. This is beneficial when the algorithm has to be implemented in hardware. Even if only a small amount of memory is used –say in the order of some kilobytes–, then the compression achieved can still be profitable.

1.2.3 Applications

Many applications make use of Lempel-Ziv compression algorithms. They are being used in both software and hardware. Some applications are:

- WinZip
- WinRAR
- ARJ
- gzip
- PKZIP
- Windows `.cab`-file compression
- `.gif` and `.png` image compression
- V.42bis modem standard

1.3 Problem statement

A problem with the Lempel-Ziv algorithms is that it is difficult to gain a good overview of all the variants.

The description of the Lempel-Ziv algorithms indicates that it is only a small part of the whole area of compression algorithms. Although it is a small part of the whole compression area, there are many Lempel-Ziv algorithms.

One reason why there are so many variants is related to patents. Some algorithms have been patented and may not be used freely. If this is the case, then new – unpatented – variants will be developed. It is possible that the new variant only differs in minor aspects

from the patented version, but it differs so much that it is not considered the same as the patented version. Take for example images in the Graphics Interchange Format(.gif). The algorithm for this image-format has been based on LZW, which is a patented Lempel-Ziv algorithm. A new algorithm with another format has been developed, namely Portable Network Graphics(.png). The new algorithm has been based on LZ77, which is an unpatented Lempel-Ziv algorithm.

Because there are so many Lempel-Ziv algorithms, the overview is lost easily. There are overviews by means of lists – see for example appendix A – but these lists do not give a good overview. That is to say, they are not structured, it are just enumerations of the variants. If there is some kind of structure in the list, then it will be a better overview. Take for example a super market. What if all articles were located at a random place in the super market. It would be very hard to find an article, for example a tomato. Fortunately, the articles have been ordered, there is structure. If you want tomatoes, then you go to the vegetable department. This narrows the search for tomatoes very much.

It is also difficult to gain overview, because the descriptions of the variants are difficult to access. All use different presentation styles and different terminologies. If the description of one variant has been read, then the same amount of time has to be spend on reading another description of another variant. This should not be the case, because the variants have several common properties.

To create a good overview of the Lempel-Ziv algorithms, a taxonomy of the algorithms has been created. The taxonomy gives a structured overview of the algorithms, by indicating the relations between the algorithms. At the top is one algorithm that contains the elementary properties of the Lempel-Ziv algorithms. Different properties can be added to create different groups. Even more properties can be added to these groups to create subgroups. This can be repeated until each group contains only one –or a few– variants.

The construction of the taxonomy for Lempel-Ziv algorithms belongs to one big project, namely the TAxonomy-BAsed Software COnstruction (TABASCO) project. This project is the primary research subproject within the Software Construction group at the Eindhoven University of Technology. The aim of TABASCO is to create order in a specific problem area. In this particular case it is the area of Lempel-Ziv algorithms.

The variants to be included have been limited in advance, because there are very much variants.

One limitation is that the variants with postprocessing have not been included. Each variant with postprocessing is a variation on another variant that uses no postprocessing. The only difference is that a special postprocessing step is being used, with the goal to compress the output of the non-postprocessing variant even more. This postprocessing can even be based on the knowledge of the output of the non-postprocessing variant.

Another limitation is that only abstract descriptions of the algorithms have been used in the taxonomy. This hides certain details, such as natural number representation. In fact, there are many methods to represent natural number on the bit-level[BCW90, appendix A]. This could form a taxonomy on itself. The use of the abstract descriptions makes it

possible to give very compact algorithms. The abstract descriptions also allow the use of non-deterministic statements.

1.4 Document structure

The general structure for the remaining chapters will be clarified in this section.

The second chapter contains mathematical and notational preliminaries. The chapter consists out of two parts. The fist part contains the elementary concepts and notations. The second part contains additional concepts and notations. This chapter only gives notations and definitions that can be used in a general way. Definitions only related to the Lempel-Ziv class are not included in this chapter.

Chapter three explains how a taxonomy can be constructed. It explains that the taxonomy is constructed bottom-up. It also explains that it is presented top-down.

The resulting taxonomy of Lempel-Ziv compression algorithms will be presented in chapter four. It first gives a general specification. Secondly, it gives more detailed specifications, which eventually leads to the abstract representations of the actual Lempel-Ziv variants.

The last chapter –chapter five– contains conclusions and directions for further work.

Chapter 2

Preliminaries

This chapter specifies the general notations and definitions that are used in subsequent chapters. Definitions which are only related to the Lempel-Ziv algorithms are not included.

The first section mainly introduces notational matters for elementary concepts, including the notation used for algorithms. The second section defines additional concepts and notations.

2.1 Elementary concepts and notations

This section contains the elementary concepts and their notation. It includes the nomenclature and the notation for algorithms. Not many definitions are given in this section. The main goal is to introduce the notation used.

Notation 2.1 (Nomenclature)
The nomenclature is listed in table 2.1 and 2.2. These tables list the nomenclature for the constants and variables respectively. The symbols f, g and h indicate functions. Propositions are indicated with the symbols P, Q and R. □

Definition 2.2 (Alphabet)
An alphabet is a nonempty finite set. Elements of an alphabet are also called characters or symbols. □

Definition 2.3 (Countable set)
A countable set is an infinite set for which each element can be placed in a one-to-one relation with an element of $\mathbb{N}$. In other words, the elements of the set can be written in some order $a_0, a_1, a_2, ...$ in such a way that each element of the set has a finite index. □

Definition 2.4 (String)
A string over Σ is a finite sequence of symbols from an alphabet Σ. Strings can be defined explicitly by use of symbol juxtaposition. For example, the string xxyxz is the sequence of the symbols x, x, y, x and z. □

Table 2.1: Constants nomenclature

symbol	meaning
Constant sets:	
• $\mathcal{T}, \mathcal{V}, \mathcal{W}$	general constant set
• $\mathbb{N}$	natural domain ($\{0, 1, 2...\}$)
• $\mathbb{B}$	boolean domain ($\{\mathsf{true}, \mathsf{false}\}$)
• Σ	alphabet
• $\mathcal{K}, \mathcal{R}$	countable set
Constant elements:	
• A, B	general constant element ($\in \mathcal{V}$)
• I, J, K, M, N	natural number constant ($\in \mathbb{N}$)
• S, T	string constant ($\in \mathcal{V}^*$)

Table 2.2: Variables nomenclature

symbol	meaning
V, W	general set ($\in \mathcal{P}.V$)
i, j, k, m, n	natural number ($\in \mathbb{N}$)
a, b, c, d	single element ($\in V$)
$\alpha, \beta, \gamma, ...$	string ($\in V^*$)

Notation 2.5 (Elementary logic, set and string symbols)
Table 2.3, 2.4 and 2.5 list the symbols for elementary concepts. The definitions of these concepts are assumed to be known. The tables list the logic concepts, set concepts and string concepts respectively. □

Notation 2.6 (Quantification)
Quantification is denoted as $\langle \bigoplus a : R.a : f.a \rangle$. It is a way to express $e_\oplus \oplus f.a_1 \oplus f.a_2 \oplus \ldots \oplus f.a_n$, if it is assumed $R.a$ is valid for n values of a and each value of a for which $R.a$ is valid is projected on a unique a_i. $e_\oplus$ is the unit for operator $\oplus$ ($b \oplus e_\oplus = b$).

Table 2.6 lists the quantified operators. Operator $\oplus$ is included for clarity. □

Example 2.7 (Quantification)
The sum of all numbers between 1 and 4(including 1 and 4) is $\langle \sum i : 1 \leq i \leq 4 : i \rangle$, which is $0 + 1 + 2 + 3 + 4 = 10$. □

Example 2.8 (Quantification)
"All squared natural numbers are at least 0" can be formalized as $\langle \forall i : i \in \mathbb{N} : i^2 \geq 0 \rangle$. This is equal to $\mathsf{true} \wedge (0^2 \geq 0) \wedge (1^2 \geq 0) \wedge (2^2 \geq 0) \wedge \ldots$ □

Table 2.3: Elementary logic concepts

symbol	meaning	example
$\wedge$	conjunction (*and*)	$a \wedge b$
$\vee$	disjunction (*or*)	$a \vee b$
$\neg$	negation (*not*)	$\neg a$
$\Rightarrow$	implication (*implies*)	$a \Rightarrow b$

Table 2.4: Elementary set concepts

symbol	meaning	example
$\emptyset$	empty set	$\emptyset$
$\cup$	union	$V \cup W$
$\cap$	intersection	$V \cap W$
$\setminus$	difference	$V \setminus W$
$\times$	Cartesian product	$\mathcal{V} \times \mathcal{W}$
$\mathcal{P}$	powerset	$\mathcal{P}.V$
$\subseteq$	subset	$V \subseteq W$
$\supseteq$	superset	$V \supseteq W$
$\mid \; \mid$	set size	$\lvert V \rvert$
$\in$	element of	$a \in V$
$\notin$	not an element of	$a \notin V$
$\{...\}$	set definition	$\{1, 2, 3\}$

Table 2.5: Elementary string concepts

symbol	meaning	example
λ	empty string	λ
Σ^i	set with all strings over Σ of length i	Σ^3
Σ^*	set with all strings over Σ of all lengths	Σ^*
$\langle \; \rangle$	singleton string	$\langle a \rangle$
$+\!\!+$	string concatenation	$\alpha +\!\!+ \beta$
$\mid \; \mid$	string length	$\lvert \alpha \rvert$

Table 2.6: Quantified operators

operator symbol	quantified operator symbol	unit
$\oplus$	$\bigoplus$	$e_{\oplus}$
$\wedge$	$\forall$	true
$\vee$	$\exists$	false
$\cup$	$\bigcup$	$\emptyset$
$+$	$\sum$	0

Notation 2.9 (Domain and range)
The domain of function f is denoted as $dom.f$. The range of a function f is denoted as $range.f$. □

Definition 2.10 (Total function)
A total function f is a function with signature $f : \mathcal{V} \rightarrow \mathcal{W}$, with requirements

- $dom.f = \mathcal{V}$
- $range.f \subseteq \mathcal{W}$

□

Definition 2.11 (Partial function)
A partial function f is a function with signature $f : \mathcal{V} \nrightarrow \mathcal{W}$, with requirements

- $dom.f \subseteq \mathcal{V}$
- $range.f \subseteq \mathcal{W}$

□

Remark 2.12 (Total and partial function)
Every total function can be seen as a partial function, but not every partial function can be seen as a total function. □

Notation 2.13 (Function application)
$f.a$ is the notation for applying argument a to function f. Other literature sometimes uses $f(a)$. Function application has the highest priority. □

Definition 2.14 (Function composition)
The composition of function f and g is denoted as $f \circ g$. The $\circ$-symbol is the "compose" symbol. $(f \circ g).a$ is defined as $f.(g.a)$. □

Definition 2.15 (Identity function)
The identity function id is defined as $id.a = a$. a can be of any type. □

Notation 2.16 (Function specification)
From a set theoretic point of view, a function f with domain-type $\mathcal{V}$ and range-type $\mathcal{W}$ is a subset of $\mathcal{V} \times \mathcal{W}$. Consequently, a function can be specified with a set containing tuples. □

Example 2.17 (Function specification)
Let function $f : \mathbb{N} \nrightarrow \mathbb{N}$ be specified with $f = \{(1,4),(3,8),(0,5)\}$. This means that –for instance– $f.3 = 8$. This specification also provides the domain and range of f, namely $dom.f = \{0,1,3\}$ and $range.f = \{4,5,8\}$. □

Notation 2.18 (Algorithm notation)
The Guarded Command Language (GCL) –which was first defined by Dijkstra [Dij76]– is used for the notation of algorithms. There are several reasons to use GCL, such as the plain and simple notation and the inclusion of nondeterminism. It is extended with **let** as described in definition 2.20. Assertions are used to prove properties of an algorithm. An assertion is denoted as a predicate surrounded with curly braces. Queries –assertions that are not yet proven– start the predicate with a question mark [FvG99, p. 98]. □

Example 2.19 (Queried assertions)
Let the following algorithm fragment be given:

$\{\ i = 0\ \}$
$i : = i + 1\{\ ?\ i \geq 1\ \}$

This indicates that the assertion $i = 0$ has already been proven, and that assertion $i \geq 1$ has to be proven. The proof of this particular queried assertion can be given with standard techniques. $i \geq 1$ is valid if the assertion $i + 1 \geq 1$ is valid just before the assignment, which is the case because $i = 0$ is valid. □

Definition 2.20 (let-statement)
Let $\overline{v}$ define a list with fresh variables and let P denote a predicate that depends on $\overline{v}$. The **let**-statement is defined as

$\langle \exists \overline{v} :\ : P.\overline{v} \rangle$
let $\overline{v}$ **such that** $P.\overline{v}$
$\{\ P.\overline{v}\ \}$

All variables in the variable list have to be assigned a value, in such a way that $P.\overline{v}$ will be valid. The assignment of the values is nondeterministic, because there can be many assignments that satisfy $P.\overline{v}$.

The precondition guarantees that $P.\overline{v}$ can be satisfied, which can be seen as a guarantee that the **let**-statement "terminates". □

Notation 2.21 (Alternative let notation)
Because one particular form is used frequently, an alternative –shorter– form is introduced for that particular form. This alternative notation is given with use of an example. The alternative notation is

let $\alpha :: \beta +\!\!+ \gamma$ **such that** $|\beta| < |\gamma|$
Which is equal to
let β, γ **such that** $\alpha = \beta +\!\!+ \gamma \wedge |\beta| < |\gamma|$
β and γ are the fresh variables, both with the same type as α. □

Example 2.22 (let-statement)
To emphasis that the **let**-statement is nondeterministic, the following fragment is given:
$\{\ \alpha = \mathsf{abc}\ \}$
let β, γ **such that** $\alpha = \beta +\!\!+ \gamma \wedge |\beta| < |\gamma|$
This can result in two (β, γ) pairs, namely (λ, abc) and $(\mathsf{a}, \mathsf{bc})$. □

2.2 Additional concepts and notations

This section gives additional concepts and notations to the elementary ones. Note that some additions are not visually used in subsequent chapters. They are used here to specify other additions.

Definition 2.23 (Disjunct sum)
The disjunct sum combines two sets $\mathcal{V}$ and $\mathcal{W}$ to a new set $\mathcal{T}$. Each element of $\mathcal{T}$ contains an element of $\mathcal{V}$ or $\mathcal{W}$. If an element of $\mathcal{T}$ is taken, it can be determined whether it is an element of $\mathcal{V}$ or $\mathcal{W}$. It is defined as $\mathcal{T} = \mathcal{V} + \mathcal{W} = \{in1.v | v \in \mathcal{V}\} \cup \{in1.w | w \in \mathcal{W}\}$ with $in1 : \mathcal{V} \rightarrow \mathcal{V} + \mathcal{W}$ and $in2 : \mathcal{W} \rightarrow \mathcal{V} + \mathcal{W}$ ($in1$ and $in2$ are so called "injection" functions). □

Definition 2.24 (Function-junc ∇)
Let $f : \mathcal{V} \rightarrow \mathcal{T}$, $g : \mathcal{W} \rightarrow \mathcal{T}$. The function-junc then has the signature $f \nabla g : \mathcal{V} + \mathcal{W} \rightarrow \mathcal{T}$. The function $f \nabla g$ is defined by:
$(f \nabla g).(in1.a) = f.a \qquad a \in \mathcal{V}$
$(f \nabla g).(in2.b) = g.b \qquad b \in \mathcal{W}$
□

Definition 2.25 (Minimum and Maximum)
The functions $\downarrow, \uparrow : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ are defined as

- minimal-operator $\downarrow$: $\quad a \downarrow b = \begin{cases} a & \text{if } a \leq b \\ b & \text{if } b \leq a \end{cases}$
- maximal-operator $\uparrow$: $\quad a \uparrow b = \begin{cases} a & \text{if } b \leq a \\ b & \text{if } a \leq b \end{cases}$

□

Definition 2.26 (String operators $\rceil, \rfloor, \lceil, \lfloor$)
These operators give a portion of the original string. $\rceil, \rfloor, \lceil, \lfloor : \Sigma^* \times \mathbb{N} \rightarrow \Sigma^*$. $\alpha \rceil i$ and $\alpha \lfloor i$ both give a prefix of α. $\alpha \rfloor i$ and $\alpha \lceil i$ both give a suffix of α.

- take-left-operator $\upharpoonleft$: $\alpha \upharpoonleft i$ denotes the $i \downarrow |\alpha|$ leftmost elements of α
- drop-left-operator $\downharpoonleft$: $\alpha \downharpoonleft i$ denotes the $(|\alpha| - i) \uparrow 0$ rightmost elements of α
- take-right-operator $\upharpoonright$: $\alpha \upharpoonright i$ denotes the $i \downarrow |\alpha|$ rightmost elements of α
- drop-right-operator $\downharpoonright$: $\alpha \downharpoonright i$ denotes the $(|\alpha| - i) \uparrow 0$ leftmost elements of α

The priority of these operators is just below the priority of $+\!\!+$. Note that these functions need an $i \in \mathbb{N}$, thus i cannot be negative. □

Property 2.27 (String operators $\upharpoonleft, \downharpoonleft, \upharpoonright, \downharpoonright$)

- $(\alpha \upharpoonleft i) +\!\!+ (\alpha \downharpoonleft i) = \alpha$
- $(\alpha \downharpoonright i) +\!\!+ (\alpha \upharpoonright i) = \alpha$ □

Example 2.28 (String operators $\upharpoonleft, \downharpoonleft, \upharpoonright, \downharpoonright$)

$\mathsf{xxyxz} \upharpoonleft 2 = \mathsf{xx}$, $\mathsf{xxyxz} \downharpoonleft 3 = \mathsf{xz}$, $\mathsf{xxyxz} \upharpoonleft 100 = \mathsf{xxyxz}$, $\mathsf{xxyxz} \downharpoonright 23 = \lambda$, $\mathsf{xxyxz} \upharpoonright 3 = \mathsf{yxz}$. □

Definition 2.29 (Prefix)

α is called a prefix of β, denoted as $\alpha \preccurlyeq_p \beta$ if $\langle \exists \gamma : \gamma \in \Sigma^* : \alpha +\!\!+ \gamma = \beta \rangle$. □

Definition 2.30 (Substring)

α is called a substring of β if $\langle \exists \gamma, \delta : \gamma, \delta \in \Sigma^* : \gamma +\!\!+ \alpha +\!\!+ \delta = \beta \rangle$, and is denoted as $\alpha \preccurlyeq_s \beta$. □

Property 2.31 (Substring)

$\alpha \preccurlyeq_s \beta = \langle \exists i : \ : \alpha \preccurlyeq_p (\beta \downharpoonleft i) \rangle$ □

Definition 2.32 (Prefix closed with base n)

A set of strings V is prefix closed with base n if

- $\langle \forall \alpha : \alpha \in V : |\alpha| \geq n \rangle$
- $\langle \forall \alpha : \alpha \in V \wedge |\alpha| > n : (\alpha \downharpoonright 1) \in V \rangle$ □

Definition 2.33 (Prefix closed)

A set of strings is prefix closed if it is prefix closed with base 0. □

Definition 2.34 (new)

new : $\mathcal{V} \to \mathcal{V}$, where $\mathcal{V}$ is any type for which the =-operation is defined. The sequence $a_0, a_1, ..., a_n$ ($\langle \forall i : 0 \leq i \leq n : a_i \in \mathcal{V} \rangle$) produced by repeatedly using **new**.a ($a \in \mathcal{V}$) has two requirements:

- $\langle \forall i, j : 0 \leq i < j \leq n : a_i \neq a_j \rangle$ (all elements in the sequence are unique)
- The sequence produced by **new**.a is always the same. □

Remark 2.35 (new)

One possibility is to combine **new** with $\mathbb{N}$. This is easy implementable if the sequence $0, 1, 2, ...$ is chosen. □

Chapter 3

Constructing and presenting taxonomies

This chapter describes how a taxonomy can be constructed and presented. First the construction is described, thereafter the presentation is described. Most of the things described here are also described in other taxonomies[Wat95, chapter 3][Jon82].

3.1 Constructing taxonomies

A taxonomy is constructed bottom-up. The process can be divided in two steps.

The first step collects algorithms that appear in literature for the problem area of interest. These algorithms – or a selection of them – are rewritten in a uniform way, all with the same level of abstraction. Several details may even be removed by the abstraction, such as implementation details.

The next step is the generalization step. Two or more algorithms are generalized to one new algorithm. The new algorithm preserves the common properties of the generalized algorithms and it removes the properties that make them different. These properties that make them different are named *discriminators*. This process includes rewriting several algorithms is such a way that they fit in one general algorithm.

These steps may sound very simple, but in practice they are not so simple. One has to search for the right level of abstraction. Many versions of an algorithm may have to be made before the right abstraction level is reached. If a too low level of abstraction is used, then the common properties may not be clear at all. Abstraction makes the essential things easier recognizable. The rewriting of the algorithms of the second step is also very difficult. In some cases it takes a very creative and inventive mind.

The result of the generalization process is in general a directed acyclic graph[Wat95]. Each vertex in the graph represents an algorithm. This can be a generalized algorithm or an algorithm out of literature. Each edge represents a discriminator. The visual presentation of this graph is given in 3.2.2.

The construction of the Lempel-Ziv taxonomy started with four well documented variants, namely LZ77[ZL77], LZ78[ZL78], LZSS[Bel86] and LZW[Wel84]. Rewriting these algorithms in a uniform way took much effort. Take for example LZ77, it is originally not even given as an algorithm, it is given in a textual format. Recognizing the common properties was even harder. Many rewriting steps were made before the algorithms could be generalized. Many attempts for generalization did not work out. Eventually it resulted in a taxonomy with a tree-shape.

3.2 Presenting taxonomies

This section describes issues related to presenting taxonomies. Firstly, issues for the presentation in general are described. Secondly, it is described how a visual summary of a taxonomy can be given. Thirdly, it is described how a (generalized) algorithm is presented.

3.2.1 Presentation in general

A taxonomy is presented top-down. At the top is an algorithm that contains the elementary properties of the various algorithms. Distinct discriminators are added to create different groups. Even more discriminators are added to these groups to create subgroups. Discriminators are added until the abstract representations of the variants emerge.

Note that it is possible that some discriminators can be added in various orders. The order that is chosen is a matter of taste. A different order results in another taxonomy graph. One graph may look more elegant than another.

A reason to use top-down presentation is that a property can be proven for a general algorithm, somewhere in the top. With that property proven, it is not needed to prove that property for the descendants, if it is guaranteed that the descendants preserve correctness.

3.2.2 Presenting a visual summary of a taxonomy

A visual summary of a taxonomy can be given by means of a figure that contains the taxonomy graph. Additional information can be added with labels. A vertex – which represents an algorithm in the taxonomy – can be labelled with a number that refers to the actual algorithm. An edge – which represents a discriminator – can be labelled with a mnemonic to refer to the discriminator.

An example of a visual summary of a taxonomy with a shape of a tree is given in figure 3.1, where "discr" as an abbreviation for discriminator.

3.2.3 Presenting an algorithm from the taxonomy

In this subsection it is explained how an algorithm from the taxonomy is presented. Because the algorithms can be incomplete – that is to say, it is not totally specified – a special

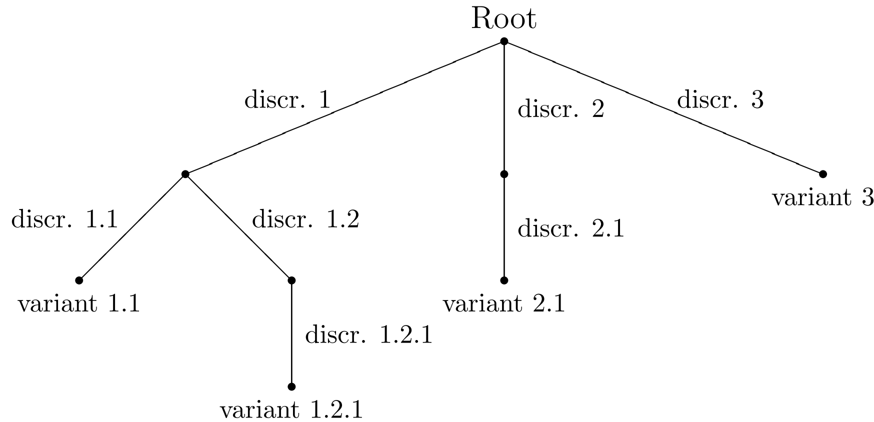


Figure 3.1: Example of a visual summary of a taxonomy

specification format is introduced(Notation 3.1). This format is defined in such a way that the incomplete algorithm can be refined easily, as clarified in Remark 3.3. An example of a specification is given in Example 3.4. Remark 3.8 explains why an algorithm is not suitable for specification.

Notation 3.1 (Specification format)
The notation to give a specification of one or more possibly incomplete algorithms in such a way that it is easily refinable has the following format:

Specification 3.2 (Format example)

Declarations:

Definitions:

Assumptions:

Lemmas:

Algorithms:

Each part is clarified next:

- Declarations

  This part contains the declarations of sets and functions. The definitions of the sets and functions are given in the definition part.

- Definitions

  This part contains definitions for the declarations. It is possible that not all declarations are defined. In that case the specification is said to be incomplete. *The not*

defined declarations are the only things that remain to be defined. There are no other things anywhere else in the specification to be defined.

The definitions may not violate the assumptions. Otherwise an assumption would be invalid, and false has to be assumed as a precondition. Assuming false means in essence that the total is nonsense.

- Assumptions

  This part contains assumed –thus unproven– predicates. These predicates can contain undefined sets or functions, consequently it may even occur that they are currently not provable. If all declarations are defined, then the assumptions must only contain easily verifiable assumptions.

- Lemmas

  This part contains proven predicates. Because assumptions are predicates, it may also contain proven assumptions. A proven assumption is actually no assumption any more. Consequently, it becomes a lemma if it is proven. Lemmas may use assumptions and other lemmas for the proof of their correctness.

- Algorithms

  This part contains special definitions. It contains algorithms that are used to define declarations.

  This part is not placed directly after the definition-part, but at the end of the specification. This is done because the algorithm-part can be viewed as the "goal". The algorithm-part contains the algorithms that will be implemented.

  If this part is not placed at the end, then the algorithms will be encountered somewhere in the middle of the specification. But if the algorithms are read, why should you read any further? The algorithms do lead to the implementation. They can be viewed as the reason to read the specification. In fact, that is not totally true, the other parts are as important as the algorithms. Take for example the lemmas. The lemmas will probably give the properties that hold for the algorithms. This assures that the algorithms are correct, which is quite important.

  To prohibit that the reading of the specification is stopped after the algorithms are encountered, the algorithms are placed at the end.

  Note that the algorithms are only implementable if all the declarations are defined.

All the parts have the same appearance. Each part starts with an appropriate heading and is followed by the items belonging to that part. Each item –except the algorithms– also has the same appearance. Each item contains

- Item name.

- Item "freshness". The "+" sign indicates that this item is fresh, it is new in comparison with the former specification.

- Item content. This contains the actual content. For example, this could be a definition.
- Item reference. This contains a reference to a further explanation in the text. For example, if the item is a lemma, then this contains a reference to the proof of the lemma.

The specification is given in such a way that it can be read top down. Each item only uses items that are listed earlier. □

Remark 3.3 (Adding refinements to specifications)
Refinements can be added to a specification by giving new definitions and proving assumptions.

The addition of refinements can be described for only definitions, because there is a similarity between the assumption/lemma parts and the declaration/definition parts. The assumption/lemma parts contain respectively unproven and proven predicates. Declarations can be viewed as unproven, because the actual definition has not yet been given. The definition is the "proof".

A refinement of a declaration is a definition. A definition can be given with use of declared – and possibly undefined – sets or functions. These declarations can even be introduced especially for this definition. This enables stepwise refinement, which is a useful property when constructing a taxonomy. With stepwise refinement it is possible to give a difficult definition, using other declarations, which are possible not defined. Stepwise refinement is successful if the possible not defined declarations are easier definable. One difficult definition is given, and some less difficult definitions have to be given. □

Example 3.4 (Specification)
Take a function that splits a string in two parts. It is not yet know how the string has to be split exactly, it is only known that it has to be split.

A specification of this looks like:

Specification 3.5 (Specification example; Step 1)

Declarations:

Split	$\mathbf{Split} : \Sigma^* \to \Sigma^* \times \Sigma^*$

Definitions:

Assumptions:

ProperSplit	$\langle \forall \alpha, \beta, \gamma : (\beta, \gamma) = \mathbf{Split}.\alpha : \beta +\!\!+ \gamma = \alpha \rangle$

Lemmas:

Algorithms:

A definition for **Split** can be given now[1]. With this definition, the proof for the as-

[1] Many correct definitions exist, one is chosen here. In a taxonomy this would be a discriminator.

sumption can be given. The new specification becomes:

Specification 3.6 (Specification example; Step 2)

Declarations:

Split $\quad$ $\textbf{Split} : \Sigma^* \rightarrow \Sigma^* \times \Sigma^*$

Definitions:

$+ \quad \textbf{Split}.\alpha = (\alpha \upharpoonleft 1, \alpha \downharpoonleft 1)$

Assumptions:

Lemmas:

ProperSplit $\quad + \quad \langle \forall \alpha, \beta, \gamma : (\beta, \gamma) = \textbf{Split}.\alpha : \beta +\!\!+ \gamma = \alpha \rangle \quad (3.7)$

Algorithms:

Proof 3.7 (ProperSplit)
For all α, β and γ with $(\beta, \gamma) = \textbf{Split}.\alpha$ holds that

$$\beta +\!\!+ \gamma$$

$$\equiv \quad \{(\beta, \gamma) = \textbf{Split}.\alpha, \text{ thus } (\beta, \gamma) = (\alpha \upharpoonleft 1, \alpha \downharpoonleft 1)\}$$

$$\alpha \upharpoonleft 1 +\!\!+ \alpha \downharpoonleft 1$$

$$\equiv \quad \{\text{property } \upharpoonleft, \downharpoonleft \text{ (2.27)}\}$$

$$\alpha$$

□

□

Remark 3.8 (Specification vs algorithm)
An algorithm in isolation is not suitable for the notation of incomplete algorithm. Example 3.4 illustrates an incomplete algorithm. That is to say, the definition of P is not given, only the declaration.

Assume that only the algorithm is given. There are two possibilities:

- Because P is not declared, it can not be used. Consequently, $P.\beta$ has to be replaced with something like "..." or some pure text. This mixes formal and informal, which is not desirable.

- To allow the usage of $P.\beta$, the declaration can be given in the text. But if the algorithm is repeated some pages later, then reader has to search for the declaration, which is not desirable.

In conclusion, the declaration has to be given, and it has to be given each time. This is precisely what the specification does. $\Box$

Chapter 4

Taxonomy

This chapter presents a taxonomy of Lempel-Ziv compression algorithms. An overview of the total taxonomy is presented first. The detailed presentation starts with the specification of the root, which includes the encode and decode algorithm. It is followed with more detailed specifications. These specifications lead to the abstract representations of the actual Lempel-Ziv variants.

Note that the encode and decode algorithm are also called encoder and decoder respectively.

4.1 Overview

An overview of the taxonomy of the Lempel-Ziv compression algorithms is given in figure 4.1 on page 24. This figure could be used as an alternative table of contents of sections 4.3 and 4.4.

Each vertex in the figure corresponds to an algorithm. If the vertex is labelled with a number, then that number refers to an algorithm given in this chapter. A vertex with the label at the bottom indicates an algorithm that appears in the literature. The label gives the acronym for that algorithm as found in literature. The acronyms of all algorithms found in literature are listed in A. Each edge in the figure corresponds with a discriminator. The labels of the edges are mnemonics that refer to the discriminators. These mnemonics are clarified in table 4.1 on page 24.

4.2 Root specification

This section contains the specification for the root. The specification will be given in three steps. Firstly, an initial root specification is constructed on basis of the textual specification in §4.2.1. Secondly, the initial specification is adapted in §4.2.2. This is done to make further specification easier. Thirdly, the adapted specification is transformed in the final root specification in §4.2.3.

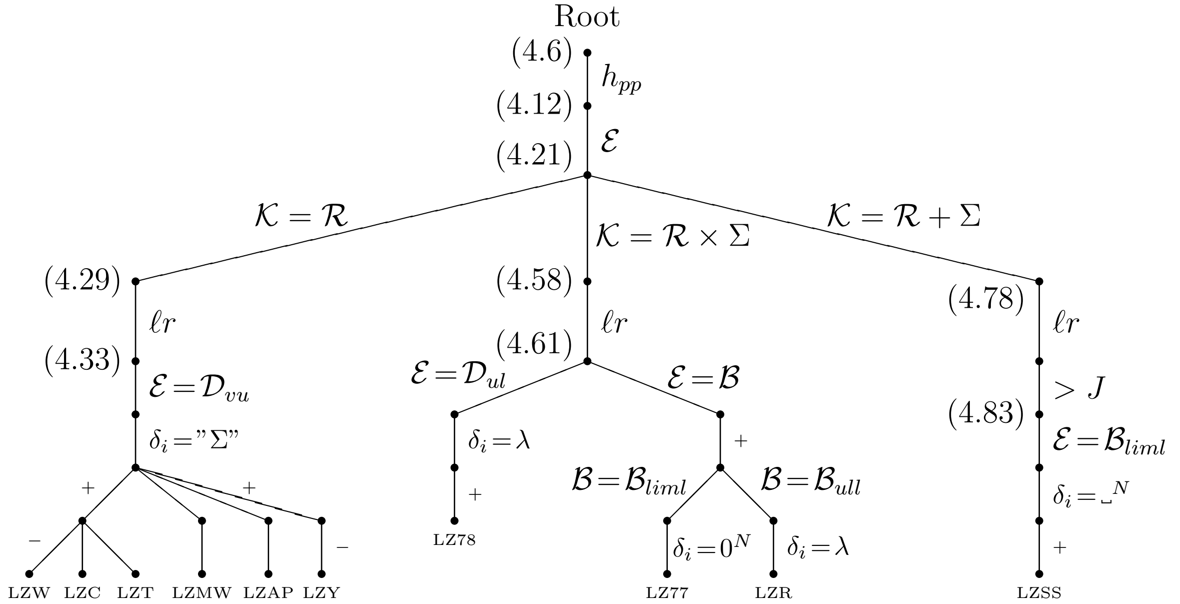


Figure 4.1: Visual summary of the taxonomy of Lempel-Ziv Compression Algorithms

Mnemonic	Explanation
h_{pp}	The encoder processes its input by repeatedly removing a prefix of the remaining input. The function h_{pp} indicates a set containing all the so called "proper prefix". That are prefixes that can be processed. The discriminator h_{pp} gives a possible definition for this function.
$\mathcal{E}$	$\mathcal{E}$ stands for the abstract data type environment. This environment is a method to handle referents and references. Referents and references will be heavily used, because they are the base of textual substitution.
$\mathcal{K} = ...$	$\mathcal{K}$ is the output type of the encoder. These discriminators gives a definition of that type in terms of the reference type $\mathcal{R}$ and Σ.
ℓr	The encoder has to select one of the proper prefixes given by set h_{pp}. This discriminator narrows this set down. It eliminates all the prefixes that are based on referents, except one. The prefix that contains the longest referent – that is to say, the longest referent present in the set – is not eliminated.
$> J$	Just like ℓr, this discriminator narrows the set h_{pp} down. It eliminates all the prefixes that are based on referents, except those with a length longer than J.
$\mathcal{E} = ...$	These discriminators give an implementation for the abstract data type $\mathcal{E}$. Two main implementations are used, the dictionary $\mathcal{D}$ and the search buffer $\mathcal{B}$.
$+, -$	These discriminators indicate how the environment has to be changed. The environment has to be changed, because it contains the referents and references, which change while the input is being processed. The $+$ and $-$ indicate respectively addition and removal.
$\delta_i = ...$	These discriminators indicate how the environment is initialized.

Table 4.1: Mnemonics for the discriminators

4.2.1 Initial root specification

The initial root specification is constructed step-by-step on basis of the textual specification. The textual specification – as given in §1.2.1 – is restated here for ease of reading:

> All Lempel-Ziv algorithms have the following properties
>
> - They are lossless.
> - They use universal textual substitution with only references to the left.
> - They process the input sequentially.

Each step of the construction introduces one of the properties of the textual specification. The last step – which is located in §4.2.1.3 – gives the initial root specification.

The construction steps are the subject of the next subsubsections.

4.2.1.1 Introducing lossless

This first construction step introduces the lossless property. This specification is one of the simplest possible, because no definition of the encoder and decoder is needed, only the declarations are needed.

The encoder has to be a total function, because it has to be able to encode every possible string. The decoder can be a partial function. The reason why it can be partial consists out of two parts. Firstly, the decoder only has to decode something that is produced by an encoder. It is not logical to decode something that is not produced by an encoder. Secondly, it is possible that the encoder can not produce all output strings, which means that the range of the encoder is limited. The combination of these two parts implies that the domain of the decoder can be limited, which means that the decoder can be a partial function.

Specification 4.1 (Construction step 1)

Declarations:

Input	$\Sigma \in$ Alphabet
Output	$\mathcal{K} \in$ Countable Set
Encode	$\mathbf{encode} : \Sigma^* \to \mathcal{K}^*$
Decode	$\mathbf{decode} : \mathcal{K}^* \nrightarrow \Sigma^*$

Definitions:

Assumptions:

Lossless	$\mathbf{decode} \circ \mathbf{encode} = \mathsf{id}$

Lemmas:

Lossless	$\mathbf{decode} \circ \mathbf{encode} = \mathsf{id}$	(4.2)

Algorithms:

Proof 4.2 (Lossless)
Because Lossless is assumed, it can be concluded as a lemma. □

Remark 4.3 (Lossless)
Assuming Lossless to conclude Lossless as a lemma may seem silly. But this is the only possibility at the moment to conclude the lemma, there is nothing else available where the proof can be based on. Another proof – which does does not assume Lossless – is given in §4.2.3. It is possible to give another proof there, because the specification included there contains a definition of the encoder and decoder. □

4.2.1.2 Introducing sequential input processing

This second construction step introduces sequential input processing. Sequential input processing – in the form of repeatedly removing a prefix of the remaining input – requires that the body of the encoder and the decoder have a particular form. This form is given by the following algorithm fragment, where α is the input and β is the output.

do $\alpha \neq \lambda \rightarrow$
  let γ **such that** $\gamma \preccurlyeq_p \alpha \wedge \gamma \neq \lambda$;
  $\beta :=\beta +\!\!+$"encoding/decoding of γ";
  $\alpha :=\alpha \downharpoonright |\gamma|$
od

Because it is not yet known how "encoding/decoding of γ" is stated formally, no formal specification of the encoder/decoder can be given.

4.2.1.3 Introducing textual substitution

This third construction step introduces textual substitution, with only references to the left. It introduces one function for the encoder and one function for the decoder. These two functions are the formal version of "encoding/decoding of γ" from the previous step.

The function $f : \Sigma^* \rightarrow \Sigma^* \nrightarrow \mathcal{K}$ is introduced for the encoder[1]. The function incorporates textual substitution with references to the left. References to the left can refer to the processed part of the input. This processed part of the input –which will be named σ in the algorithm– will be an argument of f. $f.\sigma$ is used to encode a prefix of the remaining input.

The type for references – $\mathcal{R}$ – is somehow embedded in $\mathcal{K}$. The exact embedment does not matter yet, it will be given later.

[1]A function with signature $\Sigma^* \nrightarrow \mathcal{K}$ could be used as a first specification, but it does not allow adaption.

With the inclusion of textual substitution it can be explained why $f.\sigma$ is not a total function. The domain of $f.\sigma$ is namely in some way based on the referents. But not every string is a referent. Consequently, the domain of $f.\sigma$ can be limited and $f.\sigma$ has to be a partial function.

The function $g : \Sigma^* \to \mathcal{K} \not\twoheadrightarrow \Sigma^*$ is introduced for the decoder. The first argument is needed to decode references, which are embedded in $\mathcal{K}$. To decode a reference, the decoded string before that reference – named σ in the algorithm – is enough to decode the reference. It is enough to known only σ, because only references to the left are used. $g.\sigma$ is used to decode the first symbol of the remaining input. Only one symbol at a time is decoded, because the whole decoded string before a reference is needed for decoding.

$g.\sigma$ is not a total function, because references are included in $\mathcal{K}$. It is possible that the reference can not be decoded, because it refers to a referent that does not exist.

The specification becomes

Specification 4.4 (Construction step 3; Initial Root)

Declarations:

Input		$\Sigma \in$ Alphabet
Output		$\mathcal{K} \in$ Countable Set
Encode		**encode** $: \Sigma^* \to \mathcal{K}^*$
Decode		**decode** $: \mathcal{K}^* \not\twoheadrightarrow \Sigma^*$
Substitute	+	$f : \Sigma^* \to \Sigma^* \not\twoheadrightarrow \mathcal{K}$
DeSubstitute	+	$g : \Sigma^* \to \mathcal{K} \not\twoheadrightarrow \Sigma^*$

Definitions:

Assumptions:

Lossless $\quad$ **decode** $\circ$ **encode** $= \mathsf{id}$

Lemmas:

Lossless $\quad$ **decode** $\circ$ **encode** $= \mathsf{id}$ $\quad$ (4.2)

Algorithms:

encode.σ_0 =
$[\![$ **var** $\sigma, \sigma' : \Sigma^*;\ \kappa : \mathcal{K}^*$
| $\sigma', \kappa, \sigma :\ = \sigma_0, \lambda, \lambda;$
do $\sigma' \neq \lambda \rightarrow$
let η **such that** $\eta \preccurlyeq_p \sigma' \wedge \eta \neq \lambda$;
$\kappa :\ = \kappa +\!\!+ \langle f.\sigma.\eta \rangle;$
$\sigma, \sigma' :\ = \sigma +\!\!+ \eta, \sigma' \downharpoonright |\eta|$
od;
return κ
$]\!]$

decode.κ_0 =
{ pre: $\langle \exists \sigma_0 :\ : \kappa_0 =$ **encode**.$\sigma_0 \rangle$ }
$[\![$ **var** $\sigma : \Sigma^*;\ \kappa' : \mathcal{K}^*$
| $\kappa', \sigma :\ = \kappa_0, \lambda;$
do $\kappa' \neq \lambda \rightarrow$
let $\kappa' :: \langle c \rangle +\!\!+ \rho'$;
$\sigma :\ = \sigma +\!\!+ g.\sigma.c;$
$\kappa' :\ = \rho'$
od;
return σ
$]\!]$

4.2.2 Adapted initial root specification

The initial root specification is adapted here to make further specifications easier.

The only statement that is actually adapted is the **let**-statement in the **encode** algorithm of the initial root. It is one statement, but in fact it solves two subproblems.

- The first subproblem is the determination of "proper" η's. All these η's will be determined with the function $h_{pp} : \Sigma^* \times \Sigma^* \rightarrow \mathcal{P}.\Sigma^*$. This function results in a set containing all proper η's. Two arguments are needed to construct the set, this will be σ and σ'.

  At the moment it has already been specified that a proper η is not the empty string and that η is a prefix of σ'. This results is two assumptions for the function.

  - Progress: $\langle \forall \sigma, \sigma' : \sigma' \neq \lambda : \lambda \notin h_{pp}.(\sigma, \sigma') \rangle$
  - IsPrefix: $\langle \forall \sigma, \sigma', \eta : \sigma' \neq \lambda : \eta \in h_{pp}.(\sigma, \sigma') \Rightarrow \eta \preccurlyeq_p \sigma' \rangle$

  The function is called h_{pp} because it will only contain Proper Prefixes of σ'.

- The second subproblem is the actual selecting of a proper η. This will be done with a function $h_{sel} : \mathcal{P}.\Sigma^* \times \Sigma^* \times \Sigma^* \nrightarrow \Sigma^*$. The first argument contains a set with all the proper η's. The next two arguments – which will be σ and σ' – are information on which the selection will be made. The function is partial, because the set for selection can be empty.

  One assumption has to be made for the function. If the set for selection is nonempty, then the function has to result in an element of the set. This is formally stated as ProperSelect : $\langle \forall V, \sigma, \sigma' : \sigma' \neq \lambda \wedge V \in \mathcal{P}.\Sigma^* : V \neq \emptyset \Rightarrow h_{sel}.(V, \sigma, \sigma') \in V \rangle$

An adaption of the **let**-statement will make it possible to address these two subproblems separately.

The adapted specification becomes

Specification 4.5 (Adapted Initial Root)

Declarations:

Input		$\Sigma \in$ Alphabet
Output		$\mathcal{K} \in$ Countable Set
Encode		$\textbf{encode} : \Sigma^* \to \mathcal{K}^*$
Decode		$\textbf{decode} : \mathcal{K}^* \nrightarrow \Sigma^*$
ProperPrefix	+	$h_{pp} : \Sigma^* \times \Sigma^* \to \mathcal{P}.\Sigma^*$
SelectPrefix	+	$h_{sel} : \mathcal{P}.\Sigma^* \times \Sigma^* \times \Sigma^* \nrightarrow \Sigma^*$
Substitute		$f : \Sigma^* \to \Sigma^* \nrightarrow \mathcal{K}$
DeSubstitute		$g : \Sigma^* \to \mathcal{K} \nrightarrow \Sigma^*$

Definitions:

Assumptions:

Lossless		$\textbf{decode} \circ \textbf{encode} = \mathsf{id}$
Progress	+	$\langle \forall \sigma, \sigma' : \sigma' \neq \lambda : \lambda \notin h_{pp}.(\sigma, \sigma') \rangle$
IsPrefix	+	$\langle \forall \sigma, \sigma', \eta : \sigma' \neq \lambda : \eta \in h_{pp}.(\sigma, \sigma') \Rightarrow \eta \preccurlyeq_p \sigma' \rangle$
ProperSelect	+	$\langle \forall V, \sigma, \sigma' : \sigma' \neq \lambda \wedge V \in \mathcal{P}.\Sigma^* : V \neq \emptyset \Rightarrow h_{sel}.(V, \sigma, \sigma') \in V \rangle$

Lemmas:

Lossless	$\textbf{decode} \circ \textbf{encode} = \mathsf{id}$	(4.2)

Algorithms:

$\textbf{encode}.\sigma_0 =$
$[\![$ $\textbf{var}\ \sigma, \sigma' : \Sigma^*;\ \kappa : \mathcal{K}^*$
$|\ \sigma', \kappa, \sigma :\ = \sigma_0, \lambda, \lambda;$
$\textbf{do}\ \sigma' \neq \lambda \to$
$\quad \textbf{let}\ \eta\ \textbf{such that}\ \eta = h_{sel}.(V, \sigma, \sigma')$
$\qquad \textbf{where}\ V = h_{pp}.(\sigma, \sigma')\ ;$
$\quad \kappa :\ = \kappa +\!\!+ \langle f.\sigma.\eta \rangle;$
$\quad \sigma, \sigma' :\ = \sigma +\!\!+ \eta, \sigma' \downharpoonright |\eta|$
$\textbf{od};$
$\textbf{return}\ \kappa$
$]\!]$

$\textbf{decode}.\kappa_0 =$
$\{\ \text{pre: } \langle \exists \sigma_0 : \ : \kappa_0 = \textbf{encode}.\sigma_0 \rangle\ \}$
$[\![$ $\textbf{var}\ \sigma : \Sigma^*;\ \kappa' : \mathcal{K}^*$
$|\ \kappa', \sigma :\ = \kappa_0, \lambda;$
$\textbf{do}\ \kappa' \neq \lambda \to$
$\quad \textbf{let}\ \kappa' :: \langle c \rangle +\!\!+ \rho'\ ;$
$\quad \sigma :\ = \sigma +\!\!+ g.\sigma.c;$
$\quad \kappa' :\ = \rho'$
$\textbf{od};$
$\textbf{return}\ \sigma$
$]\!]$

4.2.3 Final root specification

The final root specification is constructed with the adapted initial root specification. It is called the final root specification, because it is the first specification that does not assume Lossless to conclude Lossless.

The Lossless assumption is proven in Proof 4.7. This proof needs several new assumptions. Two are related to the progress of the encoder, namely the Substitutable and the

ProperPrefixExists assumption. The only other assumption needed for the proof is the SubstituteId assumption.

After the proof there are some remarks. The remarks are about the selection of η (Remark 4.8), the encode algorithm being a total function(Remark 4.9) and the relation between $f.\sigma$ and $g.\sigma$(Remark 4.10).

Specification 4.6 (Final Root)

Declarations:

Input	$\Sigma \in$ Alphabet
Output	$\mathcal{K} \in$ Countable Set
Encode	$\mathbf{encode} : \Sigma^* \to \mathcal{K}^*$
Decode	$\mathbf{decode} : \mathcal{K}^* \not\to \Sigma^*$
ProperPrefix	$h_{pp} : \Sigma^* \times \Sigma^* \to \mathcal{P}.\Sigma^*$
SelectPrefix	$h_{sel} : \mathcal{P}.\Sigma^* \times \Sigma^* \times \Sigma^* \not\to \Sigma^*$
Substitute	$f : \Sigma^* \to \Sigma^* \not\to \mathcal{K}$
DeSubstitute	$g : \Sigma^* \to \mathcal{K} \not\to \Sigma^*$

Definitions:

Assumptions:

SubstituteId	+	$\langle \forall \sigma, \eta : \eta \in dom.(f.\sigma) : (g.\sigma) \circ (f.\sigma) = \mathsf{id} \rangle$
Substitutable	+	$\langle \forall \sigma, \sigma' : \sigma' \neq \lambda : h_{pp}.(\sigma, \sigma') \subseteq dom.(f.\sigma) \rangle$
ProperPrefixExists	+	$\langle \forall \sigma, \sigma' : \sigma' \neq \lambda : h_{pp}.(\sigma, \sigma') \neq \emptyset \rangle$
Progress		$\langle \forall \sigma, \sigma' : \sigma' \neq \lambda : \lambda \notin h_{pp}.(\sigma, \sigma') \rangle$
IsPrefix		$\langle \forall \sigma, \sigma', \eta : \sigma' \neq \lambda : \eta \in h_{pp}.(\sigma, \sigma') \Rightarrow \eta \preccurlyeq_p \sigma' \rangle$
ProperSelect		$\langle \forall V, \sigma, \sigma' : \sigma' \neq \lambda \wedge V \in \mathcal{P}.\Sigma^* : V \neq \emptyset \Rightarrow h_{sel}.(V, \sigma, \sigma') \in V \rangle$

Lemmas:

Lossless	+	$\mathbf{decode} \circ \mathbf{encode} = \mathsf{id}$	(4.7)

Algorithms:

$\mathbf{encode}.\sigma_0 =$
$[\![$ **var** $\sigma, \sigma' : \Sigma^*;\ \kappa : \mathcal{K}^*$
$|\ \sigma', \kappa, \sigma := \sigma_0, \lambda, \lambda;$
do $\sigma' \neq \lambda \to$
let η **such that** $\eta = h_{sel}.(V, \sigma, \sigma')$
where $V = h_{pp}.(\sigma, \sigma')$;
$\kappa := \kappa +\!\!+ \langle f.\sigma.\eta \rangle;$
$\sigma, \sigma' := \sigma +\!\!+ \eta, \sigma' \downharpoonright |\eta|$
od;
return κ
$]\!]$

$\mathbf{decode}.\kappa_0 =$
{ pre: $\langle \exists \sigma_0 : \ : \kappa_0 = \mathbf{encode}.\sigma_0 \rangle$ }
$[\![$ **var** $\sigma : \Sigma^*;\ \kappa' : \mathcal{K}^*$
$|\ \kappa', \sigma := \kappa_0, \lambda;$
do $\kappa' \neq \lambda \to$
let $\kappa' :: \langle c \rangle +\!\!+ \rho'$;
$\sigma := \sigma +\!\!+ g.\sigma.c;$
$\kappa' := \rho'$
od;
return σ
$]\!]$

Proof 4.7 (Lossless)
The Lossless assumption can be proven by interweaving the encoder and decoder. But this proof would require much textual explanation. A much more elegant proof can be given if the imperative algorithms are first transformed into functional versions.

The imperative encode and decode algorithm from Specification 4.6 can be transformed into functional versions with the help of standard transformations [vGvdW02]. This results in

$\mathbf{encode}.\sigma_0 = \mathbf{enc}.\lambda.\sigma_0$
$\mathbf{enc}.\sigma.\lambda = \lambda$
$\mathbf{enc}.\sigma.\sigma' = \langle f.\sigma.\eta\rangle +\!\!+ \mathbf{enc}.(\sigma +\!\!+ \eta).(\sigma' \downharpoonleft |\eta|)$ **where** $\eta = h_{sel}.(h_{pp}.(\sigma, \sigma'), \sigma, \sigma')$

$\mathbf{decode}.\kappa_0 = \mathbf{dec}.\lambda.\kappa_0$
$\mathbf{dec}.\sigma.\lambda = \lambda$
$\mathbf{dec}.\sigma.(\langle c\rangle +\!\!+ \rho') = g.\sigma.c +\!\!+ \mathbf{dec}.(\sigma +\!\!+ g.\sigma.c).\rho'$

To prove the Lossless assumption, $\mathbf{dec}.\lambda.(\mathbf{enc}.\lambda.\sigma_0) = \sigma_0$ has to be proven. It is proven by proving the generalized version, namely $\mathbf{dec}.\sigma.(\mathbf{enc}.\sigma.\sigma_0) = \sigma_0$. The actual proof is heavily based on the work of Van Geldrop and Van der Woude [vGvdW02].

The proof is given on the basis of strong induction.

case $\sigma_0 = \lambda$

$\mathbf{dec}.\sigma.(\mathbf{enc}.\sigma.\lambda)$

$\equiv$ {definition **enc** and **dec**, both first case}

λ

case $\sigma_0 = \sigma_1 \quad (\sigma_1 \neq \lambda)$

Induction Hypothesis(IH): $\left\langle \forall \alpha, \beta : |\alpha| < |\sigma_1| : \mathbf{dec}.\beta.(\mathbf{enc}.\beta.\alpha) = \alpha \right\rangle$

$\mathbf{dec}.\sigma.(\mathbf{enc}.\sigma.\sigma_1)$

$\equiv$ {definition **enc**, second case}

$\mathbf{dec}.\sigma.(\langle f.\sigma.\eta\rangle +\!\!+ \mathbf{enc}.(\sigma +\!\!+ \eta).(\sigma_1 \downharpoonleft |\eta|))$ **where** $\eta = h_{sel}.(h_{pp}.(\sigma, \sigma_1), \sigma, \sigma_1)$

$\equiv$ {ProperPrefixExists and ProperSelect, thus an η exists, $\bullet \eta = h_{sel}.(h_{pp}.(\sigma, \sigma_1), \sigma, \sigma_1) \quad \{\eta \in h_{pp}.(\sigma, \sigma_1)\}$}

$\mathbf{dec}.\sigma.(\langle f.\sigma.\eta\rangle +\!\!+ \mathbf{enc}.(\sigma +\!\!+ \eta).(\sigma_1 \downharpoonleft |\eta|))$

$\equiv$ {definition **dec**, second case}

$(g.\sigma).((f.\sigma).\eta) \mathbin{+\!\!+} \mathbf{dec}.(\sigma \mathbin{+\!\!+} (g.\sigma).((f.\sigma).\eta)).(\mathbf{enc}.(\sigma \mathbin{+\!\!+} \eta).(\sigma_1 \downharpoonleft |\eta|))$

$\equiv$ {Substitutable, SubstituteId}

$\eta \mathbin{+\!\!+} \mathbf{dec}.(\sigma \mathbin{+\!\!+} \eta).(\mathbf{enc}.(\sigma \mathbin{+\!\!+} \eta).(\sigma_1 \downharpoonleft |\eta|))$

$\equiv$ {IH, $|(\sigma_1 \downharpoonleft |\eta|)| < |\sigma_1|$, $\eta \neq \lambda$(Progress)}

$\eta \mathbin{+\!\!+} (\sigma_1 \downharpoonleft |\eta|)$

$\equiv$ {IsPrefix}

σ_1 □

Remark 4.8 (Selection η)
The determination of η – with function $h_{sel}.(V, \sigma, \sigma')$ – is at the moment non-deterministic. There can be many proper η's, but only one will be selected.

The selection of η has effect on the compression ratio. If a string has to be compressed, then an η has to be selected in each iteration. Consequently, it is possible that a string can be compressed in various ways. All these compressed strings can have different lengths. Consequently, the choice to select one particular η can lead to a better compression ratio than the selection of another η. □

Remark 4.9 (ProperPrefixExists, Substitutable and Progress)
The ProperPrefixExists, Substitutable and the Progress assumption reflect that the encode algorithm is a total function, which must be able to process any string. The assumptions guarantee that the algorithm can always proceed and that progress is made. □

Remark 4.10 (SubstituteId assumption)
The SubstituteId assumption is valid if $g.\sigma$ were the inverse of $f.\sigma$. But this is too strict. Beside $(g.\sigma).((f.\sigma).\eta) = \eta$, it would also require $(f.\sigma).((g.\sigma).c) = c$. □

4.3 Specializing the root

This section specializes the root specification in two steps. The first step – located in §4.3.1 – defines h_{pp}. The second step – located in 4.3.2 – introduces the "environment" $\mathcal{E}$, which is closely related to referents and references. Both steps introduce one discriminator that result in one subgroup. It is possible that multiple subgroups could be indicated, but only one will be used here[2].

The specialization steps are visualized in figure 4.2. In this figure the lines of the steps that do not belong to the specialization are dashed.

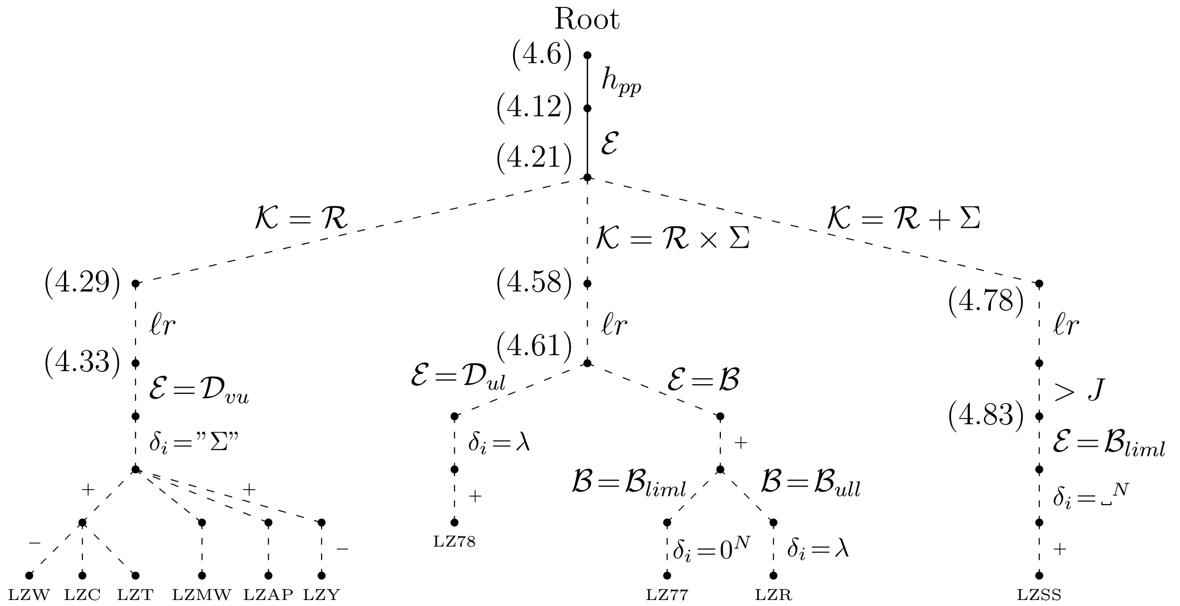


Figure 4.2: Specialization steps in the visual summary of the taxonomy

4.3.1 Defining the ProperPrefix function h_{pp}

In this subsection the definition for the ProperPrefix function h_{pp} is given. This definition can already be given because the assumptions given so far guarantee that each element of $h_{pp}.(\sigma, \sigma')$ is a proper prefix. That is to say, whichever element of $h_{pp}.(\sigma, \sigma')$ is chosen, progress is always made and it is always a prefix of σ'.

The definition of h_{pp} is given 4.11. With this definition, the proofs for assumption Substitutable(4.13), Progress(4.15) and IsPrefix(4.16) can be given. The ProperPrefixExists assumption is proven in 4.14. This proof needs the additional assumption MinDom$f$ assumption.

After the proofs there is a remark about the MinDom$f$ assumption(Remark 4.17).

[2]If multiple subgroups would be indicated, then the taxonomy would become a multi-dimensional taxonomy, which makes everything more difficult.

Discriminator 4.11 (Definition of $h_{pp}.(\sigma, \sigma')$)
There are four assumptions for $h_{pp}.(\sigma, \sigma')$. Namely Substitutable, ProperPrefixExists, Progress and IsPrefix.

Only the Substitutable, Progress and IsPrefix assumption are used for the definition of $h_{pp}.(\sigma, \sigma')$. These assumptions assume something about each element of $h_{pp}.(\sigma, \sigma')$, which can be used for the definition. The ProperPrefixExists assumption does not assume something about each element of $h_{pp}.(\sigma, \sigma')$, it assumes something for the whole set.

The definition is: $h_{pp}.(\sigma, \sigma') = \{\eta | \eta \in dom.(f.\sigma) \wedge \eta \preccurlyeq_p \sigma' \wedge \eta \neq \lambda\}$. □

Specification 4.12 (Root with h_{pp})

Declarations:

Input		$\Sigma \in$ Alphabet
Output		$\mathcal{K} \in$ Countable Set
Encode		$\mathbf{encode} : \Sigma^* \rightarrow \mathcal{K}^*$
Decode		$\mathbf{decode} : \mathcal{K}^* \nrightarrow \Sigma^*$
ProperPrefix		$h_{pp} : \Sigma^* \times \Sigma^* \rightarrow \mathcal{P}.\Sigma^*$
SelectPrefix		$h_{sel} : \mathcal{P}.\Sigma^* \times \Sigma^* \times \Sigma^* \nrightarrow \Sigma^*$
Substitute		$f : \Sigma^* \rightarrow \Sigma^* \nrightarrow \mathcal{K}$
DeSubstitute		$g : \Sigma^* \rightarrow \mathcal{K} \nrightarrow \Sigma^*$

Definitions:

	+	$h_{pp}.(\sigma, \sigma') = \{\eta | \eta \in dom.(f.\sigma) \wedge \eta \preccurlyeq_p \sigma' \wedge \eta \neq \lambda\}$

Assumptions:

SubstituteId		$\langle \forall \sigma, \eta : \eta \in dom.(f.\sigma) : (g.\sigma) \circ (f.\sigma) = \mathsf{id} \rangle$
ProperSelect		$\langle \forall V, \sigma, \sigma' : \sigma' \neq \lambda \wedge V \in \mathcal{P}.\Sigma^* : V \neq \emptyset \Rightarrow h_{sel}.(V, \sigma, \sigma') \in V \rangle$
MinDom$f$	+	$\langle \forall \sigma : \ : \Sigma \subseteq dom.(f.\sigma) \rangle$

Lemmas:

Lossless		$\mathbf{decode} \circ \mathbf{encode} = \mathsf{id}$	(4.7)
Substitutable	+	$\langle \forall \sigma, \sigma' : \sigma' \neq \lambda : h_{pp}.(\sigma, \sigma') \subseteq dom.(f.\sigma) \rangle$	(4.13)
ProperPrefixExists	+	$\langle \forall \sigma, \sigma' : \sigma' \neq \lambda : h_{pp}.(\sigma, \sigma') \neq \emptyset \rangle$	(4.14)
Progress	+	$\langle \forall \sigma, \sigma' : \sigma' \neq \lambda : \lambda \notin h_{pp}.(\sigma, \sigma') \rangle$	(4.15)
IsPrefix	+	$\langle \forall \sigma, \sigma', \eta : \sigma' \neq \lambda : \eta \in h_{pp}.(\sigma, \sigma') \Rightarrow \eta \preccurlyeq_p \sigma' \rangle$	(4.16)

Algorithms:

encode.σ_0 =
  $[\![$ **var** $\sigma, \sigma' : \Sigma^*;\ \kappa : \mathcal{K}^*$
  $\mid \sigma', \kappa, \sigma :\ = \sigma_0, \lambda, \lambda;$
    do $\sigma' \neq \lambda \rightarrow$
      let η **such that** $\eta = h_{sel}.(V, \sigma, \sigma')$
        where $V = h_{pp}.(\sigma, \sigma')$;
      $\kappa :\ = \kappa +\!\!+ \langle f.\sigma.\eta \rangle;$
      $\sigma, \sigma' :\ = \sigma +\!\!+ \eta, \sigma' \downharpoonright |\eta|$
    od;
    return κ
  $]\!]$

decode.κ_0 =
{ pre: $\left\langle \exists \sigma_0 :\ : \kappa_0 = \textbf{encode}.\sigma_0 \right\rangle$ }
  $[\![$ **var** $\sigma : \Sigma^*;\ \kappa' : \mathcal{K}^*$
  $\mid \kappa', \sigma :\ = \kappa_0, \lambda;$
    do $\kappa' \neq \lambda \rightarrow$
      let $\kappa' :: \langle c \rangle +\!\!+ \rho'$;
      $\sigma :\ = \sigma +\!\!+ g.\sigma.c;$
      $\kappa' :\ = \rho'$
    od;
    return σ
  $]\!]$

Proof 4.13 (Substitutable)
For all σ, nonempty σ' and η it holds that

$\eta \in h_{pp}.(\sigma, \sigma')$

$\Rightarrow$ $\{$definition $h_{pp}.(\sigma, \sigma')\}$

$\eta \in dom.(f.\sigma)$

Consequently, $h_{pp}.(\sigma, \sigma') \subseteq dom.(f.\sigma)$. □

Proof 4.14 (ProperPrefixExists)
For all σ and nonempty σ' it holds that

$h_{pp}.(\sigma, \sigma')$

$\equiv$ $\{$definition $h_{pp}.(\sigma, \sigma')\}$

$\{\eta | \eta \in dom.(f.\sigma) \wedge \eta \preccurlyeq_p \sigma' \wedge \eta \neq \lambda\}$

$\supseteq$ $\{\sigma' \neq \lambda$, MinDom$f\}$

$\{\sigma' \upharpoonright 1\}$

Consequently, $h_{pp}.(\sigma, \sigma')$ is not the empty set. □

Proof 4.15 (Progress)
For all σ, nonempty σ' and η it holds that

$\eta \in h_{pp}.(\sigma, \sigma')$

$\Rightarrow$ $\{$definition $h_{pp}.(\sigma, \sigma')\}$

$\eta \neq \lambda$

Consequently, $\lambda \notin h_{pp}.(\sigma, \sigma')$. □

Proof 4.16 (IsPrefix)
For all σ, nonempty σ' and η it holds that

$\eta \in h_{pp}.(\sigma, \sigma')$

$\Rightarrow$ $\{$definition $h_{pp}.(\sigma, \sigma')\}$

$\eta \preccurlyeq_p \sigma'$ □

Remark 4.17 (MinDom$f$)
The MinDom$f$ assumption is not just an assumption. It is precisely the minimum domain that is needed to prove that a proper prefix exists. If one element of Σ is left out, then it is possible that no proper prefix exists.

Take for example a σ' of length one. Because σ' can be any symbol, Σ has to be a subset of $dom.(f.\sigma)$, otherwise there is no proper prefix. But the domain of $f.\sigma$ is independent of σ'. Consequently, Σ always has to be a subset of $dom.(f.\sigma)$. □

4.3.2 Introducing environment $\mathcal{E}$

This subsection introduces the abstract data type environment $\mathcal{E}$. The environment is a special method to implement $f_{\mathcal{R}}$ and $g_{\mathcal{R}}$. These functions will be clarified first in §4.3.2.1. How these functions relate to the environment is explained secondly. As third, the environment will be defined. As fourth, the environment will be introduced to the specification.

4.3.2.1 $f_{\mathcal{R}}$ and $g_{\mathcal{R}}$

The functions $f_{\mathcal{R}}$ and $g_{\mathcal{R}}$ are two general functions used to handle referents and references. They are the core for textual substitution and will be used in all further specifications. The functions have the signatures

- $f_{\mathcal{R}} : \Sigma^* \rightarrow \Sigma^* \nrightarrow \mathcal{R}$ (Refer)
- $g_{\mathcal{R}} : \Sigma^* \rightarrow \mathcal{R} \nrightarrow \Sigma^*$ (DeRefer)

Both functions have as a first argument the string where the referents and references are based on. For both the encoder and decoder this is – not coincidentally – σ.

Because it has to be possible to de-refer a reference –that is the whole idea behind textual substitution– , it is required that $\langle \forall \sigma, \eta : \eta \in dom.(f_{\mathcal{R}}.\sigma) : (g_{\mathcal{R}}.\sigma) \circ (f_{\mathcal{R}}.\sigma) = \mathsf{id} \rangle$.

4.3.2.2 $f_{\mathcal{R}}$ and $g_{\mathcal{R}}$ related to environment $\mathcal{E}$

The function $f_{\mathcal{R}}$ and $g_{\mathcal{R}}$ are related to the abstract data type environment $\mathcal{E}$ by two loop-invariants. Let δ be an algorithm variable with type $\mathcal{E}$ and let $\bigtriangledown$ and $\bigtriangleup$ be operators of $\mathcal{E}$. The loop-invariants are then

- $f_{\mathcal{R}}.\sigma = \delta\bigtriangledown \qquad (\delta\bigtriangledown : \Sigma^* \nrightarrow \mathcal{R})$
- $g_{\mathcal{R}}.\sigma = \delta\bigtriangleup \qquad (\delta\bigtriangleup : \mathcal{R} \nrightarrow \Sigma^*)$

The functions $\delta\bigtriangledown$ and $\delta\bigtriangleup$ are the *"on the fly"* versions of $f_{\mathcal{R}}.\sigma$ and $g_{\mathcal{R}}.\sigma$. On the fly construction is a well known method to maintain a function with a variable. It introduces a variable to replace a function. If the value of an argument of the function changes, then the value of the variable has to be changed too.

Example 4.18 (On the fly construction)
Let $\mathbf{Npwr}.i = N^i$. The following algorithm fragment uses on the fly construction for $\mathbf{Npwr}.i$.

$i, j :=0, 1;$
do $j < M \rightarrow \{ j = \mathbf{Npwr}.i \}$
  $i, j := i + 1, j * N$
od

The value of $\mathbf{Npwr}.i$ and j are related by the loop-invariant $j = \mathbf{Npwr}.i$. □

4.3.2.3 Defining environment $\mathcal{E}$

Before the definition of the abstract data type environment is given, its operators are introduced stepwise. It is used that $\delta \in \mathcal{E}$.

- operator $\varnothing$
  This operator gives an "empty" environment.
- operator $\oplus$
  $\delta \oplus \alpha$ alters the functions $\delta\bigtriangledown$ and $\delta\bigtriangleup$. Because $\delta\bigtriangledown$ and $\delta\bigtriangleup$ are the on the fly versions of $f_{\mathcal{R}}.\sigma$ and $g_{\mathcal{R}}.\sigma$, δ has to be altered each time that σ is altered.
- operators $\bigtriangledown$[3]and $\bigtriangleup$[3]
  The functions $\delta\bigtriangledown$ and $\delta\bigtriangleup$ are the on the fly versions of $f_{\mathcal{R}}.\sigma$ and $g_{\mathcal{R}}.\sigma$.
- operator $\| \ \|$
  The environment has some notion of size. This operator is added to give that size.

Definition 4.19 (Abstract data type environment $\mathcal{E}$)
The abstract data type environment $\mathcal{E}$ is defines as:

[3] $\bigtriangledown$ and $\bigtriangleup$ can be "read" top down. $\bigtriangledown$ is wide at the top and narrow at the bottom. It gives a smaller representation. $\bigtriangleup$ is narrow at the top and wide at the bottom. It gives a larger representation.

declaration	meaning	pronunciation
$\varnothing \; : \; \mathcal{E}$	empty environment	empty
$\oplus \; : \; \mathcal{E} \times \Sigma^* \rightarrow \mathcal{E}$	alter environment	plussel
$\bigtriangledown \; : \; \mathcal{E} \rightarrow \Sigma^* \nrightarrow \mathcal{R}$	get reference for referent	ref
$\bigtriangleup \; : \; \mathcal{E} \rightarrow \mathcal{R} \nrightarrow \Sigma^*$	get referent for reference	de-ref
$\\| \; \\| \; : \; \mathcal{E} \rightarrow \mathbb{N}$	size of environment	size

With the assumption that

- SubstituteId$\mathcal{E}$: $\langle \forall \delta, \eta \; : \; \eta \in dom.(\delta \bigtriangledown) \; : \; (\delta \bigtriangleup) \circ (\delta \bigtriangledown) = \mathsf{id} \rangle$ □

Section 4.5 gives several implementations for the environment.

4.3.2.4 Using environment $\mathcal{E}$

The environment is embedded in the specification by Discriminator 4.20. The new specification is given thereafter. The specification is followed by a proof for the SubstituteId$\mathcal{R}$ assumption(4.22). After the proof there are some remarks. They are about the justification for the introduction of $\mathcal{E}$(4.23), the new function e(4.24) and the correctness of the loop-invariants in the algorithms in relation to $f_{\mathcal{R}}$ and $g_{\mathcal{R}}$(4.25).

Discriminator 4.20 (Environment $\mathcal{E}$)
The abstract data type $\mathcal{E}$ is used to to implement $f_{\mathcal{R}}.\sigma$ and $g_{\mathcal{R}}.\sigma$. The two functions are needed to handle referents and references. □

Specification 4.21 (Root with h_{pp}, $\mathcal{E}$)

Declarations:

Input		$\Sigma \in$ Alphabet
Output		$\mathcal{K} \in$ Countable Set
Reference	+	$\mathcal{R} \in$ Countable Set
Initial $\mathcal{E}$	+	$\delta_{init} \in \mathcal{E}$
Encode		$\mathbf{encode} : \Sigma^* \rightarrow \mathcal{K}^*$
Decode		$\mathbf{decode} : \mathcal{K}^* \nrightarrow \Sigma^*$
ProperPrefix		$h_{pp} : \Sigma^* \times \Sigma^* \rightarrow \mathcal{P}.\Sigma^*$
SelectPrefix		$h_{sel} : \mathcal{P}.\Sigma^* \times \Sigma^* \times \Sigma^* \nrightarrow \Sigma^*$
Substitute		$f : \Sigma^* \rightarrow \Sigma^* \nrightarrow \mathcal{K}$
Refer	+	$f_{\mathcal{R}} : \Sigma^* \rightarrow \Sigma^* \nrightarrow \mathcal{R}$
DeSubstitute		$g : \Sigma^* \rightarrow \mathcal{K} \nrightarrow \Sigma^*$
DeRefer	+	$g_{\mathcal{R}} : \Sigma^* \rightarrow \mathcal{R} \nrightarrow \Sigma^*$
AdaptEnvironment	+	$e : \mathcal{E} \times \Sigma^* \rightarrow \mathcal{E}$

Definitions:

$$h_{pp}.(\sigma, \sigma') = \{\eta | \eta \in dom.(f.\sigma) \wedge \eta \preccurlyeq_p \sigma' \wedge \eta \neq \lambda\}$$

Assumptions:

SubstituteId		$\langle \forall \sigma, \eta : \eta \in dom.(f.\sigma) : (g.\sigma) \circ (f.\sigma) = \mathsf{id} \rangle$	
ProperSelect		$\langle \forall V, \sigma, \sigma' : \sigma' \neq \lambda \wedge V \in \mathcal{P}.\Sigma^* : V \neq \emptyset \Rightarrow h_{sel}.(V, \sigma, \sigma') \in V \rangle$	
MinDom$f$		$\langle \forall \sigma : : \Sigma \subseteq dom.(f.\sigma) \rangle$	
SubstituteId$\mathcal{E}$	+	$\langle \forall \delta, \eta : \eta \in dom.(\delta \triangledown) : (\delta \triangle) \circ (\delta \triangledown) = \mathsf{id} \rangle$	

Lemmas:

Lossless		**decode** $\circ$ **encode** $= \mathsf{id}$	(4.7)
Substitutable		$\langle \forall \sigma, \sigma' : \sigma' \neq \lambda : h_{pp}.(\sigma, \sigma') \subseteq dom.(f.\sigma) \rangle$	(4.13)
ProperPrefixExists		$\langle \forall \sigma, \sigma' : \sigma' \neq \lambda : h_{pp}.(\sigma, \sigma') \neq \emptyset \rangle$	(4.14)
Progress		$\langle \forall \sigma, \sigma' : \sigma' \neq \lambda : \lambda \notin h_{pp}.(\sigma, \sigma') \rangle$	(4.15)
IsPrefix		$\langle \forall \sigma, \sigma', \eta : \sigma' \neq \lambda : \eta \in h_{pp}.(\sigma, \sigma') \Rightarrow \eta \preccurlyeq_p \sigma' \rangle$	(4.16)
SubstituteId$\mathcal{R}$	+	$\langle \forall \sigma, \eta : \eta \in dom.(f_{\mathcal{R}}.\sigma) : (g_{\mathcal{R}}.\sigma) \circ (f_{\mathcal{R}}.\sigma) = \mathsf{id} \rangle$	(4.22)

Algorithms:

```
encode.σ0 =
  |[ var σ, σ' : Σ*; κ : K*; δ : E
  | σ', κ, σ, δ : = σ0, λ, λ, δ_init;
    do σ' ≠ λ → { δ▽ = f_R.σ }
        let η such that η = h_sel.(V, σ, σ')
                where V = h_pp.(σ, σ') ;
        κ : = κ ++ ⟨f.σ.η⟩;
        σ, σ', δ : = σ ++ η, σ' ↓ |η|, e.(δ, η)
    od;
    return κ
  ]|
```

```
decode.κ0 =
{ pre: ⟨∃σ0 : : κ0 = encode.σ0⟩ }
  |[ var σ : Σ*; κ' : K*; δ : E
  | κ', σ, δ : = κ0, λ, δ_init;
    do κ' ≠ λ → { δ△ = g_R.σ }
        let κ' :: ⟨c⟩ ++ ρ' ;
        σ, δ : = σ ++ g.σ.c, e.(δ, g.σ.c);
        κ' : = ρ'
    od;
    return σ
  ]|
```

Proof 4.22 (SubstituteId$\mathcal{R}$)
For all σ and $\eta \in dom.(f_{\mathcal{R}}.\sigma)$ it holds that

$$(g_{\mathcal{R}}.\sigma) \circ (f_{\mathcal{R}}.\sigma) = \mathsf{id}$$

$\equiv$ {loop-invariants $\delta \triangledown = f_{\mathcal{R}}.\sigma$ and $\delta \triangle = g_{\mathcal{R}}.\sigma$, SubstituteId$\mathcal{E}$}

$$\mathsf{true}$$

□

Remark 4.23 (Justification for introduction of $\mathcal{E}$)
Although there is no actual need to introduce $\mathcal{E}$ for handling referents and references, it is is justified to so. There is no actual need because the role of the referents and references is not yet exactly known. But because it is known that referents and references will be used, it is justified to introduce $\mathcal{E}$ to handle them. If referents and references are not used, then no compression can be achieved. □

Remark 4.24 (Function e)
The function e is introduced for the adaption of the environment. It uses the old environment and the prefix that is currently being processed, which is η. The adaption is limited by the loop-invariants in the algorithm, see also Remark 4.25. □

Remark 4.25 (Correctness of the loop-invariants)
The loop-invariants given in the algorithms are correct. Because the functions $f_{\mathcal{R}}$ and $g_{\mathcal{R}}$ have not yet been defined, it is not logic to state that the proof is correct. But there is a reason why they are stated to be correct.

The functions $f_{\mathcal{R}}$ and $g_{\mathcal{R}}$ will actually never be defined. Only the initialization and adaption of the on the fly variable δ will be defined. The definition of the functions $f_{\mathcal{R}}$ and $g_{\mathcal{R}}$ could probably be derived from the on the fly variable, but this will not be done, the function is only used in the annotation of the algorithm.

The functions are maintained because they can be easily used outside the algorithms. δ is bounded to the algorithm and can not be used outside the algorithm.

Consequently, the on the fly variable δ is not limited by the definition of $f_{\mathcal{R}}$ and $g_{\mathcal{R}}$. It is only limited by the assumptions for the functions $f_{\mathcal{R}}$ and $g_{\mathcal{R}}$. As a consequence, the adaption of δ by means of function e is quite freely to define. □

4.4 Dividing the specialized root in subgroups

In this section the specialized root specification is divided in subgroups on basis of the embedding of $\mathcal{R}$ – the type for references – in $\mathcal{K}$.

Theoretically, it is possible that $\mathcal{R}$ is not embedded in $\mathcal{K}$. But if that is the case, then no references can be used and no compression can be achieved. For practical solutions, $\mathcal{R}$ has to be embedded in $\mathcal{K}$.

There are in general two methods for embedding $\mathcal{R}$ in $\mathcal{K}$.

- Only embed $\mathcal{R}$ in $\mathcal{K}$, which results in $\mathcal{K} = \mathcal{R}$
- Beside $\mathcal{R}$, embed also other types in $\mathcal{K}$. It is here chosen to use the additional embedment of Σ (see Remark 4.26). Two possibilities to combine $\mathcal{R}$ and Σ are
  - $\mathcal{K} = \mathcal{R} \times \Sigma$, this will result in a strict alternation of $\mathcal{R}$ and Σ in $\mathcal{K}^*$.
  - $\mathcal{K} = \mathcal{R} + \Sigma$, this will result in a free alternation of $\mathcal{R}$ and Σ in $\mathcal{K}^*$.

Three possibilities for embedding $\mathcal{R}$ in $\mathcal{K}$ are explicitly given above. These three possibilities form three subgroups, which are the subject of the next three subsections.

Remark 4.26 (Choosing Σ for alternation)
It is a quite reasonable choice to use Σ for alternation. It gives a prospective to satisfy the MinDom$f$ assumption ($\langle\forall\sigma\ :\ \ :\ \Sigma \subseteq dom.(f.\sigma)\rangle$). It makes it possible to directly "project" a single symbol of the input on a (part of a) symbol of the output. □

Remark 4.27 (Other possibilities for $\mathcal{K}$)
There are many other possibilities for $\mathcal{K}$. In general, any combination of types is possible. Only a few are included here, namely those that lead to existing Lempel-Ziv algorithms. □

4.4.1 Subgroup $\mathcal{K} = \mathcal{R}$

This subgroup is one of the major subgroups in this taxonomy. It is indicated in figure 4.3. In this figure the lines of the other subgroups are dashed.

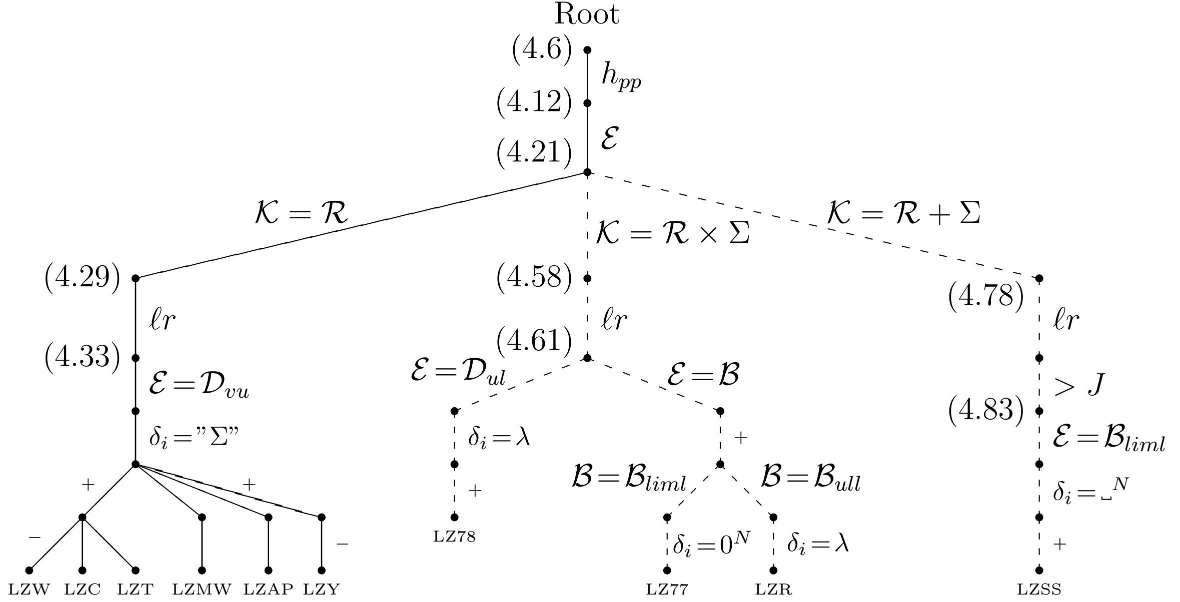


Figure 4.3: Subgroup $\mathcal{K} = \mathcal{R}$ in the visual summary of the taxonomy

Several discriminators are added in this subgroup. Firstly, the main discriminator of this subgroup – $\mathcal{K} = \mathcal{R}$ – is introduced. Secondly, the discriminator ℓr is introduced. This selects precisely one η. Thirdly, a dictionary is chosen for the environment and the dictionary is initialized. Fourthly, different kinds for altering the dictionary are introduced.

4.4.1.1 Introducing $\mathcal{K} = \mathcal{R}$

This subsubsection introduces $\mathcal{K} = \mathcal{R}$ by means of discriminator 4.28. Thereafter a new specification is given. With this discriminator the proof for assumption SubstituteId(4.30) can be given.

Discriminator 4.28 ($\mathcal{K} = \mathcal{R}$)
The embedment of $\mathcal{R}$ in $\mathcal{K}$ is chosen as $\mathcal{K} = \mathcal{R}$. The definition of the functions f and g is related to this embedment. They are defined as $f = f_{\mathcal{R}}$ and $g = g_{\mathcal{R}}$. □

Specification 4.29 (Root with h_{pp}, $\mathcal{E}$, $\mathcal{K} = \mathcal{R}$)

Declarations:

Input	$\Sigma \in$ Alphabet
Output	$\mathcal{K} \in$ Countable Set
Reference	$\mathcal{R} \in$ Countable Set
Initial $\mathcal{E}$	$\delta_{init} \in \mathcal{E}$
Encode	$\mathbf{encode} : \Sigma^* \to \mathcal{K}^*$
Decode	$\mathbf{decode} : \mathcal{K}^* \nrightarrow \Sigma^*$
ProperPrefix	$h_{pp} : \Sigma^* \times \Sigma^* \to \mathcal{P}.\Sigma^*$
SelectPrefix	$h_{sel} : \mathcal{P}.\Sigma^* \times \Sigma^* \times \Sigma^* \nrightarrow \Sigma^*$
Substitute	$f : \Sigma^* \to \Sigma^* \nrightarrow \mathcal{K}$
Refer	$f_{\mathcal{R}} : \Sigma^* \to \Sigma^* \nrightarrow \mathcal{R}$
DeSubstitute	$g : \Sigma^* \to \mathcal{K} \nrightarrow \Sigma^*$
DeRefer	$g_{\mathcal{R}} : \Sigma^* \to \mathcal{R} \nrightarrow \Sigma^*$
AdaptEnvironment	$e : \mathcal{E} \times \Sigma^* \to \mathcal{E}$

Definitions:

$+ \quad \mathcal{K} = \mathcal{R}$

$+ \quad f = f_{\mathcal{R}}$

$+ \quad g = g_{\mathcal{R}}$

$h_{pp}.(\sigma, \sigma') = \{\eta | \eta \in dom.(f.\sigma) \wedge \eta \preccurlyeq_p \sigma' \wedge \eta \neq \lambda\}$

Assumptions:

ProperSelect	$\langle \forall V, \sigma, \sigma' : \sigma' \neq \lambda \wedge V \in \mathcal{P}.\Sigma^* : V \neq \emptyset \Rightarrow h_{sel}.(V, \sigma, \sigma') \in V \rangle$
MinDom$f$	$\langle \forall \sigma : \ : \Sigma \subseteq dom.(f.\sigma) \rangle$
SubstituteId$\mathcal{E}$	$\langle \forall \delta, \eta : \eta \in dom.(\delta \bigtriangledown) : (\delta \bigtriangleup) \circ (\delta \bigtriangledown) = \mathsf{id} \rangle$

Lemmas:

Lossless		$\mathbf{decode} \circ \mathbf{encode} = \mathsf{id}$	(4.7)
Substitutable		$\langle \forall \sigma, \sigma' : \sigma' \neq \lambda : h_{pp}.(\sigma, \sigma') \subseteq dom.(f.\sigma) \rangle$	(4.13)
ProperPrefixExists		$\langle \forall \sigma, \sigma' : \sigma' \neq \lambda : h_{pp}.(\sigma, \sigma') \neq \emptyset \rangle$	(4.14)
Progress		$\langle \forall \sigma, \sigma' : \sigma' \neq \lambda : \lambda \notin h_{pp}.(\sigma, \sigma') \rangle$	(4.15)
IsPrefix		$\langle \forall \sigma, \sigma', \eta : \sigma' \neq \lambda : \eta \in h_{pp}.(\sigma, \sigma') \Rightarrow \eta \preccurlyeq_p \sigma' \rangle$	(4.16)
SubstituteId$\mathcal{R}$		$\langle \forall \sigma, \eta : \eta \in dom.(f_{\mathcal{R}}.\sigma) : (g_{\mathcal{R}}.\sigma) \circ (f_{\mathcal{R}}.\sigma) = \mathsf{id} \rangle$	(4.22)
SubstituteId	+	$\langle \forall \sigma, \eta : \eta \in dom.(f.\sigma) : (g.\sigma) \circ (f.\sigma) = \mathsf{id} \rangle$	(4.30)

Algorithms:

encode.σ_0 =
  $[\![$ **var** $\sigma, \sigma' : \Sigma^*;\ \kappa : \mathcal{K}^*;\ \delta : \mathcal{E}$
  $|\ \sigma', \kappa, \sigma, \delta :\ = \sigma_0, \lambda, \lambda, \delta_{init};$
   do $\sigma' \neq \lambda \rightarrow \{\ \delta\nabla = f_{\mathcal{R}}.\sigma\ \}$
     let η **such that** $\eta = h_{sel}.(V, \sigma, \sigma')$
       where $V = h_{pp}.(\sigma, \sigma')$;
     $\kappa :\ = \kappa +\!\!+ \langle f.\sigma.\eta \rangle;$
     $\sigma, \sigma', \delta :\ = \sigma +\!\!+ \eta, \sigma' \downharpoonleft |\eta|, e.(\delta, \eta)$
   od;
   return κ
  $]\!]$

decode.κ_0 =
$\{$ pre: $\langle \exists \sigma_0 :\ : \kappa_0 =$ **encode**.$\sigma_0 \rangle$ $\}$
  $[\![$ **var** $\sigma : \Sigma^*;\ \kappa' : \mathcal{K}^*;\ \delta : \mathcal{E}$
  $|\ \kappa', \sigma, \delta :\ = \kappa_0, \lambda, \delta_{init};$
   do $\kappa' \neq \lambda \rightarrow \{\ \delta\triangle = g_{\mathcal{R}}.\sigma\ \}$
     let $\kappa' :: \langle c \rangle +\!\!+ \rho'$;
     $\sigma, \delta :\ = \sigma +\!\!+ g.\sigma.c, e.(\delta, g.\sigma.c);$
     $\kappa' :\ = \rho'$
   od;
   return σ
  $]\!]$

Proof 4.30 (SubstituteId)
For all σ and $\eta \in dom.(f.\sigma)$ it holds that

$(g.\sigma) \circ (f.\sigma) = \mathsf{id}$

$\equiv$ {definition $f.\sigma$ and $g.\sigma$}

$(g_{\mathcal{R}}.\sigma) \circ (f_{\mathcal{R}}.\sigma) = \mathsf{id}$

$\equiv$ {SubstituteId$\mathcal{R}$, $\eta \in dom.(f_{\mathcal{R}}.\sigma)$}

true □

4.4.1.2 Selecting η (ℓr)

This subsubsection makes the selection of η deterministic, by introducing the LongestReferent (ℓr) assumption in Discriminator 4.31. The definition for $h_{sel}.(V, \sigma, \sigma')$(4.32) is also given in this step. With this definition the proof for assumption ProperSelect(4.35) can be given.

After the proofs there are some remarks. The remarks are about the selection of η(4.36) and the LongestReferent assumption(4.37).

Discriminator 4.31 (LongestReferent (ℓr))
If $h_{pp}.(\sigma, \sigma')$ has more than one element, then one of the longest is selected. This is introduced in the specification as the LongestReferent assumption. It is formally given by: $h_{sel}.(V, \sigma, \sigma') = \eta \Rightarrow \langle \forall \alpha : \alpha \in V : |\eta| \geq |\alpha| \rangle$ □

Discriminator 4.32 (Definition of $h_{sel}.(V, \sigma, \sigma')$)
There are two assumptions for $h_{sel}.(V, \sigma, \sigma')$. Namely ProperSelect and the new LongestReferent. Because $h_{sel}.(V, \sigma, \sigma')$ is always used in combination with $h_{pp}.(\sigma, \sigma')$ – which does not contains two strings of the same length –, these assumptions select precisely one element

from V. More assumptions will not change this selection. Consequently, the definition can be given now.

The definition is: $h_{sel}.(V,\sigma,\sigma') = \eta$ **such that** $\eta \in V \wedge \langle \forall \alpha : \alpha \in V : |\eta| \geq |\alpha| \rangle$. □

Specification 4.33 (Root with h_{pp}, $\mathcal{E}$, $\mathcal{K} = \mathcal{R}$, ℓr)

Declarations:

Input	$\Sigma \in$ Alphabet
Output	$\mathcal{K} \in$ Countable Set
Reference	$\mathcal{R} \in$ Countable Set
Initial $\mathcal{E}$	$\delta_{init} \in \mathcal{E}$
Encode	$\mathbf{encode} : \Sigma^* \to \mathcal{K}^*$
Decode	$\mathbf{decode} : \mathcal{K}^* \not\to \Sigma^*$
ProperPrefix	$h_{pp} : \Sigma^* \times \Sigma^* \to \mathcal{P}.\Sigma^*$
SelectPrefix	$h_{sel} : \mathcal{P}.\Sigma^* \times \Sigma^* \times \Sigma^* \not\to \Sigma^*$
Substitute	$f : \Sigma^* \to \Sigma^* \not\to \mathcal{K}$
Refer	$f_{\mathcal{R}} : \Sigma^* \to \Sigma^* \not\to \mathcal{R}$
DeSubstitute	$g : \Sigma^* \to \mathcal{K} \not\to \Sigma^*$
DeRefer	$g_{\mathcal{R}} : \Sigma^* \to \mathcal{R} \not\to \Sigma^*$
AdaptEnvironment	$e : \mathcal{E} \times \Sigma^* \to \mathcal{E}$

Definitions:

$\mathcal{K} = \mathcal{R}$

$f = f_{\mathcal{R}}$

$g = g_{\mathcal{R}}$

$h_{pp}.(\sigma,\sigma') = \{\eta | \eta \in dom.(f.\sigma) \wedge \eta \preccurlyeq_p \sigma' \wedge \eta \neq \lambda\}$

\+ $h_{sel}.(V,\sigma,\sigma') = \eta$ **such that** $\eta \in V \wedge \langle \forall \alpha : \alpha \in V : |\eta| \geq |\alpha| \rangle$

Assumptions:

MinDom$f$	$\langle \forall \sigma : \, : \Sigma \subseteq dom.(f.\sigma) \rangle$
SubstituteId$\mathcal{E}$	$\langle \forall \delta, \eta : \eta \in dom.(\delta \triangledown) : (\delta \triangle) \circ (\delta \triangledown) = \mathsf{id} \rangle$

Lemmas:

Lossless		$\mathbf{decode} \circ \mathbf{encode} = \mathsf{id}$	(4.7)
Substitutable		$\langle \forall \sigma, \sigma' : \sigma' \neq \lambda : h_{pp}.(\sigma,\sigma') \subseteq dom.(f.\sigma) \rangle$	(4.13)
ProperPrefixExists		$\langle \forall \sigma, \sigma' : \sigma' \neq \lambda : h_{pp}.(\sigma,\sigma') \neq \emptyset \rangle$	(4.14)
Progress		$\langle \forall \sigma, \sigma' : \sigma' \neq \lambda : \lambda \notin h_{pp}.(\sigma,\sigma') \rangle$	(4.15)
IsPrefix		$\langle \forall \sigma, \sigma', \eta : \sigma' \neq \lambda : \eta \in h_{pp}.(\sigma,\sigma') \Rightarrow \eta \preccurlyeq_p \sigma' \rangle$	(4.16)
SubstituteId$\mathcal{R}$		$\langle \forall \sigma, \eta : \eta \in dom.(f_{\mathcal{R}}.\sigma) : (g_{\mathcal{R}}.\sigma) \circ (f_{\mathcal{R}}.\sigma) = \mathsf{id} \rangle$	(4.22)
SubstituteId		$\langle \forall \sigma, \eta : \eta \in dom.(f.\sigma) : (g.\sigma) \circ (f.\sigma) = \mathsf{id} \rangle$	(4.30)
LongestReferent	+	$\langle \forall V, \sigma, \sigma', \eta : \sigma' \neq \lambda \wedge V \in \mathcal{P}.\Sigma^* :$ $h_{sel}.(V.\sigma,\sigma') = \eta \Rightarrow \langle \forall \alpha : \alpha \in V : |\eta| \geq |\alpha| \rangle \rangle$	(4.34)
ProperSelect	+	$\langle \forall V, \sigma, \sigma' : \sigma' \neq \lambda \wedge V \in \mathcal{P}.\Sigma^* :$ $V \neq \emptyset \Rightarrow h_{sel}.(V,\sigma,\sigma') \in V \rangle$	(4.35)

Algorithms:

```
encode.σ0 =
  |[ var σ, σ' : Σ*; κ : K*; δ : E
  | σ', κ, σ, δ : = σ0, λ, λ, δinit;
    do σ' ≠ λ → { δ▽ = fR.σ }
        let η such that η = hsel.(V, σ, σ')
                where V = hpp.(σ, σ') ;
        κ : = κ++⟨f.σ.η⟩;
        σ, σ', δ : = σ++η, σ' ↓ |η|, e.(δ, η)
    od;
    return κ
  ]|
```

```
decode.κ0 =
{ pre: ⟨∃σ0 : : κ0 = encode.σ0⟩ }
  |[ var σ : Σ*; κ' : K*; δ : E
  | κ', σ, δ : = κ0, λ, δinit;
    do κ' ≠ λ → { δ△ = gR.σ }
        let κ' :: ⟨c⟩++ρ' ;
        σ, δ : = σ++g.σ.c, e.(δ, g.σ.c);
        κ' : = ρ'
    od;
    return σ
  ]|
```

Proof 4.34 (LongestReferent)
For all V, σ, nonempty σ' and η it holds that

$h_{sel}.(V,\sigma,\sigma') = \eta$

$\Rightarrow$ $\{$definition $h_{sel}.(V,\sigma,\sigma')\}$

$\langle \forall \alpha : \alpha \in V : |\eta| \geq |\alpha| \rangle$ □

Proof 4.35 (ProperSelect)
For all nonempty V, σ and nonempty σ' it holds that

$h_{sel}.(V,\sigma,\sigma') \in V$

$\equiv$ $\{h_{sel}.(V,\sigma,\sigma')$ must have some value to be an element of $V\}$

$\langle \exists \eta : : h_{sel}.(V,\sigma,\sigma') = \eta \wedge \eta \in V \rangle$

$\equiv$ $\{$definition $h_{sel}.(V,\sigma,\sigma')$, idempotence $\wedge\}$

$\langle \exists \eta : : \eta \in V \wedge \langle \forall \alpha : \alpha \in V : |\eta| \geq |\alpha| \rangle \rangle$

$\equiv$ $\{V \neq \emptyset$, a longest exists$\}$

true □

Remark 4.36 (Greedy parsing)
The selection of the longest element of $h_{pp}.(\sigma,\sigma')$ is called "greedy parsing"[Wel84]. It is called greedy because the longest proper prefix of σ' is used. □

Remark 4.37 (LongestReferent)
The LongestReferent assumption can be simplified if V has two properties:

- V only contains prefixes of σ'
- V is prefix closed with some base M

The selection of the longest now means selecting the longest prefix of σ' in V. Because it is prefix closed, this means that the longest prefix is the only prefix in V for which a prefix of one symbol longer is not in V. There is one exception, namely the prefix that is σ'. For this prefix there is no longer prefix.

The simplified assumption is

$h_{sel}.(V, \sigma, \sigma') = \eta \Rightarrow (\eta \in V \wedge \eta \neq \sigma' \Rightarrow \sigma' \upharpoonleft (|\eta| + 1) \notin V)$.

Take for example $\sigma' = \mathsf{abcdefg}$. Assume that $V = \{\mathsf{abc}, \mathsf{abcd}, \mathsf{abcde}\}$. V contains only prefixes of σ' and it is prefix closed with base 3. abcde is the only string in V for which a prefix of one symbol longer – abcdef in this case – is not in V.

$h_{sel}.(V, \sigma, \sigma')$ is always used in combination with $h_{pp}.(\sigma, \sigma')$. $h_{pp}.(\sigma, \sigma')$ does only contain prefixes. It is not necessarily prefix closed with some base M. But if $dom.(f.\sigma)$ is prefix closed, then $h_{pp}.(\sigma, \sigma')$ is prefix closed too. □

4.4.1.3 $\mathcal{E} = \mathcal{D}_{vu}$ and definition of δ_{init}

This subsubsection introduces a virtual unlimited dictionary – see §4.5.1.2.2 – and it defines δ_{init}. Because the changes are so little, no new specification is given. No additional proofs can be given.

Discriminator 4.38 ($\mathcal{E} = \mathcal{D}_{vu}$)
A virtually unlimited dictionary $\mathcal{D}_{vu}$ is used as an implementation for the abstract data type $\mathcal{E}$. □

Discriminator 4.39 (δ_{init})
Initialize the dictionary with all elements of Σ to satisfy MinDom$f$ initially. This can be done with

$\delta_{init} := \varnothing$;
for all $\alpha \in \Sigma$ **do** $\delta_{init} := \delta_{init} \oplus \alpha$ **od**

□

4.4.1.4 Altering the dictionary

A dictionary can be altered by adding and removing referents and references. The alteration of the dictionary is managed by the function e. But because a virtual unlimited dictionary is used, removal is no concern. Consequently, e manages the addition for the dictionary. Several possibilities for addition appear in literature

- $\eta + 1$-addition

- (previousη)$+\!\!+\eta$-addition
- (previousη)$+\!\!+$(all prefixes η)-addition
- "Y"-addition

These methods for addition are will be addressed next.

Remark 4.40 (Removal used by virtually unlimited dictionaries)
Some virtual unlimited dictionaries automatically remove a tuple from the dictionary if an addition is made to a full dictionary. Because the MinDom$f$ assumption has to be satisfied, not just any tuple can be removed. □

4.4.1.4.1 $\eta + 1$-addition

Discriminator 4.41 ($\eta + 1$-addition)
The goal of this discriminator is to add η concatenated with the symbol that follows η. For the encoder this is quite simple, because the next symbol is known. For the decoder it is not so easy, the next symbol is part of the next element of κ'. To keep the addition identical for the encoder and decoder at this point, it is not used that the encoder knows its next symbol.

The string that will be added conditional is $\eta +\!\!+ (\eta \upharpoonright 1)$. If the next symbol gets known, then the conditional addition is removed and the actual addition is made.

In terms of the function e this is
$e.(\delta, \eta) = \delta \ominus \delta.LA \oplus (\delta.LA \downharpoonleft 1 +\!\!+ \eta \upharpoonright 1) \oplus (\eta +\!\!+ \eta \upharpoonright 1)$.
Operator LA stands for the "Last Added". □

There are still some possibilities for the virtual unlimited dictionary. Each kind of virtual unlimited dictionary reacts differently on addition to a full dictionary.

- $\mathcal{D}_{vu} = \mathcal{D}_{frz}$

  Discriminator 4.42 ($\mathcal{D}_{vu} = \mathcal{D}_{frz}$)
  Use a freeze dictionary $\mathcal{D}_{frz}$ as an implementation for the virtual unlimited dictionary. The freeze dictionary ignores addition to a full dictionary. If the dictionary is full, then it freezes. □

  Proof 4.43 (SubstituteId$\mathcal{E}$)
  The data type $\mathcal{D}_{frz}$ assures that SubstituteId$\mathcal{E}$ is valid. □

  ★ [[**This is LZW[Wel84] in its abstract representation.**

- $\mathcal{D}_{vu} = \mathcal{D}_{CRD}$

Discriminator 4.44 ($\mathcal{D}_{vu} = \mathcal{D}_{CRD}$)
Use a "Compression Rate Drop" dictionary $\mathcal{D}_{CRD}$ as an implementation for the virtual unlimited dictionary. This dictionary type clears the dictionary if something is added to a full dictionary, but only if the compression ratio has dropped below some value. □

Proof 4.45 (SubstituteId$\mathcal{E}$)
The data type $\mathcal{D}_{CRD}$ assures that SubstituteId$\mathcal{E}$ is valid. □

★ [[**This is LZC[BCW90] in its abstract representation.**

- $\mathcal{D}_{vu} = \mathcal{D}_{LRU}$

Discriminator 4.46 ($\mathcal{D}_{vu} = \mathcal{D}_{LRU}$)
Use a "Least Recently Used" dictionary $\mathcal{D}_{LRU}$ as an implementation for the virtual unlimited dictionary. This dictionary type removes the least recently used tuple before something is added to a full dictionary. □

Proof 4.47 (SubstituteId$\mathcal{E}$)
The data type $\mathcal{D}_{LRU}$ assures that SubstituteId$\mathcal{E}$ is valid. □

★ [[**This is LZT[BCW90] in its abstract representation.**

4.4.1.4.2 (previous η)$+\!\!+\eta$-addition

Discriminator 4.48 ((previous η)$+\!\!+\eta$**-addition)**
Use the previous η concatenated with the current η for adaption. The function $e.(\delta, \eta)$ can be specified with $\delta \oplus (\eta_{prev} +\!\!+ \eta)$. □

Discriminator 4.49 ($\mathcal{D}_{vu} = \mathcal{D}_{LRU}$)
Use a "Least Recently Used" dictionary $\mathcal{D}_{LRU}$ as an implementation for the virtual unlimited dictionary. This dictionary removes the least recently used tuple before something is added to a full dictionary. □

Proof 4.50 (SubstituteId$\mathcal{E}$)
The data type $\mathcal{D}_{LRU}$ assures that SubstituteId$\mathcal{E}$ is valid. □

★ [[**This is LZMW[MW85] in its abstract representation.**

Remark 4.51
LZMW does not construct a prefix closed $dom.(f.\sigma)$. This is done because it is reasoned that the additions that make it prefix closed are not used[MW85]. Take for example the pervious η equal to juggling_ and the current η equal to ball. juggling_b is not likely to be used, only juggling_ball is likely to be used in the future. □

4.4.1.4.3 (previous η)++(all prefixes η)-addition

Discriminator 4.52 ((previous η)++(all prefixes η)-addition)
This method adds several strings. For each nonempty prefix of the current η, add the previous η concatenated with the prefix. The function $e.(\delta, \eta)$ can be specified with

$$\textbf{for } i := 1 \textbf{ to } |\eta| \textbf{ do } \delta := \delta \oplus (\eta_{prev} +\!\!+ \eta \upharpoonleft i) \textbf{ od}$$

□

Discriminator 4.53 ($\mathcal{D}_{vu} = \mathcal{D}_{LFU}$)
Use a "Least Frequently Used" dictionary $\mathcal{D}_{LFU}$ as an implementation for the virtual unlimited dictionary. This dictionary removes the least frequently used tuple before something is added to a full dictionary. □

Proof 4.54 (SubstituteId$\mathcal{E}$)
The data type $\mathcal{D}_{LFU}$ assures that SubstituteId$\mathcal{E}$ is valid. □

★ [[**This is LZAP[Ern92, Ber91] in its abstract representation.**

4.4.1.4.4 "Y"-addition

Discriminator 4.55 ("Y"-addition)
This method adds several strings. It is describes by:
For each symbol a in η, add all possibilities for $\alpha +\!\!+ \langle a \rangle$ with

- α is a string which ends at the symbol before a.
- α is in δ_{-1}
- $\alpha +\!\!+ \langle a \rangle$ is not in δ_{-1}

δ_{-1} is the dictionary without the additions made for the previous a.

These additions can be specified with an algorithm[Ber91][4]. This algorithm needs an additional variable φ, which is initial empty.

$$
\begin{array}{l}
\textbf{do } \eta \neq \lambda \rightarrow \\
\quad \textbf{do } \varphi +\!\!+ \eta \upharpoonleft 1 \notin \delta \rightarrow \\
\quad\quad \delta := \delta \oplus (\varphi +\!\!+ \eta \upharpoonleft 1); \\
\quad\quad \varphi := \varphi \downharpoonright 1 \\
\quad \textbf{od}; \\
\quad \varphi, \eta := \varphi +\!\!+ \eta \upharpoonleft 1, \eta \downharpoonright 1 \\
\textbf{od}
\end{array}
$$

The function $e.(\delta, \eta)$ is specified with the above algorithm. □

[4] The algorithm is not formally derived, but a brief textual explantation is given.

Discriminator 4.56 ($\mathcal{D}_{vu} =?$)
Which kind of virtual unlimited dictionary has to be used is not precisely specified. Several are mentioned, but it is not clearly stated which is used. It will probably depend on the implementator which kind will be used. □

★ $\lfloor\lfloor$ **This is LZY[Ber91] in its abstract representation.**

4.4.2 Subgroup $\mathcal{K} = \mathcal{R} \times \Sigma$

This subgroup is one of the major subgroups in this taxonomy. It is indicated in figure 4.4. In this figure the lines of the other subgroups are dashed.

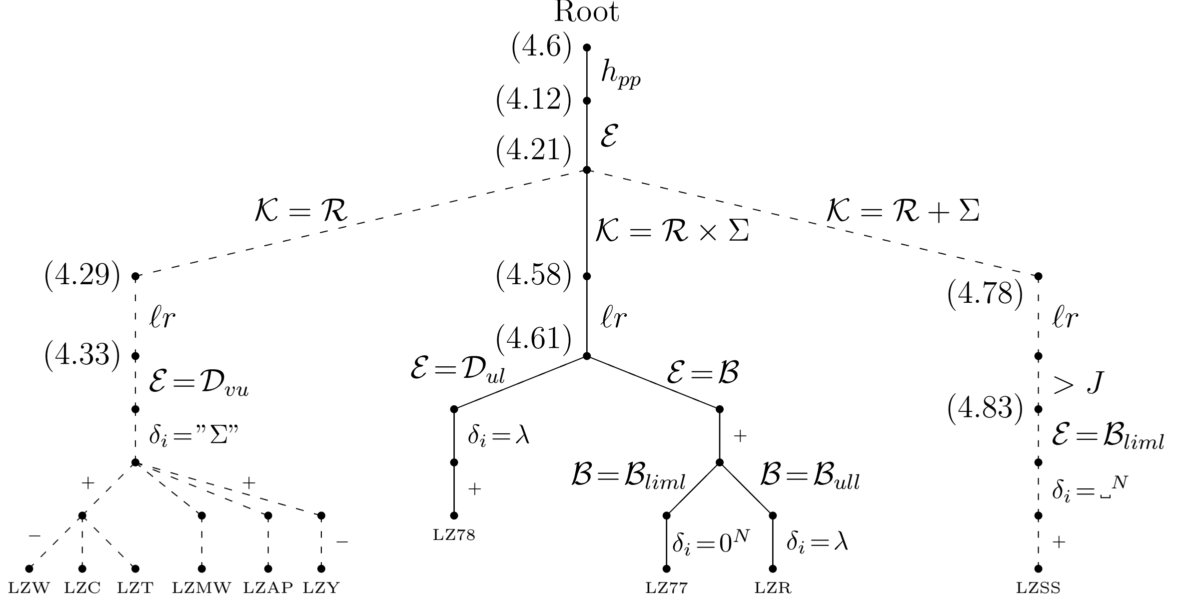


Figure 4.4: Subgroup $\mathcal{K} = \mathcal{R} \times \Sigma$ in the visual summary of the taxonomy

Several discriminators are added in this subgroup. Firstly, the main discriminator of this subgroup – $\mathcal{K} = \mathcal{R} \times \Sigma$ – is introduced. Secondly, the discriminator ℓr is introduced. This selects precisely one η. Thirdly, two different implementations for the environment are given.

4.4.2.1 Introducing $\mathcal{K} = \mathcal{R} \times \Sigma$

This subsubsection introduces $\mathcal{K} = \mathcal{R} \times \Sigma$ by means of discriminator 4.57. Thereafter a new specification is given. With this discriminator the proof for assumption SubstituteId(4.59) can be given. The proof for MinDom$f$(4.60) can also be given, if MinDom$f_{\mathcal{R}}$ is additionally assumed.

Discriminator 4.57 ($\mathcal{K} = \mathcal{R} \times \Sigma$)
The embedment of $\mathcal{R}$ in $\mathcal{K}$ is chosen as $\mathcal{K} = \mathcal{R} \times \Sigma$. The definition of the functions f and g is related to this embedment. They are defined as

- $f.\sigma.(\eta_1 +\!\!+ \langle a \rangle) = (f_{\mathcal{R}}.\sigma.\eta_1, a)$
- $g.\sigma.(r, a) = g_{\mathcal{R}}.\sigma.r +\!\!+ \langle a \rangle$

□

Specification 4.58 (Root with h_{pp}, $\mathcal{E}$, $\mathcal{K} = \mathcal{R} \times \Sigma$)

Declarations:

Input	$\Sigma \in$ Alphabet
Output	$\mathcal{K} \in$ Countable Set
Reference	$\mathcal{R} \in$ Countable Set
Initial $\mathcal{E}$	$\delta_{init} \in \mathcal{E}$
Encode	$\textbf{encode} : \Sigma^* \to \mathcal{K}^*$
Decode	$\textbf{decode} : \mathcal{K}^* \nrightarrow \Sigma^*$
ProperPrefix	$h_{pp} : \Sigma^* \times \Sigma^* \to \mathcal{P}.\Sigma^*$
SelectPrefix	$h_{sel} : \mathcal{P}.\Sigma^* \times \Sigma^* \times \Sigma^* \nrightarrow \Sigma^*$
Substitute	$f : \Sigma^* \to \Sigma^* \nrightarrow \mathcal{K}$
Refer	$f_{\mathcal{R}} : \Sigma^* \to \Sigma^* \nrightarrow \mathcal{R}$
DeSubstitute	$g : \Sigma^* \to \mathcal{K} \nrightarrow \Sigma^*$
DeRefer	$g_{\mathcal{R}} : \Sigma^* \to \mathcal{R} \nrightarrow \Sigma^*$
AdaptEnvironment	$e : \mathcal{E} \times \Sigma^* \to \mathcal{E}$

Definitions:

+	$\mathcal{K} = \mathcal{R} \times \Sigma$
+	$f.\sigma.\eta = f.\sigma.(\eta_1 +\!\!+ \langle a \rangle) = (f_{\mathcal{R}}.\sigma.\eta_1, a)$
+	$g.\sigma.(r, a) = g_{\mathcal{R}}.\sigma.r +\!\!+ \langle a \rangle$
	$h_{pp}.(\sigma, \sigma') = \{\eta \mid \eta \in dom.(f.\sigma) \wedge \eta \preccurlyeq_p \sigma' \wedge \eta \neq \lambda\}$

Assumptions:

ProperSelect		$\left\langle \forall V, \sigma, \sigma' : \sigma' \neq \lambda \wedge V \in \mathcal{P}.\Sigma^* : V \neq \emptyset \Rightarrow h_{sel}.(V, \sigma, \sigma') \in V \right\rangle$
SubstituteId$\mathcal{E}$		$\left\langle \forall \delta, \eta : \eta \in dom.(\delta\bigtriangledown) : (\delta\bigtriangleup) \circ (\delta\bigtriangledown) = \mathsf{id} \right\rangle$
MinDom$f_{\mathcal{R}}$	+	$\left\langle \forall \sigma : \ : \lambda \in dom.(f_{\mathcal{R}}.\sigma) \right\rangle$

Lemmas:

Lossless		$\textbf{decode} \circ \textbf{encode} = \mathsf{id}$	(4.7)
Substitutable		$\left\langle \forall \sigma, \sigma' : \sigma' \neq \lambda : h_{pp}.(\sigma, \sigma') \subseteq dom.(f.\sigma) \right\rangle$	(4.13)
ProperPrefixExists		$\left\langle \forall \sigma, \sigma' : \sigma' \neq \lambda : h_{pp}.(\sigma, \sigma') \neq \emptyset \right\rangle$	(4.14)
Progress		$\left\langle \forall \sigma, \sigma' : \sigma' \neq \lambda : \lambda \notin h_{pp}.(\sigma, \sigma') \right\rangle$	(4.15)
IsPrefix		$\left\langle \forall \sigma, \sigma', \eta : \sigma' \neq \lambda : \eta \in h_{pp}.(\sigma, \sigma') \Rightarrow \eta \preccurlyeq_p \sigma' \right\rangle$	(4.16)
SubstituteId$\mathcal{R}$		$\left\langle \forall \sigma, \eta : \eta \in dom.(f_{\mathcal{R}}.\sigma) : (g_{\mathcal{R}}.\sigma) \circ (f_{\mathcal{R}}.\sigma) = \mathsf{id} \right\rangle$	(4.22)
SubstituteId	+	$\left\langle \forall \sigma, \eta : \eta \in dom.(f.\sigma) : (g.\sigma) \circ (f.\sigma) = \mathsf{id} \right\rangle$	(4.59)
MinDom$f$	+	$\left\langle \forall \sigma : \ : \Sigma \subseteq dom.(f.\sigma) \right\rangle$	(4.60)

Algorithms:

encode.σ_0 =
  $[\![$ **var** $\sigma, \sigma' : \Sigma^*;\ \kappa : \mathcal{K}^*;\ \delta : \mathcal{E}$
  $|\ \sigma', \kappa, \sigma, \delta :\ = \sigma_0, \lambda, \lambda, \delta_{init};$
    do $\sigma' \neq \lambda \rightarrow \{\ \delta\triangledown = f_{\mathcal{R}}.\sigma\ \}$
      let η **such that** $\eta = h_{sel}.(V, \sigma, \sigma')$
          where $V = h_{pp}.(\sigma, \sigma')$;
      $\kappa :\ = \kappa +\!\!+ \langle f.\sigma.\eta\rangle;$
      $\sigma, \sigma', \delta :\ = \sigma +\!\!+ \eta, \sigma' \downharpoonleft |\eta|, e.(\delta, \eta)$
    od;
    return κ
  $]\!]$

decode.κ_0 =
$\{$ pre: $\langle \exists\sigma_0 :\ : \kappa_0 =$ **encode**.$\sigma_0 \rangle\ \}$
  $[\![$ **var** $\sigma : \Sigma^*;\ \kappa' : \mathcal{K}^*;\ \delta : \mathcal{E}$
  $|\ \kappa', \sigma, \delta :\ = \kappa_0, \lambda, \delta_{init};$
    do $\kappa' \neq \lambda \rightarrow \{\ \delta\triangle = g_{\mathcal{R}}.\sigma\ \}$
      let $\kappa' :: \langle c\rangle +\!\!+ \rho'$;
      $\sigma, \delta :\ = \sigma +\!\!+ g.\sigma.c, e.(\delta, g.\sigma.c);$
      $\kappa' :\ = \rho'$
    od;
    return σ
  $]\!]$

Proof 4.59 (SubstituteId)
For all σ and $\eta \in dom.(f.\sigma)$ it holds that

$g.\sigma.(f.\sigma.\eta)$

$\equiv$ {definition $f.\sigma$}

$g.\sigma.(f_{\mathcal{R}}.\sigma.(\eta \downharpoonleft 1), \eta \upharpoonright 1)$

$\equiv$ {definition $g.\sigma$}

$g_{\mathcal{R}}.\sigma.(f_{\mathcal{R}}.\sigma.(\eta \downharpoonleft 1)) +\!\!+ \eta \upharpoonright 1$

$\equiv$ {SubstituteId$\mathcal{R}$}

η □

Proof 4.60 (MinDom$f$)
For all σ it holds that

$dom.(f.\sigma)$

$\equiv$ {definition $f.\sigma$}

$\{\alpha +\!\!+ \langle a\rangle | \alpha \in dom.(f_{\mathcal{R}}.\sigma) \wedge a \in \Sigma\}$

$\supseteq$ {MinDom$f_{\mathcal{R}}$}

Σ □

4.4.2.2 Selecting η (ℓr)

This subsubsection is analog to 4.4.1.2, which also selects η. The discriminators, proofs and remarks are not given again. Only a new specification will be given.

For clarity it is summarized which things were mentioned in 4.4.1.2

- LongestReferent (ℓr) in Discriminator 4.31
- definition of $h_{sel}.(V, \sigma, \sigma')$ in Discriminator 4.32
- proof for LongestReferent in Proof 4.34
- proof for ProperSelect in Proof 4.35
- remarks 4.36 and 4.37

All these points are applicable here too.

Specification 4.61 (Root with h_{pp}, $\mathcal{E}$, $\mathcal{K} = \mathcal{R} \times \Sigma$, ℓr)

Declarations:

Input	$\Sigma \in$ Alphabet
Output	$\mathcal{K} \in$ Countable Set
Reference	$\mathcal{R} \in$ Countable Set
Initial $\mathcal{E}$	$\delta_{init} \in \mathcal{E}$
Encode	$\textbf{encode} : \Sigma^* \to \mathcal{K}^*$
Decode	$\textbf{decode} : \mathcal{K}^* \nrightarrow \Sigma^*$
ProperPrefix	$h_{pp} : \Sigma^* \times \Sigma^* \to \mathcal{P}.\Sigma^*$
SelectPrefix	$h_{sel} : \mathcal{P}.\Sigma^* \times \Sigma^* \times \Sigma^* \nrightarrow \Sigma^*$
Substitute	$f : \Sigma^* \to \Sigma^* \nrightarrow \mathcal{K}$
Refer	$f_{\mathcal{R}} : \Sigma^* \to \Sigma^* \nrightarrow \mathcal{R}$
DeSubstitute	$g : \Sigma^* \to \mathcal{K} \nrightarrow \Sigma^*$
DeRefer	$g_{\mathcal{R}} : \Sigma^* \to \mathcal{R} \nrightarrow \Sigma^*$
AdaptEnvironment	$e : \mathcal{E} \times \Sigma^* \to \mathcal{E}$

Definitions:

$$\mathcal{K} = \mathcal{R} \times \Sigma$$
$$f.\sigma.\eta = f.\sigma.(\eta_1 +\!\!\!+ \langle a \rangle) = (f_{\mathcal{R}}.\sigma.\eta_1, a)$$
$$g.\sigma.(r, a) = g_{\mathcal{R}}.\sigma.r +\!\!\!+ \langle a \rangle$$
$$h_{pp}.(\sigma, \sigma') = \{\eta \mid \eta \in dom.(f.\sigma) \wedge \eta \preccurlyeq_p \sigma' \wedge \eta \neq \lambda\}$$
$$+ \quad h_{sel}.(V, \sigma, \sigma') = \eta \textbf{ such that } \eta \in V \wedge \big\langle \forall \alpha : \alpha \in V : |\eta| \geq |\alpha| \big\rangle$$

Assumptions:

SubstituteId$\mathcal{E}$	$\big\langle \forall \delta, \eta : \eta \in dom.(\delta\triangledown) : (\delta\triangle) \circ (\delta\triangledown) = \mathsf{id} \big\rangle$
MinDom$f_{\mathcal{R}}$	$\big\langle \forall \sigma : \ : \lambda \in dom.(f_{\mathcal{R}}.\sigma) \big\rangle$

Lemmas:

Lossless		$\mathbf{decode} \circ \mathbf{encode} = \mathsf{id}$	(4.7)
Substitutable		$\langle \forall \sigma, \sigma' : \sigma' \neq \lambda : h_{pp}.(\sigma, \sigma') \subseteq dom.(f.\sigma) \rangle$	(4.13)
ProperPrefixExists		$\langle \forall \sigma, \sigma' : \sigma' \neq \lambda : h_{pp}.(\sigma, \sigma') \neq \emptyset \rangle$	(4.14)
Progress		$\langle \forall \sigma, \sigma' : \sigma' \neq \lambda : \lambda \notin h_{pp}.(\sigma, \sigma') \rangle$	(4.15)
IsPrefix		$\langle \forall \sigma, \sigma', \eta : \sigma' \neq \lambda : \eta \in h_{pp}.(\sigma, \sigma') \Rightarrow \eta \preccurlyeq_p \sigma' \rangle$	(4.16)
SubstituteId$\mathcal{R}$		$\langle \forall \sigma, \eta : \eta \in dom.(f_{\mathcal{R}}.\sigma) : (g_{\mathcal{R}}.\sigma) \circ (f_{\mathcal{R}}.\sigma) = \mathsf{id} \rangle$	(4.22)
SubstituteId		$\langle \forall \sigma, \eta : \eta \in dom.(f.\sigma) : (g.\sigma) \circ (f.\sigma) = \mathsf{id} \rangle$	(4.59)
MinDom$f$		$\langle \forall \sigma : : \Sigma \subseteq dom.(f.\sigma) \rangle$	(4.60)
LongestReferent	+	$\langle \forall V, \sigma, \sigma', \eta : \sigma' \neq \lambda \wedge V \in \mathcal{P}.\Sigma^* :$ $h_{sel}.(V.\sigma, \sigma') = \eta \Rightarrow \langle \forall \alpha : \alpha \in V : \lvert\eta\rvert \geq \lvert\alpha\rvert \rangle \rangle$	(4.34)
ProperSelect	+	$\langle \forall V, \sigma, \sigma' : \sigma' \neq \lambda \wedge V \in \mathcal{P}.\Sigma^* :$ $V \neq \emptyset \Rightarrow h_{sel}.(V, \sigma, \sigma') \in V \rangle$	(4.35)

Algorithms:

$\mathbf{encode}.\sigma_0 =$
$[\![\ \mathbf{var}\ \sigma, \sigma' : \Sigma^*;\ \kappa : \mathcal{K}^*;\ \delta : \mathcal{E}$
$\mid \sigma', \kappa, \sigma, \delta :\ = \sigma_0, \lambda, \lambda, \delta_{init};$
$\quad \mathbf{do}\ \sigma' \neq \lambda \rightarrow \{\ \delta \triangledown = f_{\mathcal{R}}.\sigma\ \}$
$\quad\quad \mathbf{let}\ \eta\ \mathbf{such\ that}\ \eta = h_{sel}.(V, \sigma, \sigma')$
$\quad\quad\quad\quad \mathbf{where}\ V = h_{pp}.(\sigma, \sigma')\ ;$
$\quad\quad \kappa :\ = \kappa +\!\!+ \langle f.\sigma.\eta \rangle;$
$\quad\quad \sigma, \sigma', \delta :\ = \sigma +\!\!+ \eta, \sigma' \downharpoonleft \lvert\eta\rvert, e.(\delta, \eta)$
$\quad \mathbf{od};$
$\quad \mathbf{return}\ \kappa$
$]\!]$

$\mathbf{decode}.\kappa_0 =$
$\{$ pre: $\langle \exists \sigma_0 : : \kappa_0 = \mathbf{encode}.\sigma_0 \rangle\ \}$
$[\![\ \mathbf{var}\ \sigma : \Sigma^*;\ \kappa' : \mathcal{K}^*;\ \delta : \mathcal{E}$
$\mid \kappa', \sigma, \delta :\ = \kappa_0, \lambda, \delta_{init};$
$\quad \mathbf{do}\ \kappa' \neq \lambda \rightarrow \{\ \delta \triangle = g_{\mathcal{R}}.\sigma\ \}$
$\quad\quad \mathbf{let}\ \kappa' :: \langle c \rangle +\!\!+ \rho'\ ;$
$\quad\quad \sigma, \delta :\ = \sigma +\!\!+ g.\sigma.c, e.(\delta, g.\sigma.c);$
$\quad\quad \kappa' :\ = \rho'$
$\quad \mathbf{od};$
$\quad \mathbf{return}\ \sigma$
$]\!]$

4.4.2.3 Implementing $\mathcal{E}$

In this subsubsection two implementations for the environment $\mathcal{E}$ are given. They will be addressed next.

4.4.2.3.1 $\mathcal{E} = \mathcal{D}_{ul}$

Several discriminators will be introduced next. First the environment is refined to an unlimited dictionary. The dictionary is initialized next. This proofs the SubstituteId$\mathcal{E}$ assumption(4.63) and the MinDom$f_{\mathcal{R}}$ assumption(4.65). There are no assumption left then any more. The function e for addition is defined last.

Discriminator 4.62 ($\mathcal{E} = \mathcal{D}_{ul}$)
An unlimited dictionary $\mathcal{D}_{ul}$ is used as an implementation for the abstract data type $\mathcal{E}$. □

Proof 4.63 (SubstituteId$\mathcal{E}$)
The data type $\mathcal{D}_{ul}$ assures that SubstituteId$\mathcal{E}$ is valid. □

Discriminator 4.64 (δ_{init})
Initialize with λ to satisfy MinDom$f$ initially. $\delta_{init} := \varnothing \oplus \lambda$ □

Proof 4.65 (MinDom$f_{\mathcal{R}}$)
For all σ it holds that

$\lambda \in dom.(f_{\mathcal{R}}.\sigma)$

$\equiv$ $\{\lambda$ is initial in the dictionary, nothing is removed$\}$

true

□

Discriminator 4.66 (Addition)
Use η for addition. $e.(\delta, \eta) = \delta \oplus \eta$. □

★ [[**This is the core of LZ78[ZL78] in its abstract representation.**

Remark 4.67 (Shell for LZ78)
The actual LZ78 algorithm has a shell around the algorithm given here. Take for example the encoder. The shell splits the actual input of arbitrary length in blocks of some length L. Each block is the input for the core algorithm given here. This blocking strategy prevents that the unlimited dictionary becomes too large. □

4.4.2.3.2 $\mathcal{E} = \mathcal{B}$

Several discriminators will be introduced next. First the environment is refined to a search buffer. This proofs the MinDom$f_{\mathcal{R}}$ assumption(4.69). The function e for addition is defined after the proof. The choices for $\mathcal{B}$ with initializations are mentioned thereafter. After a choice is made for $\mathcal{B}$ the SubstituteId$\mathcal{E}$ can be proven. There are no assumptions left then any more.

Discriminator 4.68 ($\mathcal{E} = \mathcal{B}$)
A search buffer $\mathcal{B}$ is used as an implementation for the abstract data type $\mathcal{E}$. □

Proof 4.69 (MinDom$f_{\mathcal{R}}$)
For all σ it holds that

$\lambda \in dom.(f_{\mathcal{R}}.\sigma)$

$\equiv$ {a substring in the buffer can have length 0}

true

□

Discriminator 4.70 (Addition)
Use η for addition. $e.(\delta, \eta) = \delta \oplus \eta$. □

Choices for $\mathcal{B}$ are

- Limited search buffer with lookahead

  Discriminator 4.71 ($\mathcal{B} = \mathcal{B}_{liml}$)
  A limited search buffer with lookahead $\mathcal{B}_{liml}$ is used as an implementation for the abstract data type $\mathcal{B}$. □

  Proof 4.72 (SubstituteId$\mathcal{E}$)
  The data type $\mathcal{B}_{liml}$ assures that SubstituteId$\mathcal{E}$ is valid. □

  Discriminator 4.73 (δ_{init})
  Initialize with 0^N. $\delta_{init} := \varnothing \oplus 0^N$ □

  ★ [[**This is LZ77[ZL77] in its abstract representation.**

- Unlimited search buffer with lookahead

  Discriminator 4.74 ($\mathcal{B} = \mathcal{B}_{ull}$)
  An unlimited search buffer with lookahead $\mathcal{B}_{ull}$ is used as an implementation for the abstract data type $\mathcal{B}$. □

  Proof 4.75 (SubstituteId$\mathcal{E}$)
  The data type $\mathcal{B}_{ull}$ assures that SubstituteId$\mathcal{E}$ is valid. □

  Discriminator 4.76 (δ_{init})
  Initialize with λ. $\delta_{init} := \varnothing \oplus \lambda$. □

  ★ [[**This is LZR[RPE81] in its abstract representation.**

4.4.3 Subgroup $\mathcal{K} = \mathcal{R} + \Sigma$

This subgroup is one of the major subgroups in this taxonomy. It is indicated in figure 4.5. In this figure the lines of the other subgroups are dashed.

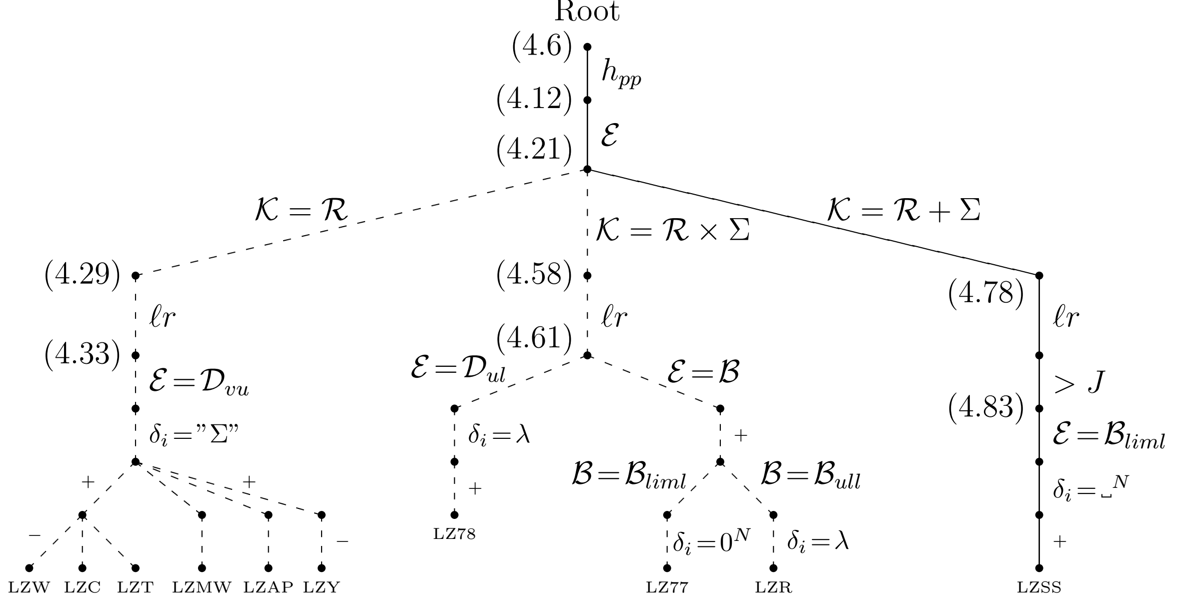


Figure 4.5: Subgroup $\mathcal{K} = \mathcal{R} + \Sigma$ in the visual summary of the taxonomy

Several discriminators are added in this subgroup. Firstly, the main discriminator of this subgroup – $\mathcal{K} = \mathcal{R} + \Sigma$ – is introduced. Secondly, the discriminators ℓr and $> J$ are introduced. This selects precisely one η. Thirdly, a search buffer is chosen for the environment, it is initialized and the method for addition is chosen.

4.4.3.1 Introducing $\mathcal{K} = \mathcal{R} + \Sigma$

This subsubsection introduces $\mathcal{K} = \mathcal{R} + \Sigma$ by means of discriminator 4.77. Thereafter a new specification is given. With this discriminator the proofs for assumptions SubstituteId(4.79) and MinDom$f$(4.80) can be given.

Discriminator 4.77 **($\mathcal{K} = \mathcal{R} + \Sigma$)**
The embedment of $\mathcal{R}$ in $\mathcal{K}$ is chosen as $\mathcal{K} = \mathcal{R} + \Sigma$. The definition of the functions f and g is related to this embedment. They are defined as

- $f.\sigma.\eta = \begin{cases} in_1.(f_{\mathcal{R}}.\sigma.\eta) & \text{if } \eta \in dom.(f_{\mathcal{R}}.\sigma) \\ in_2.\eta & \text{if } \eta \in \Sigma \end{cases}$

- $g.\sigma = (g_{\mathcal{R}}.\sigma)\nabla\mathsf{id}$

□

Specification 4.78 (Root with h_{pp}, $\mathcal{E}$, $\mathcal{K} = \mathcal{R} + \Sigma$)

Declarations:

Input	$\Sigma \in$ Alphabet
Output	$\mathcal{K} \in$ Countable Set
Reference	$\mathcal{R} \in$ Countable Set
Initial $\mathcal{E}$	$\delta_{init} \in \mathcal{E}$
Encode	$\mathbf{encode} : \Sigma^* \to \mathcal{K}^*$
Decode	$\mathbf{decode} : \mathcal{K}^* \nrightarrow \Sigma^*$
ProperPrefix	$h_{pp} : \Sigma^* \times \Sigma^* \to \mathcal{P}.\Sigma^*$
SelectPrefix	$h_{sel} : \mathcal{P}.\Sigma^* \times \Sigma^* \times \Sigma^* \nrightarrow \Sigma^*$
Substitute	$f : \Sigma^* \to \Sigma^* \nrightarrow \mathcal{K}$
Refer	$f_{\mathcal{R}} : \Sigma^* \to \Sigma^* \nrightarrow \mathcal{R}$
DeSubstitute	$g : \Sigma^* \to \mathcal{K} \nrightarrow \Sigma^*$
DeRefer	$g_{\mathcal{R}} : \Sigma^* \to \mathcal{R} \nrightarrow \Sigma^*$
AdaptEnvironment	$e : \mathcal{E} \times \Sigma^* \to \mathcal{E}$

Definitions:

$$+ \quad \mathcal{K} = \mathcal{R} + \Sigma$$

$$+ \quad f.\sigma.\eta = \begin{cases} in_1.(f_{\mathcal{R}}.\sigma.\eta) & \text{if } \eta \in dom.(f_{\mathcal{R}}.\sigma) \\ in_2.\eta & \text{if } \eta \in \Sigma \end{cases}$$

$$+ \quad g.\sigma = (g_{\mathcal{R}}.\sigma)\nabla\mathsf{id}$$

$$h_{pp}.(\sigma, \sigma') = \{\eta | \eta \in dom.(f.\sigma) \wedge \eta \preccurlyeq_p \sigma' \wedge \eta \neq \lambda\}$$

Assumptions:

ProperSelect	$\langle \forall V, \sigma, \sigma' : \sigma' \neq \lambda \wedge V \in \mathcal{P}.\Sigma^* : V \neq \emptyset \Rightarrow h_{sel}.(V, \sigma, \sigma') \in V \rangle$
SubstituteId$\mathcal{E}$	$\langle \forall \delta, \eta : \eta \in dom.(\delta\triangledown) : (\delta\triangle) \circ (\delta\triangledown) = \mathsf{id} \rangle$

Lemmas:

Lossless		$\mathbf{decode} \circ \mathbf{encode} = \mathsf{id}$	(4.7)
Substitutable		$\langle \forall \sigma, \sigma' : \sigma' \neq \lambda : h_{pp}.(\sigma, \sigma') \subseteq dom.(f.\sigma) \rangle$	(4.13)
ProperPrefixExists		$\langle \forall \sigma, \sigma' : \sigma' \neq \lambda : h_{pp}.(\sigma, \sigma') \neq \emptyset \rangle$	(4.14)
Progress		$\langle \forall \sigma, \sigma' : \sigma' \neq \lambda : \lambda \notin h_{pp}.(\sigma, \sigma') \rangle$	(4.15)
IsPrefix		$\langle \forall \sigma, \sigma', \eta : \sigma' \neq \lambda : \eta \in h_{pp}.(\sigma, \sigma') \Rightarrow \eta \preccurlyeq_p \sigma' \rangle$	(4.16)
SubstituteId$\mathcal{R}$		$\langle \forall \sigma, \eta : \eta \in dom.(f_{\mathcal{R}}.\sigma) : (g_{\mathcal{R}}.\sigma) \circ (f_{\mathcal{R}}.\sigma) = \mathsf{id} \rangle$	(4.22)
SubstituteId	+	$\langle \forall \sigma, \eta : \eta \in dom.(f.\sigma) : (g.\sigma) \circ (f.\sigma) = \mathsf{id} \rangle$	(4.79)
MinDom$f$	+	$\langle \forall \sigma : \ : \Sigma \subseteq dom.(f.\sigma) \rangle$	(4.80)

Algorithms:

encode.σ_0 =
  $[\![$ **var** $\sigma, \sigma' : \Sigma^*;\ \kappa : \mathcal{K}^*;\ \delta : \mathcal{E}$
  $|\ \sigma', \kappa, \sigma, \delta :=\sigma_0, \lambda, \lambda, \delta_{init};$
   do $\sigma' \neq \lambda \rightarrow \{\ \delta\triangledown = f_{\mathcal{R}}.\sigma\ \}$
    let η **such that** $\eta = h_{sel}.(V, \sigma, \sigma')$
      where $V = h_{pp}.(\sigma, \sigma')$;
    $\kappa := \kappa +\!\!+ \langle f.\sigma.\eta\rangle;$
    $\sigma, \sigma', \delta := \sigma +\!\!+ \eta, \sigma' \downharpoonleft |\eta|, e.(\delta, \eta)$
   od;
   return κ
  $]\!]$

decode.κ_0 =
 $\{$ pre: $\langle \exists \sigma_0 : \ : \kappa_0 =$ **encode**.$\sigma_0 \rangle\ \}$
  $[\![$ **var** $\sigma : \Sigma^*;\ \kappa' : \mathcal{K}^*;\ \delta : \mathcal{E}$
  $|\ \kappa', \sigma, \delta := \kappa_0, \lambda, \delta_{init};$
   do $\kappa' \neq \lambda \rightarrow \{\ \delta\triangle = g_{\mathcal{R}}.\sigma\ \}$
    let $\kappa' :: \langle c\rangle +\!\!+ \rho'$;
    $\sigma, \delta := \sigma +\!\!+ g.\sigma.c, e.(\delta, g.\sigma.c);$
    $\kappa' := \rho'$
   od;
   return σ
  $]\!]$

Proof 4.79 (SubstituteId)
For all σ and $\eta \in dom.(f.\sigma)$ it holds that

$g.\sigma.(f.\sigma.\eta)$

$\equiv$ {definition $f.\sigma$}

$$g.\sigma.c \textbf{ where } c = \begin{cases} in_1.(f_{\mathcal{R}}.\sigma.\eta) & \text{if } \eta \in dom.(f_{\mathcal{R}}.\sigma) \\ in_2.\eta & \text{if } \eta \in \Sigma \end{cases}$$

$\equiv$ {definition $g.\sigma$ and property of ∇}

$$\begin{cases} (g_{\mathcal{R}}.\sigma).(f_{\mathcal{R}}.\sigma.\eta) & \text{if } \eta \in dom.(f_{\mathcal{R}}.\sigma) \\ \mathsf{id}.\eta & \text{if } \eta \in \Sigma \end{cases}$$

$\equiv$ {SubstituteId$\mathcal{R}$, $\eta \in dom.(f_{\mathcal{R}}.\sigma)$}

η □

Proof 4.80 (MinDom$f$)
For all σ it holds that

$dom.(f.\sigma)$

$\equiv$ {definition $f.\sigma$}

$\{\alpha | \alpha \in dom.(f_{\mathcal{R}}.\sigma) \vee \alpha \in \Sigma\}$

$\supseteq$ {all $\alpha \in \Sigma$ are in the set}

Σ □

4.4.3.2 Selecting η (ℓr and $> J$)

This subsubsection makes the selection of η deterministic. $h_{sel}.(V, \sigma, \sigma')$ is defined in such a way that only one element of V is selected(4.81). The definition uses the new function $h_{sel_{\mathcal{R}\Sigma}}$, which is defined in 4.82. The definitions are respectively "signed" by ℓr and $> J$.

With these definitions the proof for assumption ProperSelect(4.84) can be given. After the proof the name LongestReferent (ℓr) is clarified in Remark 4.85.

Discriminator 4.81 (LongestReferent (ℓr))
In this discriminator, the set V form the function h_{sel} is "filtered". Two elements are filtered out of the set, the shortest and the longest. The selection of one of the two filtered elements will be done with a new function, $h_{sel_{\mathcal{R}\Sigma}} : \Sigma^* \times \Sigma^* \times \Sigma^* \times \Sigma^* \to \Sigma^*$.

The definition for $h_{sel}.(V, \sigma, \sigma')$ is now given directly, without introducing assumptions first:

$h_{sel}.(V, \sigma, \sigma') = h_{sel_{\mathcal{R}\Sigma}}.(\alpha, \beta, \sigma, \sigma')$ **where**
$\quad\alpha$ **such that** $\alpha \in V \wedge \left\langle \forall\gamma : \gamma \in V : |\alpha| \geq |\gamma| \right\rangle$
$\quad\beta$ **such that** $\beta \in V \wedge \left\langle \forall\gamma : \gamma \in V : |\beta| \leq |\gamma| \right\rangle$

□

Discriminator 4.82 (Choosing $\mathcal{R}$ or Σ ($> J$))
$h_{sel_{\mathcal{R}\Sigma}}$ chooses only the first string if it is long enough[5]. No assumption is introduced for this, the definition is given directly:

$$h_{sel_{\mathcal{R}\Sigma}}.(\alpha, \beta, \sigma, \sigma') = \begin{cases} \alpha & \text{if } |\alpha| > J \\ \beta & \text{if } |\alpha| \leq J \end{cases}$$

□

[5]This is a choice that originates from the implementation of $\mathcal{R}$. If a reference needs two bytes, but the referent is one symbol, then it is more efficient to use the symbol, because a symbol needs one byte. Even if the referent is two symbols long, it is more efficient to use two separate symbols. This is not more efficient in space, but in time. No time is needed to de-refer.

Specification 4.83 (Root with h_{pp}, $\mathcal{E}$, $\mathcal{K} = \mathcal{R} + \Sigma$, ℓr, $> J$)

Declarations:

Input		$\Sigma \in$ Alphabet
Output		$\mathcal{K} \in$ Countable Set
Reference		$\mathcal{R} \in$ Countable Set
Initial $\mathcal{E}$		$\delta_{init} \in \mathcal{E}$
J	+	$J \in \mathbb{N}$
Encode		$\mathbf{encode} : \Sigma^* \rightarrow \mathcal{K}^*$
Decode		$\mathbf{decode} : \mathcal{K}^* \nrightarrow \Sigma^*$
ProperPrefix		$h_{pp} : \Sigma^* \times \Sigma^* \rightarrow \mathcal{P}.\Sigma^*$
SelectPrefix		$h_{sel} : \mathcal{P}.\Sigma^* \times \Sigma^* \times \Sigma^* \nrightarrow \Sigma^*$
$h_{sel_{\mathcal{R}\Sigma}}$	+	$h_{sel_{\mathcal{R}\Sigma}} : \Sigma^* \times \Sigma^* \times \Sigma^* \times \Sigma^* \rightarrow \Sigma^*$
Substitute		$f : \Sigma^* \rightarrow \Sigma^* \nrightarrow \mathcal{K}$
Refer		$f_{\mathcal{R}} : \Sigma^* \rightarrow \Sigma^* \nrightarrow \mathcal{R}$
DeSubstitute		$g : \Sigma^* \rightarrow \mathcal{K} \nrightarrow \Sigma^*$
DeRefer		$g_{\mathcal{R}} : \Sigma^* \rightarrow \mathcal{R} \nrightarrow \Sigma^*$
AdaptEnvironment		$e : \mathcal{E} \times \Sigma^* \rightarrow \mathcal{E}$

Definitions:

$$\mathcal{K} = \mathcal{R} + \Sigma$$

$$f.\sigma.\eta = \begin{cases} in_1.(f_{\mathcal{R}}.\sigma.\eta) & \text{if } \eta \in dom.(f_{\mathcal{R}}.\sigma) \\ in_2.\eta & \text{if } \eta \in \Sigma \end{cases}$$

$$g.\sigma = (g_{\mathcal{R}}.\sigma)\nabla\mathsf{id}$$

$$h_{pp}.(\sigma, \sigma') = \{\eta | \eta \in dom.(f.\sigma) \wedge \eta \preccurlyeq_p \sigma' \wedge \eta \neq \lambda\}$$

$$h_{sel}.(V, \sigma, \sigma') = h_{sel_{\mathcal{R}\Sigma}}.(\alpha, \beta, \sigma, \sigma') \ \textbf{where}$$

$+ \quad$ $$\alpha \ \textbf{such that} \ \alpha \in V \wedge \langle \forall \gamma : \gamma \in V : |\alpha| \geq |\gamma| \rangle$$
$$\beta \ \textbf{such that} \ \beta \in V \wedge \langle \forall \gamma : \gamma \in V : |\beta| \leq |\gamma| \rangle$$

$+ \quad$ $$h_{sel_{\mathcal{R}\Sigma}}.(\alpha, \beta, \sigma, \sigma') = \begin{cases} \alpha & \text{if } |\alpha| > J \\ \beta & \text{if } |\alpha| \leq J \end{cases}$$

Assumptions:

SubstituteId$\mathcal{E}$ $\quad$ $\langle \forall \delta, \eta : \eta \in dom.(\delta \bigtriangledown) : (\delta \bigtriangleup) \circ (\delta \bigtriangledown) = \mathsf{id} \rangle$

Lemmas:

Lossless		$\mathbf{decode} \circ \mathbf{encode} = \mathsf{id}$	(4.7)
Substitutable		$\langle \forall \sigma, \sigma' : \sigma' \neq \lambda : h_{pp}.(\sigma, \sigma') \subseteq dom.(f.\sigma) \rangle$	(4.13)
ProperPrefixExists		$\langle \forall \sigma, \sigma' : \sigma' \neq \lambda : h_{pp}.(\sigma, \sigma') \neq \emptyset \rangle$	(4.14)
Progress		$\langle \forall \sigma, \sigma' : \sigma' \neq \lambda : \lambda \notin h_{pp}.(\sigma, \sigma') \rangle$	(4.15)
IsPrefix		$\langle \forall \sigma, \sigma', \eta : \sigma' \neq \lambda : \eta \in h_{pp}.(\sigma, \sigma') \Rightarrow \eta \preccurlyeq_p \sigma' \rangle$	(4.16)
SubstituteId$\mathcal{R}$		$\langle \forall \sigma, \eta : \eta \in dom.(f_{\mathcal{R}}.\sigma) : (g_{\mathcal{R}}.\sigma) \circ (f_{\mathcal{R}}.\sigma) = \mathsf{id} \rangle$	(4.22)
SubstituteId		$\langle \forall \sigma, \eta : \eta \in dom.(f.\sigma) : (g.\sigma) \circ (f.\sigma) = \mathsf{id} \rangle$	(4.79)
MinDom$f$		$\langle \forall \sigma : : \Sigma \subseteq dom.(f.\sigma) \rangle$	(4.80)
ProperSelect	+	$\langle \forall V, \sigma, \sigma' : \sigma' \neq \lambda \wedge V \in \mathcal{P}.\Sigma^* :$ $V \neq \emptyset \Rightarrow h_{sel}.(V, \sigma, \sigma') \in V \rangle$	(4.84)

Algorithms:

$\mathbf{encode}.\sigma_0 =$
$[\![$ **var** $\sigma, \sigma' : \Sigma^*;\ \kappa : \mathcal{K}^*;\ \delta : \mathcal{E}$
$|\ \sigma', \kappa, \sigma, \delta : = \sigma_0, \lambda, \lambda, \delta_{init};$
do $\sigma' \neq \lambda \rightarrow \{\ \delta \triangledown = f_{\mathcal{R}}.\sigma\ \}$
let η **such that** $\eta = h_{sel}.(V, \sigma, \sigma')$
where $V = h_{pp}.(\sigma, \sigma')$;
$\kappa : = \kappa +\!\!+ \langle f.\sigma.\eta \rangle;$
$\sigma, \sigma', \delta : = \sigma +\!\!+ \eta, \sigma' \downharpoonleft |\eta|, e.(\delta, \eta)$
od;
return κ
$]\!]$

$\mathbf{decode}.\kappa_0 =$
$\{$ pre: $\langle \exists \sigma_0 : : \kappa_0 = \mathbf{encode}.\sigma_0 \rangle$ $\}$
$[\![$ **var** $\sigma : \Sigma^*;\ \kappa' : \mathcal{K}^*;\ \delta : \mathcal{E}$
$|\ \kappa', \sigma, \delta : = \kappa_0, \lambda, \delta_{init};$
do $\kappa' \neq \lambda \rightarrow \{\ \delta \triangle = g_{\mathcal{R}}.\sigma\ \}$
let $\kappa' :: \langle c \rangle +\!\!+ \rho'$;
$\sigma, \delta : = \sigma +\!\!+ g.\sigma.c, e.(\delta, g.\sigma.c);$
$\kappa' : = \rho'$
od;
return σ
$]\!]$

Proof 4.84 (ProperSelect)
For all nonempty V, σ and nonempty σ' it holds that

$h_{sel}.(V, \sigma, \sigma') \in V$

$\equiv$ $\{h_{sel}.(V, \sigma, \sigma')$ must have some value to be an element of $V\}$

$\langle \exists \eta : : h_{sel}.(V, \sigma, \sigma') = \eta \wedge \eta \in V \rangle$

$\equiv$ $\{V \neq \emptyset$, a shortest and a longest exists, $h_{sel_{\mathcal{R}\Sigma}}.(\alpha, \beta, \sigma, \sigma')$ always selects one$\}$

true □

Remark 4.85 (Name of LongestReferent (ℓr))
The name LongestReferent (ℓr) may seem strange, because the shortest and the longest are filtered out. But h_{sel} is always combined with h_{pp}. With this combination something more can be said than just "shortest and longest".

First, –for clarity– the definition of $h_{pp}.(\sigma, \sigma')$ is rewritten with use of the definition for $f.\sigma$. This results in $\{\alpha | \alpha \in dom.(f_{\mathcal{R}}.\sigma)\ \wedge \alpha \preccurlyeq_p \sigma' \wedge \alpha \neq \lambda\} \cup \{\sigma' \upharpoonright 1\}$.

The elements that are filtered out of $h_{pp}.(\sigma, \sigma')$ are:

- The shortest element always is $\sigma' \upharpoonright 1$, this element is always in the set and there is no shorter one possible.
- There are two possibilities for the longest
  - If the set contains any referents, then the longest one of the set is a referent. This is valid to say, because all referents in the set have at least length 1 and the non-referent has precisely length 1.
  - If the set contains no referents, the longest element of the set is $\sigma' \upharpoonright 1$.

If now a selection is made between the shortest and the longest, then this can result in $\sigma' \upharpoonright 1$ or in the longest referent. Consequently, if a referent is eventually selected, then this can only be the longest referent. □

4.4.3.3 $\mathcal{E} = \mathcal{B}_{lim}$, definition of δ_{init} and addition

This subsubsection introduces a search buffer – see §4.5.2 – , it defines δ_{init} and it chooses the addition method. Because the changes are so little, no new specification is given.

The proofs for the SubstituteId$\mathcal{E}$(4.87) assumption and MinDom$f_{\mathcal{R}}$(4.88) assumption can be given on basis of the choice of the search buffer. There are no assumption left then any more.

Discriminator 4.86 ($\mathcal{E} = \mathcal{B}_{liml}$)
A limited search buffer with lookahead $\mathcal{B}_{liml}$ is used as an implementation for the abstract data type $\mathcal{E}$. □

Proof 4.87 (SubstituteId$\mathcal{E}$)
The data type $\mathcal{B}_{liml}$ assures that SubstituteId$\mathcal{E}$ is valid. □

Proof 4.88 (MinDom$f_{\mathcal{R}}$)
For all σ it holds that

$\lambda \in dom.(f_{\mathcal{R}}.\sigma)$

$\equiv$ {a substring in the buffer can have length 0}

true

□

Discriminator 4.89 (δ_{init})
It is not precisely stated how the buffer has to be initiated. In examples it is used to fill the buffer initially with all blanks. $\delta_{init} := \varnothing \oplus \llcorner^{N}$ □

Discriminator 4.90 (Addition)
Use η for addition. $e.(\delta, \eta) = \delta \oplus \eta$. □

★ [[**This is LZSS[Bel86] in its abstract representation.**

Remark 4.91 (Appearance of algorithm)
The appearance of the algorithm given here is bit different to the abstract version of LZSS. Although they look different, they are equivalent.

As explained in 4.85, $h_{sel}.(h_{pp}.(\sigma, \sigma'), \sigma, \sigma')$ can only result in the longest referent or in the single symbol $\sigma' \upharpoonleft 1$. LZSS first determines this longest referent. It also allows λ as a referent to guarantee that a longest referent exists. Thereafter, the length of the longest referent is evaluated. If it is too short – for example if the "fail-safe"-referent λ is the longest one –, then the symbol $\sigma' \upharpoonleft 1$ has to be taken.

The total **let**-statement of the algorithm given here can be rewritten as:

let η **such that** $\eta \in dom.(f_{\mathcal{R}}.\sigma) \wedge \eta \preccurlyeq_p \sigma \wedge$
$\langle \forall \alpha : \alpha \in dom.(f_{\mathcal{R}}.\sigma) \wedge \alpha \preccurlyeq_p \sigma : |\eta| \geq |\alpha| \rangle$;
{ η is the longest prefix of σ' that is in $dom.(f_{\mathcal{R}}.\sigma)$ }
if $|\eta| > J \rightarrow$ skip
⫿ $|\eta| \leq J \rightarrow \eta :=\sigma' \upharpoonleft 1$
fi

For this alternative, it is additionally needed that λ is a referent ($\lambda \in dom.(f_{\mathcal{R}}.\sigma)$). Because a search buffer is used, it is known that λ is a referent. Although λ has to be present for progress, it is never selected, because $|\lambda| \leq J$ ($J \in \mathbb{N}$, thus $0 \leq J$) □

4.5 Concrete data types for environment $\mathcal{E}$

The environment $\mathcal{E}$ is an abstract data type, which is defined in Definition 4.19 on page 37. Two concrete data types for $\mathcal{E}$ can be extracted from the literature, the dictionary and the search buffer. They are the subject of the next subsections.

Remark 4.92 (Multiple implementations for a concrete data type)
It can be profitable to make two implementations for one concrete data type. One implementation can be made that performs good for the operators that are mainly used by the encoder, namely the operators $\bigtriangledown$ and $\oplus$. Another implementation can be made that performs good for the operators that are mainly used by the decoder, namely $\bigtriangleup$ and $\oplus$. □

4.5.1 Dictionary

A dictionary is a concrete data type for the abstract environment data type. It is in essence an element of $\mathcal{P}(\Sigma^* \times \mathcal{R})$, which is a set of tuples. Each tuple contains a left and a right member. The left member is a string, which contains the referent. The reference is the right member. The reference is the *key* for the tuple. It is unique for each tuple. The key is here defined as a natural number, which means that $\mathcal{R} = \mathbb{N}$.

Example 4.93 (Dictionary)
Let the tuples in the dictionary be given by the following table

referent(string)	reference(key)
a	90
abc	14
b	3

This specifies that the reference for abc is 14, and the referent for 90 is a. The size of this dictionary is 3. □

All dictionaries are based on the abstract data type $\mathcal{D}$.

Definition 4.94 (Abstract data type dictionary $\mathcal{D}$)
The abstract data type dictionary $\mathcal{D}$ is given by:

$$\mathcal{D} = (\mathcal{P}(\Sigma^* \times \mathcal{R}) | \varnothing, \oplus, \bigtriangledown, \bigtriangleup, \| \;\|)$$

with operators-definitions ($\delta \in \mathcal{D}$)

declaration	definition	precondition
$\varnothing : \mathcal{D}$	$\varnothing = \emptyset$	
$\bigtriangledown : \mathcal{D} \to \Sigma^* \not\to \mathcal{R}$	$\delta \bigtriangledown \alpha = i$ **such that** $(\alpha, i) \in \delta$	$\alpha \in dom.(\delta\bigtriangledown)$
$\bigtriangleup : \mathcal{D} \to \mathcal{R} \not\to \Sigma^*$	$\delta \bigtriangleup i = \alpha$ **such that** $(\alpha, i) \in \delta$	$i \in dom.(\delta\bigtriangleup)$
$\| \;\| : \mathcal{D} \to \mathbb{N}$	$\|\delta\| = \lvert\delta\rvert$	

With the assumption that

- SubstituteId$\mathcal{E}$: $\langle \forall \delta, \eta : \eta \in dom.(\delta\bigtriangledown) : (\delta\bigtriangleup) \circ (\delta\bigtriangledown) = \mathsf{id} \rangle$ □

All the dictionary data types can be divided in two groups, the unlimited and the limited dictionary data types.

4.5.1.1 Unlimited dictionary

The unlimited dictionary has the property that its size is unlimited. It can contain an unlimited number of tuples.

Definition 4.95 (Unlimited dictionary data type)
The unlimited dictionary data type $\mathcal{D}_{ul}$ is a data type that is given by
$\mathcal{D}_{ul} = (\mathcal{P}(\Sigma^* \times \mathcal{R}) | \varnothing, \oplus, \bigtriangledown, \bigtriangleup, \|\ \|)$
with operators-definitions ($\delta \in \mathcal{D}_{ul}$)

declaration	definition	precondition
$\varnothing : \mathcal{D}_{ul}$	$\varnothing = \emptyset$	
$\oplus : \mathcal{D}_{ul} \times \Sigma^* \rightarrow \mathcal{D}_{ul}$	$\delta \oplus \alpha = \delta \cup \{(\alpha, \mathbf{new}.i)\}$	
$\bigtriangledown : \mathcal{D}_{ul} \rightarrow \Sigma^* \not\rightarrow \mathcal{R}$	$\delta \bigtriangledown \alpha = i$ **such that** $(\alpha, i) \in \delta$	$\alpha \in dom.(\delta\bigtriangledown)$
$\bigtriangleup : \mathcal{D}_{ul} \rightarrow \mathcal{R} \not\rightarrow \Sigma^*$	$\delta \bigtriangleup i = \alpha$ **such that** $(\alpha, i) \in \delta$	$i \in dom.(\delta\bigtriangleup)$
$\|\ \| : \mathcal{D}_{ul} \rightarrow \mathbb{N}$	$\|\delta\| = \lvert\delta\rvert$	

The proof for SubstituteId$\mathcal{E}$ is given in 4.96. □

Proof 4.96 (SubstituteId$\mathcal{E}$)
For all δ and $\eta \in dom.(\delta\bigtriangledown)$ it holds that

$\delta \bigtriangleup (\delta \bigtriangledown \eta)$

$\equiv \quad \left\{ \begin{array}{l} \text{definition } \bigtriangledown,\ \eta \in dom.(\delta\bigtriangledown) \\ \bullet\ i \textbf{ such that } (\eta, i) \in \delta \end{array} \right\}$

$\delta \bigtriangleup i$

$\equiv \quad \{\text{definition } \bigtriangleup,\ i \in dom.(\delta\bigtriangleup)\}$

α **such that** $(\alpha, i) \in \delta$

$\equiv \quad \{i \text{ is unique, only one tuple has } i \text{ as right-member}\}$

η □

Remark 4.97 (Minimal domain requirement)
If a dictionary has a minimal domain requirement $\langle \forall \delta, \alpha : \alpha \in W : \alpha \in dom.(\delta\bigtriangledown) \rangle$, then it can be satisfied by initiating the dictionary with all the elements of W. Because nothing

is removed from the unlimited dictionary, every element of W will always stay in the dictionary. $\square$

Remark 4.98 (Alternatives for $\alpha \in dom.(\delta\triangledown)$ and $i \in dom.(\delta\triangle)$)

- $\alpha \in dom.(\delta\triangledown) \equiv \langle \exists i : \ : (\alpha, i) \in \delta \rangle$
- $i \in dom.(\delta\triangle) \equiv \langle \exists \alpha : \ : (\alpha, i) \in \delta \rangle$ $\square$

4.5.1.2 Limited dictionary

There are several limited dictionaries. They have the common property that their size is limited to some number N. Two limited kind of dictionaries will be given next, the standard limited dictionary and the virtually unlimited dictionary.

4.5.1.2.1 Standard limited dictionary

The standard limited dictionary is almost the same as a unlimited dictionary. Because the limited dictionary is full after N additions, the operator $\ominus$ is introduced to make it possible to remove a tuple from the dictionary. It does the opposite of $\oplus$.

Definition 4.99 (Standard limited dictionary data type)
The standard limited dictionary data type $\mathcal{D}_{lim}$ is a data type that is given by
$\mathcal{D}_{lim} = (\mathcal{P}(\Sigma^* \times \mathcal{R}) | \varnothing, \oplus, \ominus, \triangledown, \triangle, \| \ \|)$
with operators-definitions ($\delta \in \mathcal{D}_{lim}$)

declaration	definition	precondition
$\varnothing : \mathcal{D}_{lim}$	$\varnothing = \emptyset$	
$\oplus : \mathcal{D}_{lim} \times \Sigma^* \to \mathcal{D}_{lim}$	$\delta \oplus \alpha = \delta \cup \{(\alpha, \mathbf{new}.i)\}$	$\|\delta\| < N$
$\ominus : \mathcal{D}_{lim} \times \Sigma^* \to \mathcal{D}_{lim}$	$\delta \ominus \alpha = \delta \backslash (\alpha, \delta \triangledown \alpha)$	$(\alpha, \delta \triangledown \alpha) \in \delta$
$\triangledown : \mathcal{D}_{lim} \to \Sigma^* \nrightarrow \mathcal{R}$	$\delta \triangledown \alpha = i$ **such that** $(\alpha, i) \in \delta$	$\alpha \in dom.(\delta\triangledown)$
$\triangle : \mathcal{D}_{lim} \to \mathcal{R} \nrightarrow \Sigma^*$	$\delta \triangle i = \alpha$ **such that** $(\alpha, i) \in \delta$	$i \in dom.(\delta\triangle)$
$\| \ \| : \mathcal{D}_{lim} \to \mathbb{N}$	$\|\delta\| = \lvert\delta\rvert$	

The proof for SubstituteId$\mathcal{E}$ is analog to Proof 4.96. The size limitation has no influence on the proof. $\square$

Remark 4.100 (Minimal domain requirement)
A minimal domain requirement stated by $\langle \forall \delta, \alpha : \alpha \in W : \alpha \in dom.(\delta\triangledown) \rangle$ can be satisfied.

It can be satisfied by initiating the dictionary with all the elements of W, and introducing the precondition $\alpha \notin W$ for the operator $\ominus$. $\square$

4.5.1.2.2 Virtually unlimited dictionary

A virtual unlimited dictionary is a limited dictionary that looks unlimited. It allows addition to a full dictionary. What will be done with the addition if the dictionary is full depends on the kind of virtual unlimited dictionary.

Definition 4.101 (Virtual unlimited dictionary abstract data type)
The virtual unlimited dictionary $\mathcal{D}_{vu}$ is an abstract data type. The definition is almost the same as the standard limited dictionary, except that the operator $\oplus$ is changed to

$$\delta \oplus \alpha = \begin{cases} \delta \cup \{(\alpha, \mathbf{new}.i)\} & \text{if } \|\delta\| < N \\ h.(\delta, \alpha) & \text{if } \|\delta\| = N \end{cases}$$

The assumption SubstituteId$\mathcal{E}$ still has to be proven:

- SubstituteId$\mathcal{E}$: $\langle \forall \delta, \eta : \eta \in dom.(\delta\triangledown) : (\delta\triangle) \circ (\delta\triangledown) = \mathsf{id} \rangle$

□

There are several concrete implementations for the abstract virtual unlimited dictionary. The proofs for the SubstituteId$\mathcal{E}$ assumption are analog to Proof 4.96.

- Freeze dictionary $\mathcal{D}_{frz}$
  $h.(\delta, \alpha) = \delta$

- Clear dictionary $\mathcal{D}_{clr}$
  $h.(\delta, \alpha) = \varnothing \oplus \alpha$

- LRU (Least Recently Used) dictionary $\mathcal{D}_{LRU}$[6]
  $h.(\delta, \alpha) = \delta \ominus \delta.LRU \oplus \alpha$
  This introduces a new operator LRU, which indicates the least recently used tuple.

- LFU (Least Frequently Used) dictionary $\mathcal{D}_{LFU}$
  $h.(\delta, \alpha) = \delta \ominus \delta.LFU \oplus \alpha$
  This introduces a new operator LFU, which indicates the least frequently used tuple.

- Compression Ratio Drop dictionary $\mathcal{D}_{CRD}$
  $$h.(\delta, \alpha) = \begin{cases} \delta & \text{if } \delta.CR > X \\ \varnothing \oplus \alpha & \text{if } \delta.CR \leq X \end{cases}$$
  This introduces a new operator CR, which gives the compression ratio. One method to determine the compression ratio is to accumulate the length of the referents of the last few used tuples.

Remark 4.102 (Minimal domain requirement)
As stated in Remark 4.100, a minimal domain requirement can be satisfied. The requirement for removal has some impact on virtually unlimited dictionaries. The LRU and LFU operators may not result in an element of W. If a dictionary is cleaned, then not all tuples may be removed, only those that do not contain an element of W may be removed. □

[6] all reasonable ways of interpreting LRU will work[MW85]

4.5.2 Search buffer

The search buffer[7] is a concrete data type for the abstract environment data type. The search buffer is in essence a string. A referent is a substring of the buffer-string. A reference is a tuple of two naturals ($\mathcal{R} = \mathbb{N} \times \mathbb{N}$). The first natural gives the offset of the referent in the buffer-string. The offset can be measured form various positions. Here it is chosen to use the offset from the left, with the first symbol of a string having offset 0. Other offset-measurements can also be used. The second natural is the length of the referent.

Example 4.103 (Search buffer)
Assume the buffer-string is abc. This results in the following referents and references

referent	reference
λ	(0,0)
a	(0,1)
ab	(0,2)
abc	(0,3)
λ	(1,0)
b	(1,1)
bc	(1,2)
λ	(2,0)
c	(2,1)

This specifies that the reference for abc is $(0, 3)$, and the referent for $(0, 1)$ is a. The size of this buffer is the length of the buffer-string, which is 3. □

All search buffers are based on the abstract data type $\mathcal{B}$.

Definition 4.104 (Abstract data type buffer $\mathcal{B}$)
The abstract data type buffer $\mathcal{B}$ is given by:
$\mathcal{B} = (\Sigma^* | \varnothing, \oplus, \bigtriangledown, \bigtriangleup, \| \ \|)$
with operators-definitions ($\delta \in \mathcal{B}$)

declaration	definition	precondition
$\varnothing : \mathcal{B}$	$\varnothing = \lambda$	
$\| \ \| : \mathcal{B} \to \mathbb{N}$	$\|\delta\| = \lvert\delta\rvert$	

With the assumption that

- SubstituteId$\mathcal{E}$: $\langle \forall \delta, \eta : \eta \in dom.(\delta \bigtriangledown) : (\delta \bigtriangleup) \circ (\delta \bigtriangledown) = \mathsf{id} \rangle$ □

All the search buffer data types can be divided in two groups, the unlimited and the limited search buffer data types.

[7] A search buffer is sometimes called an implicit dictionary. The opposite – an explicit dictionary – is the dictionary here.

4.5.2.1 Unlimited search buffer

The unlimited buffer has the property that its size is unlimited. The buffer string can have an unlimited length. All substrings are possible as referent. Two unlimited search buffers are included here. The standard unlimited search buffer and the unlimited search buffer with lookahead.

4.5.2.1.1 Standard unlimited search buffer

The standard unlimited search buffer is based on substrings, therefor a definition to determine a substring is given first.

Definition 4.105 (Substring)
The function that determines a substring of a string α on basis of an offset i and a length j is defined as[8]
$SS.(\alpha, i, j) = \alpha \downharpoonleft i \upharpoonleft j$ pre: $0 \leq i \leq |\alpha| \wedge 0 \leq j \leq |\alpha \downharpoonleft i|$ □

Definition 4.106 (Standard unlimited search buffer data type)
The standard unlimited search buffer data type $\mathcal{B}_{ul}$ is a data type that is given by
$\mathcal{B}_{ul} = (\Sigma^*|\varnothing, \oplus, \bigtriangledown, \bigtriangleup, \| \ \|)$
with operators-definitions ($\delta \in \mathcal{B}_{ul}$)

declaration	definition	precondition
$\varnothing : \mathcal{B}_{ul}$	$\varnothing = \lambda$	
$\oplus : \mathcal{B}_{ul} \times \Sigma^* \to \mathcal{B}_{ul}$	$\delta \oplus \alpha = \delta +\!\!+ \alpha$	
$\bigtriangledown : \mathcal{B}_{ul} \to \Sigma^* \not\to \mathcal{R}$	$\delta \bigtriangledown \alpha = (i, j)$ **such that** $SS.(\delta, i, j) = \alpha$	$\alpha \in dom.(\delta\bigtriangledown)$
$\bigtriangleup : \mathcal{B}_{ul} \to \mathcal{R} \not\to \Sigma^*$	$\delta \bigtriangleup (i, j) = SS.(\delta, i, j)$	$(i, j) \in dom.(\delta\bigtriangleup)$
$\| \ \| : \mathcal{B}_{ul} \to \mathbb{N}$	$\|\delta\| = |\delta|$	

The proof for SubstituteId$\mathcal{E}$ is given in 4.107. □

Proof 4.107 (SubstituteId$\mathcal{E}$)
For all δ and $\eta \in dom.(\delta\bigtriangledown)$ it holds that
$\delta \bigtriangleup (\delta \bigtriangledown \eta)$

$$\equiv \quad \left\{ \begin{array}{l} \text{definition } \bigtriangledown,\ \eta \in dom.(\delta\bigtriangledown) \\ \bullet(i, j) \textbf{ such that } SS.(\delta, i, j) = \eta \end{array} \right\}$$

$\delta \bigtriangleup (i, j)$

$$\equiv \quad \{\text{definition } \bigtriangleup,\ i \in dom.(\delta\bigtriangleup)\}$$

[8]This definition uses the offset from the left, starting at 0. The offset from the right, starting at 0, can also be used. This would be defined as $SS.(\alpha, i, j) = \alpha \upharpoonleft i \upharpoonright j$ pre: $0 \leq i \leq |\alpha| \wedge 0 \leq j \leq |\alpha \upharpoonleft i|$.

$SS.(\delta, i, j)$

$\equiv$ $\{SS.(\delta, i, j)$ is deterministic, one substring starts at offset i and has length $j\}$

η

□

Remark 4.108 (Alternatives for $\alpha \in dom.(\delta\triangledown)$ and $i \in dom.(\delta\triangle)$)

- $\alpha \in dom.(\delta\triangledown) \equiv \alpha \preccurlyeq_s \delta$
- $(i, j) \in dom.(\delta\triangle) \equiv 0 \le i \le |\delta| \wedge 0 \le j \le |\delta \downharpoonleft i|$ □

Remark 4.109 (Multiple solutions for $\delta \triangledown \alpha$)
It is possible that there are multiple solutions for $\delta \triangledown \alpha$. That is to say, α may occur on multiple offsets in δ. If this is the case, one has to be selected. For example, it could be chosen to select the one with the smallest offset. □

4.5.2.1.2 Unlimited search buffer with lookahead

The unlimited search buffer with lookahead expands the substring-idea of the standard unlimited search buffer. Except the substrings, there are other strings that can be used as referent, see Example 4.112.

Definition 4.110 (Lookahead Substring)
The function that determines a substring of a string $\alpha + \! + \beta$ on basis of an offset i and a length j is defined as

$SS_{LA}.(\alpha, \beta, i, j) = (\alpha + \! + \beta) \downharpoonleft i \upharpoonleft j \qquad \text{pre: } 0 \le i < |\alpha| \wedge 0 \le j \le |(\alpha + \! + \beta) \downharpoonleft i|$

This definition requires that the substring starts in α. □

Definition 4.111 (Unlimited search buffer with lookahead data type)
The unlimited search buffer with lookahead data type $\mathcal{B}_{ull}$ is a data type that is given by

$\mathcal{B}_{ull} = (\Sigma^* | \varnothing, \oplus, \triangledown, \triangle, \| \ \|)$

with operators-definitions ($\delta \in \mathcal{B}_{ull}$)

declaration	definition	precondition
$\varnothing : \mathcal{B}_{ull}$	$\varnothing = \lambda$	
$\oplus : \mathcal{B}_{ull} \times \Sigma^* \to \mathcal{B}_{ull}$	$\delta \oplus \alpha = \delta + \! + \alpha$	
$\triangledown : \mathcal{B}_{ull} \to \Sigma^* \nrightarrow \mathcal{R}$	$\delta \triangledown \alpha = (i, j)$ **such that** $SS_{LA}.(\delta, \alpha, i, j) = \alpha$	$\alpha \in dom.(\delta\triangledown)$
$\triangle : \mathcal{B}_{ull} \to \mathcal{R} \nrightarrow \Sigma^*$	$\delta \triangle (i, j) = \alpha$ **such that** $SS_{LA}.(\delta, \alpha, i, j) = \alpha$	$(i, j) \in dom.(\delta\triangle)$
$\| \ \| : \mathcal{B}_{ull} \to \mathbb{N}$	$\|\delta\| = |\delta|$	

The proof for SubstituteId$\mathcal{E}$ is analog to Proof 4.107. □

Example 4.112 (Lookahead)
Let the buffer-string be ban. Without lookahead only the substrings of ban are usable. Lookahead adds several strings. It adds for example ana, because $SS_{LA}.(\mathsf{ban}, \mathsf{ana}, 1, 3) = \mathsf{ana}$ is valid. □

Remark 4.113 (Practical definition for $\delta \bigtriangleup (i, j)$)
The definition for $\delta \bigtriangleup (i, j)$ looks different to calculate. But it is not so hard as it seems.

Because $i < |\delta|$ the first element of α can be determined. Let this element be a. It is now known what $\delta +\!\!+ \langle a \rangle$ is. Consequently, the second element of α can be determined too, because $i + 1 < |\delta +\!\!+ \langle a \rangle|$. This process can be repeated until α is totally known.

It can be stated more formally as

$$\alpha \textbf{ such that } SS_{LA}.(\delta, \alpha, i, j) = \alpha$$

$$\equiv \qquad \{i < |\delta|, \text{let } \alpha = \langle a \rangle +\!\!+ \beta\}$$

$$\langle a \rangle +\!\!+ \beta \textbf{ such that } SS_{LA}.(\delta +\!\!+ \langle a \rangle, \beta, i + 1, j - 1) = \beta$$

□

4.5.2.2 Limited search buffer

A limited search buffer is limited in two ways. One limitation is the length of the buffer string, it is limited to length N. The second limitation is the length of the substrings, they are is limited to length M.

Both unlimited search buffers can be limited. The following changes have to be made

- $\delta \oplus \alpha = (\delta +\!\!+ \alpha) \upharpoonright N$
- Limit the substring function, by adding an additional precondition $0 \leq j \leq M$ to the definition of the substring function.

The SubstituteId$\mathcal{E}$ proofs are analog to the proofs of the unlimited versions.

This gives two new search buffers, namely $\mathcal{B}_{lim}$ and $\mathcal{B}_{liml}$.

Chapter 5

Conclusions and directions for further work

5.1 Conclusions

The main result of this thesis is a taxonomy of several Lempel-Ziv variants. In the taxonomy, all the included variants have been based on one general specification, which means that all variants share a common skeleton. The presentation of the variants in a taxonomy has several advantages:

- The accessibility of the included variants improves because all variants have been given in one presentation style with one nomenclature. Originally, all variants were presented in different styles, at different levels of abstraction, and using different terminologies.
- A structured overview of the variants has been given. All differences and similarities between the variants have been indicated.
- Formal proofs have been given for each variant included. These proofs include, for instance, that the algorithms make progress and that the compression is lossless. In the literature, not a single algorithm description has been found that includes formal proofs of the correctness of encoder and decoder.
- Some possibilities for new Lempel-Ziv variants are indicated by the taxonomy. If in one point in the taxonomy a certain solution has been chosen for a problem, and if in another point another solution has been chosen for the same problem, then the solutions can be swapped, leading to two new variants. There are several places in the taxonomy where solutions can be swapped. Because the problems have been identified precisely, it is also possible to create a totally new solution for a problem.
- It reveals that the adjustments to the environment – the base for referents and references – are almost free to define. In some cases there is no requirement for

the adjustment at all. The importance of the adjustment will become clear if the compression ratios are determined. But for the correctness the environments can be altered almost freely.

Although the taxonomy has been presented as a one-dimensional taxonomy in this thesis, it is actually a multi-dimensional taxonomy. It is multi-dimensional because some subproblems can be solved independently. Examples of this are the selection of the prefix that will be processed next, and the method to handle referents and references.

The taxonomy has not been presented as a multidimensional one because there is no actual need to do so. In this limited taxonomy, only one or a few possible solutions have been used for each independently solvable subproblem. The addition of the extra dimension would probably only make it harder to understand the taxonomy. The visual summary of an one-dimensional taxonomy is a two-dimensional figure, which easy to visualize. If more dimensions would be added, then the visual summary would be a figure with more than two dimensions, which is not easy to visualize, or can not be visualized at all.

If the taxonomy will be expanded rigorously, then it can probably not be avoided to add one or more dimensions.

The construction of the taxonomy took much effort. There are several reasons for this:

- The original descriptions of the variants are not always as accessible as one would like. Each variant is presented in its own style with its own terminology. Some are presented in a more text like style, some are presented in some meta-notation. To make it possible to compare – and thus structure – the variants, all the descriptions had to be transformed to a common presentation style. To do so, the articles in which the variants have been described had to be read very thoroughly and the essential information had to be extracted. The presentation-style-transformation took much time, especially for the text like descriptions of LZ77[ZL77] and [ZL78].

  Note that all this work is not visible in the taxonomy.

- The similarities between two algorithms are very hard to identify if an algorithm uses a trick for computation that is not directly recognizable as such. The description of LZW[Wel84] contained such a hard to recognize computation trick.

- If the differences and similarities between two algorithms can be indicated, then it does not automatically result in an elegant generalization. A generalization removes some details; consequently, the generalized version ought to look simpler. Some generalizations that were made looked very complicated, even more complicated than the non-generalized algorithms. Such a ”generalized” version was actually more a unification.

A method that makes it possible to refine an incomplete algorithm has been constructed. This method with its notation evolved from nothing to the current version. It enables the refinement of an algorithm in several steps. This is useful because this is precisely what is

done in a taxonomy. The notation makes it possible to read successive specifications of an incomplete algorithm independently. All parts of the specification – the declarations, definitions, assumptions, lemmas and algorithms – have been given in each specification. This prevents for example that the reader has to search backwards in the text for a declaration of a certain function.

5.2 Directions for further work

Although a taxonomy has been presented, a lot of further work can be done.

- Making a toolkit – also called class library – is a next step after constructing a taxonomy[Wat95]. It is an implementation of the algorithms derived in the taxonomy. Besides writing the toolkit, it also has to be benchmarked. The benchmarks will have to evaluate the compression ratios and the time complexity. It has to be examined if the implementations in the toolkit will perform as good as already existing separately developed implementations, which have been hand-fine-tuned in most cases.
- Several Lempel-Ziv variants have been included now, but there are a lot more variants:
  - There are several other well known variants, all with their own acronym, see for example appendix A.
  - In the literature, several ideas have been described that may lead to new variants. For example, for some of the "greedy" variants a non-greedy variant could also be imagined.
  - The taxonomy indicates several new variants. These variants also have to be included in the taxonomy. Some variations are:
    - LZSS with a dictionary
    - Use a new kind of search buffer in the subgroup $\mathcal{K} = \mathcal{R}$. To assure progress, the actual buffer part of this new search buffer has to be prefixed with a string containing all symbols of the alphabet.
    - New types of adjustment can be developed for the environment. From the correctness point of view, the adjustments are in all cases almost totally free to choose. But the compression ratio is greatly influenced by the adjustments. Experimental results have to determine that other adjustments are useful.
- The method with its notation used to refine an incomplete algorithm needs further attention. Take for example the notation, it seems to work quite well for this taxonomy. But it already needs almost one and a half page for the algorithms located almost at the bottom of the taxonomy. If a small thing is added, then the whole specification has to be given again, which is not very practical.

The taxonomy that has been given here is probably be a good basis for the further work.

Bibliography

[BCW90] Timothy C. Bell, John G. Cleary, and Ian H. Witten. *Text Compression*. Prentice-Hall, 1990. ISBN 0-13-911991-4.

Abstract: Contains several data compression methods, such as statistical methods and Lempel-Ziv methods. The Lempel-Ziv algorithms are mostly described in a textual way and each description includes a formal specification of the domain of the so called dictionary.

[Bel86] T.C. Bell. Better OPM/L text compression. *IEEE Transactions on communications*, 34(12):1176–1182, December 1986.

Abstract: Original documentation of the LZSS algorithm.

[Bel87] T.C. Bell. *A Unifying Theory and Improvements for Existing Approaches to Text Compression*. Ph.D. thesis, Department of Computer Science, University of Canterbury, Christchurch, New Zealand, 1987.

[Ber91] Daniel J. Bernstein. Y coding, March 1991. `www-ftp.lip6.fr/pub/unix/archive/yabba.tar.z`.

Abstract: Original documentation of the LZY algorithm.

[Blo96a] C.R. Bloom. LZP: a new data compression algoritm, 1996. `http://www.cbloom.com/papers/lzp.zip`. See also [Blo96b],

Abstract: Original documentation of the LZP algorithms,

Comment: See also [Blo96b].

[Blo96b] C.R. Bloom. Using prediction to improve LZ77 coders, 1996. `http://www.cbloom.com/papers/lzp_old.zip`.

Comment: Form C. Bloom: This version is somewhat more thorough and complete as [Blo96a], however the details in it are out-dated and the presentation is not as thoroughly editted. It is inteded for people who have read the primary paper and want more information.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976. ISBN 0-13-215871-X.

[Ern92] Michael Ernst. Partial list of software patents, November 1992. `http://www.funet.fi/pub/gnu/lpf/patent-list.txt`.

Abstract: Summary of software patents.

[FG89] E.R. Fiala and D.H. Greene. Data compression with finite windows. *Communications of the ACM*, 32(4):490–505, April 1989. ISSN 0001-0782. `http://doi.acm.org/10.1145/63334.63341`.

Abstract: Original documentation of the LZFG algorithm.

[FvG99] W.H.J. Feijen and A.J.M. van Gasteren. *On a method of multiprogramming*. Monographs in computer science. Springer-Verlag, 1999. ISBN 0-387-98870-X.

[vGvdW02] Rik van Geldrop and Jaap van der Woude. Reverse engeneering: Recurrente betrekkingen uit imperatieve programma's. Private communication, August 2002.

[IEE81] IEEE. 1979 information theory group paper award. *IEEE Transactions on Information Theory*, 27(1):4, January 1981.

Abstract: Background information about Jacob Ziv and Abraham Lempel.

[IEE96] IEEE Information Theory Society. Jacob Ziv wins the 1997 Claude E. Shannon Award. *IEEE Information Theory Society Newsletter*, December 1996. `http://www.ieeeits.org/publications/nltr/96_dec/ziv.html`.

Abstract: Background information about Jacob Ziv.

[Jak85] M. Jakobsson. Compression of character strings by an adaptive dictionary. *BIT*, 25(4):593–603, December 1985. ISSN 0006-3835.

[Jen97] Brian Strack Jensen. Lempel-Ziv-Oberhumer Compressed Animation (LZA), 1997. `http://honors.tntech.edu/~will/fileformat.virtualave.net/graphics/lza/lza.txt`.

Abstract: Specification of the LZA file format,

Comment: No real Lempel-Ziv variant.

[Jon82] H.B.M. Jonkers. *Abstraction, specification and implementation techniques*. PhD thesis, Eindhoven University of Technology, department of Mathematics and Computer Science, 1982.

[Mic97] Microsoft Corporation. Microsoft LZX data compression format, March 1997. `www.microsoft.com`. Included in the Cabinet Software Development Kit,

Abstract: Original documentation of the LZX algorithm.

[MW85] V.S. Miller and M.N. Wegman. *Variations on a theme by Ziv and Lempel*, volume 12 of *NATO ASI, series F*, pages 131–140. Springer-Verlag, 1985. ISBN 3-540-15227-x.

Abstract: Original documentation of the LZMW algorithm.

[Nel89] Mark Nelson. LZW data compression. *Dr. Dobb's journal*, October 1989. `http://www.dogma.net/markn/articles/lzw/lzw.htm`.

Abstract: Documentation of the LZW algorithm.

[Obe02] Markus F.X.J. Oberhumer. LZO; a real-time data compression library, July 2002. `http://www.oberhumer.com/opensource/lzo/lzodoc.php`.

Abstract: Introduction to LZO; no real specification,

Comment: Homepage of the LZO algorithms: `http://www.oberhumer.com/opensource/lzo`.

[Oku98] Haruhiko Okumura. History of data compression in Japan, 1998. `http://matsusaku-u.ac.jp/~okumura/compression/history.html`.

Abstract: Small history of LZARI and spin offs.

[RPE81] Michael Rodeh, Vaughan R. Pratt, and Shimon Even. Linear algorithm for data compression via string matching. *Journal of the ACM (JACM)*, 28(1):16–24, January 1981. ISSN 0004-5411. `http://doi.acm.org/10.1145/322234.322237`.

Abstract: Original documentation of the LZR algorithm.

[Sal98] David Salomon. *Data Compression; The complete reference*. Springer-Verlag, 1998. ISBN 0-387-98280-9.

Abstract: Contains several data compression methods, such as statistical methods, Lempel-Ziv methods and methods for image compression. The Lempel-Ziv algorithms are mostly described in a textual way and they are given in a non-uniform format.

[SS78] James A. Storer and Thomas G. Szymanski. The macro model for data compression (extended abstract). In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 30–39, San Diego, California, United States, 1978. Digital available via `portal.acm.org`,

Abstract: Defines several methods of textual substitution,

Comment: Earlier version of [SS82].

[SS82] Jams A. Storer and Thomas G. Szymanski. Data compression via textual substitution. *Journal of the ACM*, 29(4):928–951, October 1982. ISSN 0004-5411. `http://doi.acm.org/10.1145/322344.322346`. Revised version of [SS78],

Abstract: Defines several methods of textual substitution.

[Wat95] B.W. Watson. *Taxonomies and toolkits of regular language expressions*. PhD thesis, Eindhoven University of Technology, department of Mathematics and Computer Science, 1995. ISBN 90-386-0396-7.

[Wel84] T.A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, June 1984.

Abstract: Original documentation of the LZW algorithm.

[Wil91] R.N. Willians. An extremely fast Ziv-Lempel data compression algorithm. In *Data Compression Conferance '91*, Proceedings of Data Compression Conference, pages 362–371. IEEE Computer Society Press, April 1991. Digital available via `www.ieee.org`,

Abstract: Original documentation of the LZRW1 algorithm,

Comment: Homepage of the LZRW algorithms: `http://www.ross.net/compression`.

[ZL76] Jacob Ziv and Abraham Lempel. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, January 1976.

[ZL77] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977. `http://compression.graphicon.ru/download/articles/lz/ziv_lempel_1977_universal_algorithm.pdf`.

Abstract: Original documentation of the LZ77 algorithm.

[ZL78] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, September 1978. `http://einstein.univalle.edu.co/~ctelecom/Information_Theory/Final_Project-2002-I/LZ-78.pdf`.

Abstract: Original documentation of the LZ78 algorithm.

Appendix A

Inventory of Lempel-Ziv variants

Table A.1 on page 84 gives an inventory of all variants found, in alphabetical order. It also includes variants that are not in the taxonomy. The table has several columns, namely:

I : "✓" indicates that this variant is included in the taxonomy

variant : name of the variant

year : year of –probably first– publication
This is included to indicate a general time line.

appear : place where the variant appears, with or without any specification

specification : reference to specification
The first indicates the original specification. This can be [], which indicates that the original specification could not be determined with certainty. Note that it is possible that only the code has been found, without any explanation at all. This is not seen as the original specification.
The next references are additional references which give an alternative specification or references to relevant articles.

remark : remarks about the variant
If the variant is not contained in the taxonomy and if a short description could be determined clearly, then a description is also included.

G : "grade" for the found specification; varying from ++ to −−

Note that the names LZ1 and LZ2 are not uniformly defined, but are document dependant. Take for example LZ1. In some documents this indicates LZ76. In other documents this indicates LZ77.

There are several other inventories [Sal98, BCW90]. These inventories are listings which contain one-line summaries. Each listing is followed with more detailed descriptions, which are mostly textual.

Table A.1: Inventory of Lempel-Ziv variants

I	variant	year	appear	specification	remark	G
	LZ76	1976	[Bel86, RPE81]	[ZL76]	implemented as LZR	+/−
✓	LZ77	1977	[Sal98, BCW90]	[ZL77]	”mother-algorithm” for buffer based variants	+
✓	LZ78	1978	[Sal98, BCW90]	[ZL78]	”mother-algorithm” for dictionary based variants	+
	LZA	1997	[Jen97]	[Jen97]	animation file format; not an Lempel-Ziv variant. LZA stands for ”LZO compressed animation”.	+/−
	LZA		newsgroup comp.compression	[]	Stacker LZA	−−
✓	LZAP	1987	[Sal98]	[] , [Ern92, Ber91]		−
	LZARI	1988	[Oku98]	[] , [Oku98]	LZSS with arithmic coding	−
	LZB	1987	[BCW90]	[] , [BCW90]	LZSS with infinite lookahead; probably original specification in [Bel87]	−
✓	LZC	1985	[BCW90]	[] , [BCW90]		−
	LZCB	1995	`www.cbloom.com`	[] , `www.cbloom.com`	LZCB11-14 are better known as LZP1-4(only these are well documented)	−
	LZF	2000	`liblzf.plan9.de`	[] , `liblzf.plan9.de`	reimplementation of LZV	−−
	LZFG	1989	[Sal98, BCW90]	[FG89]	some resemblance with LZSS; uses references and uncompressed strings	+
	LZH	1987	[Sal98, BCW90]	[] , [Sal98]	LZSS with Huffman coding	−
	LZJ	1985	[BCW90]	[Jak85]		+
	LZJH	2000	`www.lzjh.com`	[] , `www.lzjh.com`	used in V.44 modem protocol. Lempel-Ziv-Jeff-Heath	−−
✓	LZMW	1984	[Sal98, BCW90]	[MW85]		+
	LZO	1996	newsgroup comp.compression	[Obe02]	buffer based	−
	LZP	1996	[Sal98]	[Blo96a] , [Blo96b]	4 variants: LZP1-4; also known as LZCB11-14; buffer based; reference without offset, only lenght	++
✓	LZR	1981	[BCW90]	[RPE81] , [ZL76]		+

Continued on next page

Table A.1: Inventory of Lempel-Ziv variants –continued–

I	variant	year	appear	specification	remark	G
	LZRW	1991	[Sal98] , `www.ross.net/compression`	[] , `www.ross.net/compression`	5 variants: LZRW1-5	+/−
	LZRW-1	1991	[Sal98]	[Wil91]	buffer based; reference or symbol; not necessary the longest prefix	+
	LZS		internet	[]		−−
✓	LZSS	1986	[Sal98, BCW90]	[Bel86] , [SS82]		++
✓	LZT	1987	[BCW90]	[] , [BCW90]		−
	LZV	2000	[Obe02]	[]	Lev-Zimpel-Vogt (miss-spelled!); Herman Vogt	−−
✓	LZW	1984	[Sal98, BCW90]	[Wel84] , [Nel89]		+
	LZX	1997	newsgroup comp.compression	[Mic97]	Used for Windows CAB-files; some resemblance with LZSS with Huffman coding	+
✓	LZY	1991	[Sal98]	[Ber91]		+

Appendix B

List of selected specifications

This appendix lists several specifications that are located in the main text. The reason to give the specifications again is clarity. The specification can be better compared, because each specification will start on a new page and because they are listed successively.

The specifications that are selected are

- the specification at the top, which is the root specification (Specification 4.6 from chapter 4)
- the specifications at the bottom (Specifications 4.33, 4.61, 4.83 from chapter 4)

The listing of the specifications is related to the structure of the taxonomy. First the root is listed. Thereafter the specifications at the bottom are listed. The listing of the bottom specifications starts at the left of the taxonomy, and it ends at the right of the taxonomy.

Each specification will start on the left-hand side of a page. This achieves that both pages of a two-paged specification are visible at one time. One page is visible at the left-hand side, one at the right-hand side.

Specification B.1 (Root)

Declarations:

Input	$\Sigma \in$ Alphabet
Output	$\mathcal{K} \in$ Countable Set
Encode	$\mathbf{encode} : \Sigma^* \to \mathcal{K}^*$
Decode	$\mathbf{decode} : \mathcal{K}^* \nrightarrow \Sigma^*$
ProperPrefix	$h_{pp} : \Sigma^* \times \Sigma^* \to \mathcal{P}.\Sigma^*$
SelectPrefix	$h_{sel} : \mathcal{P}.\Sigma^* \times \Sigma^* \times \Sigma^* \nrightarrow \Sigma^*$
Substitute	$f : \Sigma^* \to \Sigma^* \nrightarrow \mathcal{K}$
DeSubstitute	$g : \Sigma^* \to \mathcal{K} \nrightarrow \Sigma^*$

Definitions:

Assumptions:

SubstituteId	$\langle \forall \sigma, \eta \,:\, \eta \in dom.(f.\sigma) \,:\, (g.\sigma) \circ (f.\sigma) = \mathsf{id} \rangle$
Substitutable	$\langle \forall \sigma, \sigma' \,:\, \sigma' \neq \lambda \,:\, h_{pp}.(\sigma, \sigma') \subseteq dom.(f.\sigma) \rangle$
ProperPrefixExists	$\langle \forall \sigma, \sigma' \,:\, \sigma' \neq \lambda \,:\, h_{pp}.(\sigma, \sigma') \neq \emptyset \rangle$
Progress	$\langle \forall \sigma, \sigma' \,:\, \sigma' \neq \lambda \,:\, \lambda \notin h_{pp}.(\sigma, \sigma') \rangle$
IsPrefix	$\langle \forall \sigma, \sigma', \eta \,:\, \sigma' \neq \lambda \,:\, \eta \in h_{pp}.(\sigma, \sigma') \Rightarrow \eta \preccurlyeq_p \sigma' \rangle$
ProperSelect	$\langle \forall V, \sigma, \sigma' \,:\, \sigma' \neq \lambda \wedge V \in \mathcal{P}.\Sigma^* \,:\, V \neq \emptyset \Rightarrow h_{sel}.(V, \sigma, \sigma') \in V \rangle$

Lemmas:

Lossless	$\mathbf{decode} \circ \mathbf{encode} = \mathsf{id}$	(4.7)

Algorithms:

$\mathbf{encode}.\sigma_0 =$
$[\![\ \mathbf{var}\ \sigma, \sigma' : \Sigma^*;\ \kappa : \mathcal{K}^*$
$|\ \sigma', \kappa, \sigma :\, = \sigma_0, \lambda, \lambda;$
$\quad \mathbf{do}\ \sigma' \neq \lambda \to$
$\qquad \mathbf{let}\ \eta\ \mathbf{such\ that}\ \eta = h_{sel}.(V, \sigma, \sigma')$
$\qquad\qquad \mathbf{where}\ V = h_{pp}.(\sigma, \sigma')\ ;$
$\qquad \kappa :\, = \kappa +\!\!+ \langle f.\sigma.\eta \rangle;$
$\qquad \sigma, \sigma' :\, = \sigma +\!\!+ \eta, \sigma' \downharpoonright |\eta|$
$\quad \mathbf{od};$
$\quad \mathbf{return}\ \kappa$
$]\!]$

$\mathbf{decode}.\kappa_0 =$
$\{\ \text{pre: } \langle \exists \sigma_0 \,:\,\,:\, \kappa_0 = \mathbf{encode}.\sigma_0 \rangle\ \}$
$[\![\ \mathbf{var}\ \sigma : \Sigma^*;\ \kappa' : \mathcal{K}^*$
$|\ \kappa', \sigma :\, = \kappa_0, \lambda;$
$\quad \mathbf{do}\ \kappa' \neq \lambda \to$
$\qquad \mathbf{let}\ \kappa' :: \langle c \rangle +\!\!+ \rho'\ ;$
$\qquad \sigma :\, = \sigma +\!\!+ g.\sigma.c;$
$\qquad \kappa' :\, = \rho'$
$\quad \mathbf{od};$
$\quad \mathbf{return}\ \sigma$
$]\!]$

Specification B.2 (Root with h_{pp}, $\mathcal{E}$, $\mathcal{K} = \mathcal{R}$, ℓr)

Declarations:

Input	$\Sigma \in$ Alphabet
Output	$\mathcal{K} \in$ Countable Set
Reference	$\mathcal{R} \in$ Countable Set
Initial $\mathcal{E}$	$\delta_{init} \in \mathcal{E}$
Encode	$\mathbf{encode} : \Sigma^* \to \mathcal{K}^*$
Decode	$\mathbf{decode} : \mathcal{K}^* \nrightarrow \Sigma^*$
ProperPrefix	$h_{pp} : \Sigma^* \times \Sigma^* \to \mathcal{P}.\Sigma^*$
SelectPrefix	$h_{sel} : \mathcal{P}.\Sigma^* \times \Sigma^* \times \Sigma^* \nrightarrow \Sigma^*$
Substitute	$f : \Sigma^* \to \Sigma^* \nrightarrow \mathcal{K}$
Refer	$f_{\mathcal{R}} : \Sigma^* \to \Sigma^* \nrightarrow \mathcal{R}$
DeSubstitute	$g : \Sigma^* \to \mathcal{K} \nrightarrow \Sigma^*$
DeRefer	$g_{\mathcal{R}} : \Sigma^* \to \mathcal{R} \nrightarrow \Sigma^*$
AdaptEnvironment	$e : \mathcal{E} \times \Sigma^* \to \mathcal{E}$

Definitions:

$$\mathcal{K} = \mathcal{R}$$
$$f = f_{\mathcal{R}}$$
$$g = g_{\mathcal{R}}$$
$$h_{pp}.(\sigma, \sigma') = \{\eta | \eta \in dom.(f.\sigma) \wedge \eta \preccurlyeq_p \sigma' \wedge \eta \neq \lambda\}$$
$$h_{sel}.(V, \sigma, \sigma') = \eta \textbf{ such that } \eta \in V \wedge \langle \forall \alpha : \alpha \in V : |\eta| \geq |\alpha| \rangle$$

Assumptions:

MinDom$f$	$\langle \forall \sigma : \ : \Sigma \subseteq dom.(f.\sigma) \rangle$
SubstituteId$\mathcal{E}$	$\langle \forall \delta, \eta : \eta \in dom.(\delta \triangledown) : (\delta \triangle) \circ (\delta \triangledown) = \mathsf{id} \rangle$

Lemmas:

Lossless	$\mathbf{decode} \circ \mathbf{encode} = \mathsf{id}$	(4.7)
Substitutable	$\langle \forall \sigma, \sigma' : \sigma' \neq \lambda : h_{pp}.(\sigma, \sigma') \subseteq dom.(f.\sigma) \rangle$	(4.13)
ProperPrefixExists	$\langle \forall \sigma, \sigma' : \sigma' \neq \lambda : h_{pp}.(\sigma, \sigma') \neq \emptyset \rangle$	(4.14)
Progress	$\langle \forall \sigma, \sigma' : \sigma' \neq \lambda : \lambda \notin h_{pp}.(\sigma, \sigma') \rangle$	(4.15)
IsPrefix	$\langle \forall \sigma, \sigma', \eta : \sigma' \neq \lambda : \eta \in h_{pp}.(\sigma, \sigma') \Rightarrow \eta \preccurlyeq_p \sigma' \rangle$	(4.16)
SubstituteId$\mathcal{R}$	$\langle \forall \sigma, \eta : \eta \in dom.(f_{\mathcal{R}}.\sigma) : (g_{\mathcal{R}}.\sigma) \circ (f_{\mathcal{R}}.\sigma) = \mathsf{id} \rangle$	(4.22)
SubstituteId	$\langle \forall \sigma, \eta : \eta \in dom.(f.\sigma) : (g.\sigma) \circ (f.\sigma) = \mathsf{id} \rangle$	(4.30)
LongestReferent	$\langle \forall V, \sigma, \sigma', \eta : \sigma' \neq \lambda \wedge V \in \mathcal{P}.\Sigma^* :$ $h_{sel}.(V.\sigma, \sigma') = \eta \Rightarrow \langle \forall \alpha : \alpha \in V : \lvert\eta\rvert \geq \lvert\alpha\rvert \rangle \rangle$	(4.34)
ProperSelect	$\langle \forall V, \sigma, \sigma' : \sigma' \neq \lambda \wedge V \in \mathcal{P}.\Sigma^* :$ $V \neq \emptyset \Rightarrow h_{sel}.(V, \sigma, \sigma') \in V \rangle$	(4.35)

Algorithms:

$\mathbf{encode}.\sigma_0 =$
$\quad [\![\ \mathbf{var}\ \sigma, \sigma' : \Sigma^*;\ \kappa : \mathcal{K}^*;\ \delta : \mathcal{E}$
$\quad |\ \sigma', \kappa, \sigma, \delta := \sigma_0, \lambda, \lambda, \delta_{init};$
$\quad\quad \mathbf{do}\ \sigma' \neq \lambda \rightarrow \{\ \delta \triangledown = f_{\mathcal{R}}.\sigma\ \}$
$\quad\quad\quad \mathbf{let}\ \eta\ \mathbf{such\ that}\ \eta = h_{sel}.(V, \sigma, \sigma')$
$\quad\quad\quad\quad\quad \mathbf{where}\ V = h_{pp}.(\sigma, \sigma')\ ;$
$\quad\quad\quad \kappa := \kappa +\!\!+ \langle f.\sigma.\eta \rangle;$
$\quad\quad\quad \sigma, \sigma', \delta := \sigma +\!\!+ \eta, \sigma' \downharpoonleft |\eta|, e.(\delta, \eta)$
$\quad\quad \mathbf{od};$
$\quad\quad \mathbf{return}\ \kappa$
$\quad]\!]$

$\mathbf{decode}.\kappa_0 =$
$\{\ \text{pre: } \big\langle \exists \sigma_0 :\ : \kappa_0 = \mathbf{encode}.\sigma_0 \big\rangle\ \}$
$\quad [\![\ \mathbf{var}\ \sigma : \Sigma^*;\ \kappa' : \mathcal{K}^*;\ \delta : \mathcal{E}$
$\quad |\ \kappa', \sigma, \delta := \kappa_0, \lambda, \delta_{init};$
$\quad\quad \mathbf{do}\ \kappa' \neq \lambda \rightarrow \{\ \delta \triangle = g_{\mathcal{R}}.\sigma\ \}$
$\quad\quad\quad \mathbf{let}\ \kappa' :: \langle c \rangle +\!\!+ \rho'\ ;$
$\quad\quad\quad \sigma, \delta := \sigma +\!\!+ g.\sigma.c, e.(\delta, g.\sigma.c);$
$\quad\quad\quad \kappa' := \rho'$
$\quad\quad \mathbf{od};$
$\quad\quad \mathbf{return}\ \sigma$
$\quad]\!]$

Specification B.3 (Root with h_{pp}, $\mathcal{E}$, $\mathcal{K} = \mathcal{R} \times \Sigma$, ℓr)

Declarations:

Input	$\Sigma \in$ Alphabet
Output	$\mathcal{K} \in$ Countable Set
Reference	$\mathcal{R} \in$ Countable Set
Initial $\mathcal{E}$	$\delta_{init} \in \mathcal{E}$
Encode	$\mathbf{encode} : \Sigma^* \to \mathcal{K}^*$
Decode	$\mathbf{decode} : \mathcal{K}^* \nrightarrow \Sigma^*$
ProperPrefix	$h_{pp} : \Sigma^* \times \Sigma^* \to \mathcal{P}.\Sigma^*$
SelectPrefix	$h_{sel} : \mathcal{P}.\Sigma^* \times \Sigma^* \times \Sigma^* \nrightarrow \Sigma^*$
Substitute	$f : \Sigma^* \to \Sigma^* \nrightarrow \mathcal{K}$
Refer	$f_{\mathcal{R}} : \Sigma^* \to \Sigma^* \nrightarrow \mathcal{R}$
DeSubstitute	$g : \Sigma^* \to \mathcal{K} \nrightarrow \Sigma^*$
DeRefer	$g_{\mathcal{R}} : \Sigma^* \to \mathcal{R} \nrightarrow \Sigma^*$
AdaptEnvironment	$e : \mathcal{E} \times \Sigma^* \to \mathcal{E}$

Definitions:

$$\mathcal{K} = \mathcal{R} \times \Sigma$$
$$f.\sigma.\eta = f.\sigma.(\eta_1 +\!\!+ \langle a \rangle) = (f_{\mathcal{R}}.\sigma.\eta_1, a)$$
$$g.\sigma.(r, a) = g_{\mathcal{R}}.\sigma.r +\!\!+ \langle a \rangle$$
$$h_{pp}.(\sigma, \sigma') = \{\eta | \eta \in dom.(f.\sigma) \wedge \eta \preccurlyeq_p \sigma' \wedge \eta \neq \lambda\}$$
$$h_{sel}.(V, \sigma, \sigma') = \eta \textbf{ such that } \eta \in V \wedge \big\langle \forall \alpha : \alpha \in V : |\eta| \geq |\alpha| \big\rangle$$

Assumptions:

SubstituteId$\mathcal{E}$	$\big\langle \forall \delta, \eta : \eta \in dom.(\delta \triangledown) : (\delta \triangle) \circ (\delta \triangledown) = \mathsf{id} \big\rangle$
MinDom$f_{\mathcal{R}}$	$\big\langle \forall \sigma : \ : \lambda \in dom.(f_{\mathcal{R}}.\sigma) \big\rangle$

Lemmas:

Lossless	$\mathbf{decode} \circ \mathbf{encode} = \mathsf{id}$	(4.7)
Substitutable	$\big\langle \forall \sigma, \sigma' : \sigma' \neq \lambda : h_{pp}.(\sigma, \sigma') \subseteq dom.(f.\sigma) \big\rangle$	(4.13)
ProperPrefixExists	$\big\langle \forall \sigma, \sigma' : \sigma' \neq \lambda : h_{pp}.(\sigma, \sigma') \neq \emptyset \big\rangle$	(4.14)
Progress	$\big\langle \forall \sigma, \sigma' : \sigma' \neq \lambda : \lambda \notin h_{pp}.(\sigma, \sigma') \big\rangle$	(4.15)
IsPrefix	$\big\langle \forall \sigma, \sigma', \eta : \sigma' \neq \lambda : \eta \in h_{pp}.(\sigma, \sigma') \Rightarrow \eta \preccurlyeq_p \sigma' \big\rangle$	(4.16)
SubstituteId$\mathcal{R}$	$\big\langle \forall \sigma, \eta : \eta \in dom.(f_{\mathcal{R}}.\sigma) : (g_{\mathcal{R}}.\sigma) \circ (f_{\mathcal{R}}.\sigma) = \mathsf{id} \big\rangle$	(4.22)
SubstituteId	$\big\langle \forall \sigma, \eta : \eta \in dom.(f.\sigma) : (g.\sigma) \circ (f.\sigma) = \mathsf{id} \big\rangle$	(4.59)
MinDom$f$	$\big\langle \forall \sigma : \ : \Sigma \subseteq dom.(f.\sigma) \big\rangle$	(4.60)
LongestReferent	$\big\langle \forall V, \sigma, \sigma', \eta : \sigma' \neq \lambda \wedge V \in \mathcal{P}.\Sigma^* :$ $h_{sel}.(V.\sigma, \sigma') = \eta \Rightarrow \big\langle \forall \alpha : \alpha \in V : \lvert\eta\rvert \geq \lvert\alpha\rvert \big\rangle \big\rangle$	(4.34)
ProperSelect	$\big\langle \forall V, \sigma, \sigma' : \sigma' \neq \lambda \wedge V \in \mathcal{P}.\Sigma^* :$ $V \neq \emptyset \Rightarrow h_{sel}.(V, \sigma, \sigma') \in V \big\rangle$	(4.35)

Algorithms:

encode.σ_0 =
$\|[$ **var** $\sigma, \sigma' : \Sigma^*;\ \kappa : \mathcal{K}^*;\ \delta : \mathcal{E}$
$|\ \sigma', \kappa, \sigma, \delta :\ = \sigma_0, \lambda, \lambda, \delta_{init};$
do $\sigma' \neq \lambda \rightarrow \{\ \delta\bigtriangledown = f_{\mathcal{R}}.\sigma\ \}$
  let η **such that** $\eta = h_{sel}.(V, \sigma, \sigma')$
    where $V = h_{pp}.(\sigma, \sigma')$;
  $\kappa :\ = \kappa +\!\!+ \langle f.\sigma.\eta \rangle;$
  $\sigma, \sigma', \delta :\ = \sigma +\!\!+ \eta, \sigma' \downharpoonleft |\eta|, e.(\delta, \eta)$
od;
return κ
$]\|$

decode.κ_0 =
$\{$ pre: $\langle \exists \sigma_0 :\ : \kappa_0 =$ **encode**.$\sigma_0 \rangle\ \}$
$\|[$ **var** $\sigma : \Sigma^*;\ \kappa' : \mathcal{K}^*;\ \delta : \mathcal{E}$
$|\ \kappa', \sigma, \delta :\ = \kappa_0, \lambda, \delta_{init};$
do $\kappa' \neq \lambda \rightarrow \{\ \delta\bigtriangleup = g_{\mathcal{R}}.\sigma\ \}$
  let $\kappa' :: \langle c \rangle +\!\!+ \rho'$;
  $\sigma, \delta :\ = \sigma +\!\!+ g.\sigma.c, e.(\delta, g.\sigma.c);$
  $\kappa' :\ = \rho'$
od;
return σ
$]\|$

Specification B.4 (Root with h_{pp}, $\mathcal{E}$, $\mathcal{K} = \mathcal{R} + \Sigma$, ℓr, $> J$)

Declarations:

Input	$\Sigma \in$ Alphabet
Output	$\mathcal{K} \in$ Countable Set
Reference	$\mathcal{R} \in$ Countable Set
Initial $\mathcal{E}$	$\delta_{init} \in \mathcal{E}$
J	$J \in \mathbb{N}$
Encode	$\mathbf{encode} : \Sigma^* \to \mathcal{K}^*$
Decode	$\mathbf{decode} : \mathcal{K}^* \nrightarrow \Sigma^*$
ProperPrefix	$h_{pp} : \Sigma^* \times \Sigma^* \to \mathcal{P}.\Sigma^*$
SelectPrefix	$h_{sel} : \mathcal{P}.\Sigma^* \times \Sigma^* \times \Sigma^* \nrightarrow \Sigma^*$
$h_{sel_{\mathcal{R}\Sigma}}$	$h_{sel_{\mathcal{R}\Sigma}} : \Sigma^* \times \Sigma^* \times \Sigma^* \times \Sigma^* \to \Sigma^*$
Substitute	$f : \Sigma^* \to \Sigma^* \nrightarrow \mathcal{K}$
Refer	$f_{\mathcal{R}} : \Sigma^* \to \Sigma^* \nrightarrow \mathcal{R}$
DeSubstitute	$g : \Sigma^* \to \mathcal{K} \nrightarrow \Sigma^*$
DeRefer	$g_{\mathcal{R}} : \Sigma^* \to \mathcal{R} \nrightarrow \Sigma^*$
AdaptEnvironment	$e : \mathcal{E} \times \Sigma^* \to \mathcal{E}$

Definitions:

$$\mathcal{K} = \mathcal{R} + \Sigma$$

$$f.\sigma.\eta = \begin{cases} in_1.(f_{\mathcal{R}}.\sigma.\eta) & \text{if } \eta \in dom.(f_{\mathcal{R}}.\sigma) \\ in_2.\eta & \text{if } \eta \in \Sigma \end{cases}$$

$$g.\sigma = (g_{\mathcal{R}}.\sigma)\nabla\mathsf{id}$$

$$h_{pp}.(\sigma, \sigma') = \{\eta | \eta \in dom.(f.\sigma) \wedge \eta \preccurlyeq_p \sigma' \wedge \eta \neq \lambda\}$$

$$h_{sel}.(V, \sigma, \sigma') = h_{sel_{\mathcal{R}\Sigma}}.(\alpha, \beta, \sigma, \sigma') \textbf{ where}$$

$$\alpha \textbf{ such that } \alpha \in V \wedge \langle \forall \gamma : \gamma \in V : |\alpha| \geq |\gamma| \rangle$$

$$\beta \textbf{ such that } \beta \in V \wedge \langle \forall \gamma : \gamma \in V : |\beta| \leq |\gamma| \rangle$$

$$h_{sel_{\mathcal{R}\Sigma}}.(\alpha, \beta, \sigma, \sigma') = \begin{cases} \alpha & \text{if } |\alpha| > J \\ \beta & \text{if } |\alpha| \leq J \end{cases}$$

Assumptions:

SubstituteId$\mathcal{E}$	$\langle \forall \delta, \eta : \eta \in dom.(\delta\triangledown) : (\delta\triangle) \circ (\delta\triangledown) = \mathsf{id} \rangle$

Lemmas:

Lossless	$\mathbf{decode} \circ \mathbf{encode} = \mathsf{id}$	(4.7)
Substitutable	$\langle \forall \sigma, \sigma' : \sigma' \neq \lambda : h_{pp}.(\sigma, \sigma') \subseteq dom.(f.\sigma) \rangle$	(4.13)
ProperPrefixExists	$\langle \forall \sigma, \sigma' : \sigma' \neq \lambda : h_{pp}.(\sigma, \sigma') \neq \emptyset \rangle$	(4.14)
Progress	$\langle \forall \sigma, \sigma' : \sigma' \neq \lambda : \lambda \notin h_{pp}.(\sigma, \sigma') \rangle$	(4.15)
IsPrefix	$\langle \forall \sigma, \sigma', \eta : \sigma' \neq \lambda : \eta \in h_{pp}.(\sigma, \sigma') \Rightarrow \eta \preccurlyeq_p \sigma' \rangle$	(4.16)
SubstituteId$\mathcal{R}$	$\langle \forall \sigma, \eta : \eta \in dom.(f_{\mathcal{R}}.\sigma) : (g_{\mathcal{R}}.\sigma) \circ (f_{\mathcal{R}}.\sigma) = \mathsf{id} \rangle$	(4.22)
SubstituteId	$\langle \forall \sigma, \eta : \eta \in dom.(f.\sigma) : (g.\sigma) \circ (f.\sigma) = \mathsf{id} \rangle$	(4.79)
MinDom$f$	$\langle \forall \sigma : : \Sigma \subseteq dom.(f.\sigma) \rangle$	(4.80)
ProperSelect	$\langle \forall V, \sigma, \sigma' : \sigma' \neq \lambda \wedge V \in \mathcal{P}.\Sigma^* :$ $V \neq \emptyset \Rightarrow h_{sel}.(V, \sigma, \sigma') \in V \rangle$	(4.84)

Algorithms:

$\mathbf{encode}.\sigma_0 =$
$[\![$ **var** $\sigma, \sigma' : \Sigma^*;\ \kappa : \mathcal{K}^*;\ \delta : \mathcal{E}$
$|\ \sigma', \kappa, \sigma, \delta := \sigma_0, \lambda, \lambda, \delta_{init};$
  do $\sigma' \neq \lambda \rightarrow \{ \delta \triangledown = f_{\mathcal{R}}.\sigma \}$
    let η **such that** $\eta = h_{sel}.(V, \sigma, \sigma')$
        where $V = h_{pp}.(\sigma, \sigma')$;
    $\kappa := \kappa +\!\!+ \langle f.\sigma.\eta \rangle;$
    $\sigma, \sigma', \delta := \sigma +\!\!+ \eta, \sigma' \downharpoonleft |\eta|, e.(\delta, \eta)$
  od;
  return κ
$]\!]$

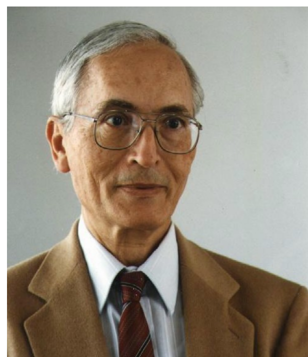
$\mathbf{decode}.\kappa_0 =$
$\{$ pre: $\langle \exists \sigma_0 : : \kappa_0 = \mathbf{encode}.\sigma_0 \rangle \}$
$[\![$ **var** $\sigma : \Sigma^*;\ \kappa' : \mathcal{K}^*;\ \delta : \mathcal{E}$
$|\ \kappa', \sigma, \delta := \kappa_0, \lambda, \delta_{init};$
  do $\kappa' \neq \lambda \rightarrow \{ \delta \triangle = g_{\mathcal{R}}.\sigma \}$
    let $\kappa' :: \langle c \rangle +\!\!+ \rho'$;
    $\sigma, \delta := \sigma +\!\!+ g.\sigma.c, e.(\delta, g.\sigma.c);$
    $\kappa' := \rho'$
  od;
  return σ
$]\!]$

Appendix C

Backgrounds of Ziv and Lempel

This appendix gives a general background of Jacob Ziv and Abraham Lempel. The information is quoted from two articles [IEE81][IEE96]. The articles are out-dated, but they are good enough to give a general background.

C.1 Jacob Ziv



Jacob Ziv was born in Tiberias, Israel, on November 27, 1931. He received the B.Sc., Dipl. Eng., and M.Sc. degrees, all in Electrical Engineering, from the Technion—Israel Institute of Technology, Haifa, Israel, in 1954, 1955 and 1957, respectively, and the D.Sc. degree from the Massachusetts Institute of Technology, Cambridge, U.S.A., in 1962.

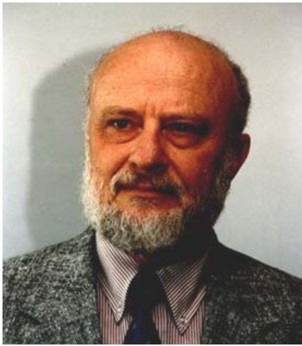
From 1955 to 1959, he was a Senior Research Engineer in the Scientific Department of the Israel Ministry of Defense, and was assigned to the research and development of communication systems. From 1961 to 1962, while studying for his doctorate at M.I.T., he joined the Applied Science Division of Melpar, Inc., Watertown, MA, where he was a Senior Research Engineer doing research in communication theory. In 1962 he returned to the Scientific Department, Israel Ministry of Defense, as Head of the Communications Division and was also an Adjunct of the Faculty of Electrical Engineering, Technion—Israel Institute of Technology. From 1968 to 1970 he was a Member of the Technical Staff of Bell Laboratories, Inc., Murray Hill, NJ. He joined the Technion in 1970 and is Herman Gross Professor of Electrical Engineering.

He was Dean of the Faculty of Electrical Engineering from 1974 to 1976 and Vice President for Academic Affairs from 1978 to 1982. In 1982 he was elected Member of the Israeli Academy of Science, and was appointed as a Technion Distinguished Professor. He is a Fellow of the IEEE. In 1988 he was elected as a Foreign Associate of the US National Academy of Engineering. In 1993 he was awarded the Israel Prize in Exact Sciences (Engineering and Technology). He has twice been the recipient of the IEEE—Information

Theory Best Paper Award (for 1977 and 1979). He is the recipient of the 1995 International Marconi Award and the 1995 IEEE Richard W. Hamming Medal.

From 1977 to 1978, 1983 to 1984, and 1991 to 1992 he was on Sabbatical leave at Bell Laboratories. He has been the Chairman of the Israeli Universities Planning and Grants Committee from 1985 to 1991

C.2 Abraham Lempel



Abraham Lempel was born in Lvov, Poland, on February 10, 1936. He received the B.Sc., M.Sc., and D.Sc. degrees from the TechnionIsrael Institute of Technology, Haifa, Israel, in 1963, 1965, and 1967, respectively.

From 1963 to 1968 he was with the Department of Electrical Engineering, Technion—Israel Institute of Technology. During the academic year 1968-1969 he was a Visiting Research Associate at the University of Southern California, Los Angeles. From 1969 to 1971 he was a Research Staff Member at the Sperry Rand Research Center, Sudbury, MA. In 1971 he rejoined the Technion, where he is currently a Professor in the Department of Computer Science. During the academic year 1975-1976 he was on sabbatical leave with the Department of Mathematical Sciences, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, and during the 1976-1977 academic year he was on leave with the Sperry Research Center as a full-time consultant.