Ludwig–Maximilians–Universität

München


Promotionsarbeit

Christoph-Simon Senjak

Fach:

Informatik

Thema:

# An Implementation of Deflate in Coq

| | |
|---|---|
| Abgabetermin | …2018 |
| Betreuer | Prof. Dr. Martin Hofmann, PhD |

## Eidesstattliche Versicherung

(Siehe Promotionsordnung vom 12.07.11, § 8, Abs. 2 Pkt. .5.)

Hiermit erkläre ich an Eidesstatt, dass die Dissertation von mir selbstständig, ohne unerlaubte Beihilfe angefertigt ist.

---

Name, Vorname

.....

Ort, Datum

.....

Unterschrift Doktorand/in

Formular 3.2

In Memoriam
Prof. Dr. Martin Hofmann

**Abstract**

The compression format "Deflate" is defined in RFC 1951. It is a container format that can utilize prefix-free codings ("Huffman codings"), back-references for deduplication, and run length encoding. Its unverified reference implementation is the ZLib. It is widely used, for example in several network protocols like HTTP or SSH; modern file systems like ZFS and BTRFS even support online compression. Since Deflate is only a container format, there are many possibilities to compress a given data stream, with different tradeoffs. Hence, alternative implementations exist, like the popular Zopfli library from Google.

The standard is quite hard to read, and there are several sources for confusion. We try to remedy this problem by giving a rigorous mathematical specification, which we formalized in Coq. The formalization is axiomatic, and can itself not be verified, which is why we test it empirically against the Canterbury Corpus. Our formalization is furthermore very modular, which makes it easier to test single parts of the implementation, and put them together later. This formalism should be applicable to other data formats in verified implementations in the future.

We produced a verified implementation of a decompression algorithm in Coq which achieves reasonable performance on inputs of several megabytes. To achieve performance, we investigated several efficient functional data structures, and "semi-functional" data structures like DiffArrays, the latter being the ones to perform best.

In this work we present the several parts of our implementation. Our main contribution is a well-tested formalization of the standard. We furthermore produced a fully verified implementation of canonical prefix-free codings, which can be used for other compression formats as well.

We also programmed a compression algorithm in Coq which we formally prove to be inverse to the decompression algorithm – the first time this has been achieved to our knowledge.

We will also talk about the difficulties, specifically regarding memory and runtime requirements, and our approaches to overcome them.

**German Abstract**

Das Kompressionsformat „Deflate“ ist in RFC 1951 spezifiziert. Es ist ein Containerformat, welches präfixfreie Kodierungen („Huffmankodierungen“), Rückreferenzen zur Deduplikation und Lauflängenkodierung unterstützt. Seine unverifizierte Referenzimplementierung ist die ZLib. Das Format ist weit verbreitet, zum Beispiel in diversen Netzwerkprotokollen wie HTTP oder SSH; moderne Dateisysteme wie ZFS und BTRFS unterstützen sogar transparente Kompression. Da es sich nur um ein Containerformat handelt, gibt es viele Möglichkeiten, einen gegebenen Datenstrom zu komprimieren, mit unterschiedlichen Vor- und Nachteilen. Entsprechend gibt es auch alternative Implementierungen, wie das bekannte, von Google implementierte Zopfli.

Der Standard ist schwierig zu lesen, und es gibt etliche Unklarheiten. Wir versuchen, dieses Problem zu beheben, indem wir eine streng mathematische, in Coq formalisierte Spezifikation angeben. Da die Formalisierung axiomatisch ist, und somit selbst nicht verifiziert werden kann, testen wir sie empirisch mit dem Canterbury Corpus. Unsere Formalisierung ist außerdem sehr modular, sodass es einfach ist, einzelne Teile der Implementierung zu testen, bevor sie zusammengefügt werden. Dieser Formalismus sollte sich auch auf andere Datenformate in künftigen verifizierten Implementierungen anwenden lassen.

Wir entwickelten eine verifizierte Implementierung eines Dekompressionsalgorithmus in Coq, welcher bei Eingabedaten von mehreren Megabyte eine angemessene Performanz erzielt. Um Performance zu erreichen, haben wir diverse effiziente funktionale und „semi-funktionale“ Datenstrukturen wie zum Beispiel DiffArrays untersucht; letztere zeigten das beste Verhalten.

In dieser Arbeit stellen wir die verschiedenen Teile unserer Implementierung vor. Das wichtigste Ergebnis ist eine gut getestete Formalisierung des Standards. Wir haben außerdem eine vollverifizierte Implementierung kanonischer präfixfreier Kodierungen, die auch für andere Kompressionsformate benutzt werden kann.

Des Weiteren programmierten wir einen Kompressionsalgorithmus in Coq, von dem wir formal beweisen, dass er invers zum Dekompressionsalgorithmus ist – unseres besten Wisses nach das erste Mal, dass so etwas gemacht wurde.

Wir werden außerdem über die Schwierigkeiten reden, speziell bezüglich Speicher- und Laufzeitverhalten, und über unsere Ansätze um sie zu lösen.

# Contents

# Chapter 1

# Introduction

> Program testing can be used to show the presence of bugs, but never to show their absence!
>
> Edsger Wybe Dijkstra

In this work, we will describe our implementation of the Deflate compression standard in the Coq proof assistant, and the various aspects of the proof and optimization techniques we used. In this chapter, we will give a short introduction to the history of formal methods and compression, as well as our motivation for choosing Deflate, and working on this project.

It is more and more recognized that traditional methods for maintenance of software security reach their limits, and different approaches become inevitable. At the same time, formal program verification has reached a state where it becomes realistic to prove correctness of low-level system components and combine them to prove the correctness of larger systems. This is an important step towards fully verified software, but it is also desirable to verify the low-level middleware. While for these components the adherence of access restrictions would be assured by an underlying sandbox, functional correctness becomes the main concern.

It is desirable to have some guarantees on data integrity, in the sense that the implementation itself will not produce corrupted output. The possibility of faking output could lead to leaks of information like – in a browser setting – passwords or session IDs, which can be hidden inside injected parts of websites, or worse, in the case of software packages like JAR and APK, it could directly inject binary code. Considering the "Raising Lazarus" bug of LZ4 [28], and several ZLIB vulnerabilities [50, 51], such a scenario is realistic.

In Section 1.4.1 we will show some projects that work at this level. We propose to add to this list an implementation of one such middleware, the widely-used compression format Deflate, and analyze the difficulties.

In addition to the aforementioned data corruption bugs, there might also

be situations in which sandboxing is not possible, like embedded devices. Our implementation of Deflate so far may not be suitable for embedded devices, but it can be used as a specification for other algorithms. For example, one could think of an implementation in the Cminor-language which can be compiled by the CompCert compiler [71] and is verified against our specification. To go even deeper, there is the Bedrock system [36], which allows for specification of low-level algorithms in assembly language, and therefore could be used to make a verified implementation suitable for embedded devices. We will have a closer look at these possibilities in section 10.1.4.

A common complaint at this point is that you can get this guarantee by just re-defining your unverified implementations of compression, say $c$, and decompression, say $d$, by

$$\begin{aligned} c'x &= \begin{cases} (\top, cx) & \text{for } d(cx) = x \\ (\bot, x) & \text{otherwise} \end{cases} \\ d'x &= \begin{cases} dy & \text{for } x = (\top, y) \\ y & \text{for } x = (\bot, y) \end{cases} \end{aligned}$$

This works well as long as only one computer architecture is involved, and as long as no bugs are fixed and new features are added. However, for secure long-term-archiving of important data, this is not sufficient: It is not clear that there will be working processors being able to run our $d$ implementation in, say, 50 years; but a formal, mathematical, human-readable specification of the actual data format being used can mitigate against such digital obsolescence: The language of mathematics is universal.

One limitation of all of these approaches is that, of course, one has to rely on *some* specification. Besides having to rely on some hardware specification, as pointed out in [69], finding the right formal specification is not trivial.

A rigorous formal specification of an informally stated standard must be carefully crafted, and we consider our mathematical specification of Deflate as a contribution in this direction.

Parts of this work, specifically Sections 5, 6.1-6.3 and 8.3, have already been published in [89, 90], which was joint work with Martin Hofmann, of which I was the leading author with more than 50% contribution. Small parts of the source code of this work have been discussed in the IRC channel `#coq` in the irc.freenode.net network.

## 1.1 Historical Overview

### 1.1.1 Formal Verification

Formal systems of computability have been of interest at least since it became clear that all of these systems are limited: Wilhelm Ackermann showed

in 1928 that there are effectively computable functions which cannot be expressed as primitive recursive functions [23]. This paper already talks about "types" of functions, but only in a very rudimentary way, which would today probably be called the rank of the type of a function: Functions $\mathbb{N} \to \mathbb{N}$ are called "type 1", functions with type 1 arguments or results, like $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$, are called "type 2", etc. In 1931, Kurt Gödel proved that the decision problem for sufficiently strong formal systems is not generally solvable [53]. Alan Turing famously proved in 1937 that the halting problem is not solvable by a recursive function [95]. In a further famous paper from 1949, he gives a correctness proof of a procedure that computes factorials, which he does to point out that it is easier to prove correctness of programs by splitting it to smaller pieces and pointing out their invariants, which is essentially proving correctness using pre- and postconditions [96]. The notion of recursive functions were usually in terms of primitive recursion and the $\mu$-operator. While being turing complete, this widened the gap to real machines, which is why John McCarthy introduced a formalism with conditionals [74], being closer to if-then-else instructions of contemporary programming languages, and closer to structural recursion operators.

An early version of a computational interpretation of mathematics was given 1958 by Kurt Gödel's Dialectica interpretation [54], see Section 3.4. In 1969, William Howard presented a theory which would later become known as the Curry-Howard correspondence [60], interpreting mathematical formulae as types, and derivations in intuitionistic natural deduction as lambda terms. Related to this development is Dana Scott's logic of computable functionals from the same year, which was first presented in an unpublished manuscript in 1969 [87]. Later, in 1982, Per Martin-Löf introduces type theory, and points out the parallels between mathematics and programming in [73]. These papers form the basis of program extraction, and much of Section 3.2 is about this topic.

In 1967, Robert Floyd introduced the notion of a "flowchart", which is a directed graph with instructions on its nodes, and of an "interpretation" of a flowchart, which adds assertions to its edges [48]. He defines a "verification" of such an interpretation as a proof of the postconditions from the preconditions and the instructions. This approach is extended by Charles Hoare in 1969 [57] to a fully axiomatic theory of imperative programming. The idea of pre- and postconditions is still the basis of most systems for formal verification of imperative programs, a more modern example being the Ynot library [22], which adds pre- and postconditions to state monads.

### 1.1.2 Data Compression

Mechanical communication using signals dates back to at least 1837 [15], when the Morse code was invented. The lengths of the letters is selected according to the frequency in which they occur in the english language,

which is the basic idea of entropy coding. Probably the first binary coding is given by Louis Braille in 1825 [3] , who invented a system for encoding letters into patterns of up to 6 dots, which enables blind people to read by feeling them. In the context of Braille, there is also an entropy coding, the German *Kurzschrift* ("short script") from 1904 [2], which replaces several letters with syllables. For example, there is a special sign for "ch" (⠡), and as "C" (⠨) almost never stands alone in German, the meaning of the letter "C" is changed to mean "en" – a syllable which is frequent in the German language – while to write a single "C", one has to apply a prefix ( ⠲⠨). In 1949, Robert Fano presented Shannon-Fano-Codings in [46], which are a predecessor of Huffman codings, but not always optimal, though. He also introduces a notion of "bits", which he calls a binary "unit of information". In 1952, David Huffman presented an improved method of generating such minimum redundancy codes [61], the Huffman codings, with which we will deal, which are used in Deflate. Besides improving the coding of information, data often contains repetitions. Abraham Lempel and Jacob Ziv invented a simple algorithm in 1977 [104], the famous LZ77, which is able to deduplicate texts using a "sliding window". This algorithm has seen several refinements since then, for example, the Lempel-Ziv-Storer-Szymanski-Algorithm from 1982, see [91], which can be used in Deflate. The newer BZip2 format, which came after Deflate, can utilize Burrows-Wheeler-Transformation [34]. Deflate as such was introduced by Phil Katz in his implementation of pkZip in 1993, and standardized as an RFC in 1999. Today, even though there are better compression algorithms, the Deflate format is widely used, and many newer formats, for example PNG and HTTP, have support for it. Its reference implementation is the ZLib [84], but there are several other implementations with different tradeoffs, for example the Zopfli [56] library from Google.

## 1.2 Notation

Regarding the exposition of this work, a challenge is that we are working with more than one language. Besides the mathematical notion, there are at least Coq, in which the project is written, and Haskell, to which the code is compiled, which have to be considered. It is not always possible to stick with one notation, especially when we have to switch between abstraction levels. Therefore, we at least try to maintain continuity inside each abstraction level. We furthermore use different monospace fonts for `Coq` and `Haskell`. Programming language examples which are imperative will be in Java, and we will use `this font` for them.

Type annotations in Coq are made with a colon `a : t`; in Haskell, they are made with two colons `a :: t`. For the consing and pattern matching on lists, it is the other way around, `x :: xs` and `x : xs`. In mathematical

notation, we will usually use superscript notation $a^t$, but in certain typographic situations, the Coq-like notation $a : t$ will be used. In all of these languages, type annotations are mostly optional when it is clear which type a term has. For list consing, we use :: in mathematical notation.

Coq uses the distinction between `Set` and `Prop` to realize computational irrelevance. We will use Prop to denote the `Prop` type in mathematical notation. Coq therefore has multiple concepts of existence. The usual `exists x, A` is computationally irrelevant. Mathematically, we will denote it as usual as $\exists_x A$. However, there are also two existential quantifications with computationally relevant eigenvariables, namely `{ x : A & Q }`, where `Q` may itself be computationally relevant, and `{ x : A | Q }`, where `Q` is computationally irrelevant. Though the notation is obviously inspired by the mathematical notion of set comprehension, we will not use this notation. We will use the more common notation $\Sigma_{x^A} Q$ for both, as it will always be clear whether $Q$ should be computational relevant.

In Haskell, the type of lists with elements of type `a` is denoted by `[a]`. Similarly, in mathematical notation, we will write $[a]$. In Coq, it is denoted by `list a`. `[]`, `[]` and `[]` denote the empty list, respectively, and explicit lists are denoted by $[1, 2, \ldots]$, `[1,2,...]` and `[1;2;...]`. We may call the empty list "nil". Notice that the two notations are overloaded, that is, $[a]$ can mean a type or a list with one element, depending on $a$, but this will always be clear. To talk about the $n$-th element of a list, Haskell has the `(!!)` operator, and we will use $a \mathbin{!!} n$ in mathematical notation as well. There is no directly corresponding function in Coq, though there are `nth : (forall A : Type, nat -> list A -> A -> A)`, which returns a default value if out of bounds, and `nth_error : (forall A : Type, list A -> nat -> option A)`. In mathematical notation, we will write $a^+$ for the lists of elements of type $a$ which are not $[]$. Furthermore, we will write $[t]^n$ for the list $[\underbrace{t, \ldots, t}_{n\times}]$.

Standard functions like `map`, `drop`, `find`, etc., will be denoted as map, drop, find, etc., in mathematical notation, and we will put the non-list argument in the index, so $\mathrm{map}_f$ translates to `map f`, as we think this improves readability.

In Coq, the unit type is denoted by `unit`, and its only inhabitant is called `tt`. In Haskell, the unit type is called `()`, and its inhabitant is also called `()`. In mathematical notation, we will call the unit type 1, and its inhabitant $()$.

For product types, Coq has several notions, depending on the computational content. Inside `Prop`, there is `A /\ B`, which we will mathematically denote as $A \wedge B$. For the actual pair type, Coq has the notion `A * B`, Haskell denotes it as `(A, B)` and mathematically, we will denote it as $A \times B$. For sum types, analogeous, there is `A \/ B`, which we will mathematically denote as $A \vee B$, and `A + B` for the computationally relevant sum type.

Mathematically, we will also denote it by $A + B$, with constructors inl and inr. In Haskell, we will usually use the `Either` type for it, which has the constructors `Left` and `Right`.

There is the special case of the type $1 + E$. It is called `option` in Coq, and its constructors are called `Some` and `None`. In Haskell, it is called `Maybe`, and has the constructors `Just` and `Nothing`. Mathematically, we will stick with $1 + E$.

We will not distinguish sets and types. In most cases, the sets we talk about are finite or inductively defined. For example, we will call the natural numbers $\mathbb{N}$.

In our examples, we might use the `DiffArray` [10] library. In our implementation, we use `CpdtTactics` [37].

## 1.3 Reasons for Deflate

We started our work with the observation that dependently typed languages fit perfectly for verifying purely functional, high-level algorithms like the one given in our next example below, while low-level algorithms, like the ones used for programming embedded devices or in high-performance computing, are usually not machine-verified, even though these algorithms are usually more complicated to understand.

A very simple but beautiful example of such a high-level algorithm is list reversal. There is a canonical implementation of list reversal, which intuitively does the right thing (though formally, it depends on the ++ function, which makes it more complex):

```
Fixpoint lrev {A} (l : list A) :=
  match l with
  | [] => []
  | (x :: l_) => (lrev l_) ++ [x]
  end.
```

This algorithm is so simple that it needs no explanation and no verification at all. It is fairly a specification of list reversal. However, as concatenation takes linear time, this algorithm takes quadratic time. The following algorithm only uses linear time:

```
Fixpoint lrev2_ {A} (l y : list A) :=
  match l with
  | [] => y
  | (x :: l_) => lrev2_ l_ (x :: y)
  end.

Function lrev2 {A} (l : list A) := lrev2_ l [].
```

However, this algorithm is less obvious. We can, however, easily prove that this is equivalent. The first thing we prove is a lemma that states that

`lrev2_ l m = lrev l ++ m`, by a simple inductive argument. From this follows directly our claim by setting `m = []`.

```
Lemma lrev_lrev_ : forall {A} (l m : list A),
    lrev2_ l m = lrev l ++ m.
Proof.
  intros A.
  induction l as [|a l IHl].
  + auto.
  + intro m.
    simpl.
    rewrite -> IHl.
    rewrite <- app_assoc.
    reflexivity.
Qed.

Corollary lrev_lrev : forall {A} (l : list A), lrev2 l = lrev l.
Proof.
  intros.
  unfold lrev2.
  rewrite -> lrev_lrev_.
  apply app_nil_r.
Qed.
```

The great advantage of functional purity is that it helps writing algorithms which are "correct by design", so the work of actually verifying them is usually not needed. However, in the presence of tight space and time requirements, it might be inevitable to write low-level code, which is not trivially correct. This holds even in the purely functional realm: Structures like real-time catenable deques [66] are not "obviously" correct anymore.

In the search of something from the "real world", we decided in favor of the Deflate compression standard [41], since it is a widely used standard for lossless general purpose compression, and since so many other formats refer to it: HTTP can make use of it through an HTTP Content-Encoding [47], so does ZIP and derived formats (APK, JAR). Source-code-tarballs are often compressed with GZip [42], which is a container around a Deflate stream [42]. Zlib [43] is another such container format. Both GZip and Zlib are often confused with Deflate. TLS supports Deflate compression [59], even though it is considered deprecated due to the BREACH family of exploits [67]. The filesystem ZFS allows to select GZip as compression format. GZip allows programs to "squeeze out" every bit of memory. Besides that, it uses some nice properties from coding theory, which, to our best knowledge, have not been formalized yet, and which are used by other compression formats like BZip2, too. It was therefore an ideal candidate for a case study on low-level formal verification.

## 1.4 Related Work

In this chapter we want to give an overview of the current state of the art of formal program verification, and relate the several projects to our project. We can roughly separate these in projects that use a similar methodology, and projects that have a similar goal.

### 1.4.1 Similar Goals

Our original goal was to verify higher-level middleware, and we considered compression to be an important part of it, and could not find any implementation of some realistic compression- and decompression-format yet. Therefore, we chose the Deflate compression standard was worthwile.

From the general topic of data compression, there is a formalization of Shannon's theorems in Coq [24]. While being interesting in general, it is not useful for our specific project. A formalization of Huffman's algorithm can be found in [33] in Isabelle, and in [94] in Coq. As we focussed on decompression rather than compression, we had no use of this in our project so far. Our proof-of-concept compression algorithm only uses backreferences, not codings. Furthermore, as we will point out in Section 5, the codings Deflate requires do not need to be Huffman codings, but they need to satisfy a canonicity condition, and the code lengths are bounded.

An important example of middleware is compiler infrastructure. The CompCert compiler is a realistic compiler infrastructure for a large subset of the C programming language. [71] points out that a compiler bug can invalidate all guarantees obtained by formal methods, which is similar to our concerns about data integrity when the data is compressed. This compiler uses several layers of differently complex intermediate languages, with operational semantics specified on the abstract syntax trees. A correctness property is then derived that guarantees that the final code – the assembly code – is equivalent to the original source code. From this project evolved the Verified Software Toolchain [26], which uses it to be able to verify C algorithms, using a semantic which bases on separation logic. We think that it is possible to use our specification to verify some highly optimized C source for deflate. This is, however, further work.

Furthermore, there is the CakeML [4] compiler, which aims, similarly to CompCert, to become a realistically usable and formally verified subset of ML. It also uses a formal semantic of the underlying processor architecture, and a formal language semantic, and proves that during compilation, the semantic is kept. Furthermore, there are efforts to create a verified extraction mechanism to CakeML [62]. Hence, CakeML would also be a candidate for a more optimized implementation of Deflate to be verified against our specification.

In [72], a relational database management system is presented, which is

implemented in Coq. It specifies a semantic on abstract syntax trees of SQL commands, and has a verified query optimizer that recompiles the queries. It is therefore related to compiler infrastructures, but as well to the general topic of data integrity.

The Vellvm project [20, 103] aims to build "a framework for reasoning about programs expressed in LLVM's intermediate representation and transformations that operate on it". Similar to our project, it formalizes an informal specification, implements it, and tests it with real-life examples.

The RockSalt software fault isolator [79] is a sandboxing mechanism that checks executable memory pages before allowing them to be executed, enforcing certain policies. RockSalt is implemented in Coq, and, like CompCert and CakeML, uses a semantic for the X86 instruction set. This kind of security enforcement became popular with Google's Native Client [21]. It can be seen as a special form of virtual machine which realizes security not by just-in-time compilation or processor level privilege separation, but by checking.

The Quark Browser [63] is a web browser which uses "shim verification", that is, a capability-based sandboxing mechanism. With this approach, it can provide security guarantees of unverified components. This makes it safe against attacks on implementations of web standards. On the other hand, of course, there are no guarantees on data integrity.

The method they use is generalized in [83], were the *Reflex* DSL is introduced, with which several components can be isolated, and only the kernel of the browser has to be verified to actually satisfy permissions. We think that this is a reasonable approach to get formal guarantees from unverified code, and a similar approach could be used to reimplement some larger software project one part at a time. However, directly verifying data integrity is the better way on the long run, in our opinion.

Quark uses kernel-level privilege separation. Therefore, the sandboxing mechanism is still part of the trusted codebase. It might be interesting to use something similar to the aforementioned RockSalt project for separation instead.

The ConCon conference management system [64] has a large trusted codebase around a small kernel which assures that no information is leaked, except according to specified capabilities. It generates a RESTful web service adhering a specification in Isabelle, and a web interface around this service. This is a further example of verified middleware: The underlying operating system kernel and even parts of the underlying middleware are part of the trusted codebase.

The FSCQ Filesystem [35] is a filesystem that uses a specification which also considers system crashes, and is hence crash-tolerant – a data integrity property. It is verified in Coq and uses the FUSE virtual filesystem API [17], so it is not part of the kernel, but also middleware. This project is also interesting with respect to the fact that it uses a specification that was no

widely used standard before, and therefore shows that it is possible to start a new protocol using a formal specification from the start.

The MiTLS [32] project implements TLS. Besides correctness and data integrity, cryptographic security properties become the main concern. In [32], an example about "alert fragmentation" is given, which might give an attacker the possibility to change error codes by injection of a byte. This is standard-compliant, but obviously not intended. In this case, as this is a cryptographic protocol, eliminating such specification bugs is more important than adhering to a formal specification.

In the context of cryptography, one should also name CertiCrypt, a framework for cryptographic proofs, which "takes a language-based approach to cryptography: security goals and assumptions are expressed by means of probabilistic programs. In a similar way, adversarial models are specified in terms of complexity classes, e.g. probabilistic polynomial-time programs." [6]

### 1.4.2 Formal Methods in General

At the time of writing, the most well-known project that is fully verified is probably the seL4 [69] kernel. It is an implementation of a microkernel which is – except for small parts – formally verified. It uses Haskell-code, and after refinement, it uses a formal semantic on the binary level, to assure that the compiler is not part of the trusted codebase. Its permission-system is based on *capabilities* through a capability distribution language *capDL*. As it is a microkernel, drivers being processes do not belong to the verified core but run in sandboxes. Thus, the amount of work to get a fully verified system is smaller. Besides being much larger than our project, it has focus on the kernel-level rather than on user-level middleware, and it uses an entirely different approach than we do.

Another approach of verifying software components is automatic software analysis. The Linux kernel has been analyzed by several model checkers [102], and several bugs have been found. Components of Windows have been analyzed, and in [52], it is argued that formal verification is costy and only gives minor advantages over modern program analysis: The SAGE system made it vitally impossible to find buffer overflows, and it is argued that "if nobody can find bugs in P, P is observationally equiv to 'verified'". While we also think that program analysis is absolutely worthwile, we do not really agree with this standpoint. We consider dynamic analysis as more a way of software testing than a way of software verification, and, as Dijkstra said in [101], "Program testing can be used to show the presence of bugs, but never to show their absence!".

Furthermore, there is more to software correctness than security against explicit attacks. While static and dynamic analysis is a good way of ensuring that old code does not contain certain types of vulnerabilities, for

a compression algorithm like in our case, it is desirable to get the formal property that decompression after compression yields the original data, no matter what that data was. Also notice that software which has been fully formally verified is usually also fully specified, and therefore documented.

### 1.4.3 Similar Methodology

We are trying to use program extraction as our main programming technique, in the sense that we prove properties and existences mathematically, and then extract a program which can compute them. This approach is somewhat unusual to directly produce application software, rather than verifying other tools.

For example, in [70], one such tool, a DPLL-based SAT-Solver, is extracted by "[proving] a theorem that just states that each formula in CNF is either unsatisfiable or has a model, and [synthesising] the program from the proof": It proves that a certain given calculus can derive pairs of valuations and formulae in conjunctive normal form

$$(\Gamma^{\text{Variables}\to\text{Maybe Bool}}, \Delta)$$

where $\Delta$ is a set of subsets of the set $\text{Variables} \cup \{\neg v \mid v \in \text{Variables}\}$, and to be read as the logical formula $\bigwedge_{C\in\Delta} \bigvee_{g\in C} g$, which is false after being assigned the truth values from an arbitrary function $\tilde{\Gamma}^{\text{Variables}\to\text{Bool}}$ that agrees with $\Gamma$, meaning $\Gamma(g) = \text{Just } b$ implies $\tilde{\Gamma}(g) = b$. From the completeness of this calculus follows an algorithm which decides such pairs, and from such a decision procedure, we get a SAT-solver, by checking whether $(\lambda_x \text{Nothing}, \Delta)$ is derivable for some formula $\Delta$. In our diagram in Section 3, this would be reconstruction, translation, pruning and compilation; Minlog was used in this project. They also give some information on efficiency considerations, which are, however, not applicable to our case.

The case study [81], albeit on a different topic (Myhill-Nerode), is an interesting source of inspiration in that it distills general principles for improving efficiency of extracted programs which we have integrated where applicable. We will discuss this in further detail in Section 3.

Our theory of getting parsers from constructive proofs from Section 6 follows an idea similar to [30], trying to produce parsers directly from proofs, as opposed to other approaches, for example [40], which defines a formal semantic on parser combinators. Most of the algorithms involved in parsing are short, and therefore, we directly use program extraction for the largest part, and we think it is a feasible way of producing parsers for complicated grammars.

In [45], a verified LTL-Solver is shown, which was verified internally inside a theory in Isabelle, and then extracted to ML. In our diagram in Section 3, the extraction part would be "compilation", but notice that the

reason for this is that this project focuses on refinement of programs, and keeps the formal, computationally irrelevant parts apart from the program in advance, and therefore avoids the "pruning" part. While we preferred the former method of program extraction, for the other parts, this method is comparable to ours in the sense that we also sometimes implement code in the Coq language and verify properties separately; however, while at some positions we use refinement, we do not use a sophisticated framework for it, as this is not the focus of our work.

Notice that DPLL- and LTL-solvers are rather a tool for automatic verification than actual middleware for application software like our Deflate implementation.

# Chapter 2

# Technical Overview

## 2.1 Design Decisions

Our first approach was to implement Deflate in Haskell, and then try to use the information we gained to give a verified implementation in Agda [1]. However, at that time, Agda was missing a working library of rational numbers with addition (which is useful in the realm of canonical Huffman codings), and it did not have many proof strategies. While programming this would have been possible, it was not our chosen topic. Similarly, Idris [13] would have been a language of choice, since it aims for practical usability, but it was not ready for our project. Therefore, we decided to switch to Coq. Coq is an *LCF-Style* proof checker, meaning that it has a very small kernel written in OCaml, through which all proofs are checked, and an extensive library of proof strategies. It furthermore allows for program extraction, which we used to test our specifications empirically: While it was clear that we specify *some* compression format, it was not clear that it actually is what the world understands by Deflate. The Deflate standard [41] is informally specified, and it is not impossible that our formal specification has errors: In fact, during our work, there occured some mistakes, partially because of the strange nature of the standard in some places; for examples, see Section 7.1. However, this assured us even more that a fully formal specification of the standard, against which new implementations could check, is useful. There has been lots of development in program verification and type theory since we started with our project, meaning that some parts might not be up to date with respect to the techniques we used; however, our main contribution is not the implementation itself, but the specification, which should be easily portable to other languages.

Of course, the ZLib [84] is a reference implementation of Deflate, and we could just have taken its source, and verify it against our specification. This way, we would probably have been able to find bugs in both our relation as well as the ZLib. However, the ZLib itself is written in highly optimized

C, and verification of existing and highly optimized C code is still a field of research on its own (see [26]). While we certainly think that formally verifying the original ZLib is an interesting and worthwile topic, we decided to create our own new implementation of the standard, the consequence being that we have to test it against real-world data. Program extraction from simple proofs yields algorithms which can be of bad performance, but are usually sufficient for testing against small datasets as a plausibility check.

This was one reason for separating the specification from the implementation: In theory, just programming a decompression algorithm and verifying that it is inverse to some compression algorithm would be sufficient, but we wanted to be able to give differrent implementations of the standard, combine them, test them and maintain them. Another reason was that the Deflate format is itself not entirely strict: It leaves the details of how the data is compressed to the implementation (though the standard gives suggestions). This way, it is possible to tune an implementation for speed or saved memory. The Zopfli library [56] for example tunes for better compression at the cost of runtime. Having a relational specification which is sufficiently easy to understand (and should also be comparably easy to "port" to other proof checkers) seemed to be the better alternative.

We decided to give high-level proofs that involve lots of tactics, rather than giving proof terms, whenever this was possible. However, we structured and annotated our proofs, and in the later code we used Coq's tactic combination language. One reason is that Coq itself is under active development, and tactics might change their behavior. Another reason lies in the improbable case of a specification bug. A clear presentation of the proofs lowers the impact of such changes, and makes it easier to adapt proofs when necessary.

Our concepts of *strong decidability* and *strong uniqueness*, which we will introduce in Section 6, allow to replace parts of the implementation, and makes our specification and implementation very modular. This way, we could test parts of the specification without needing to use all the other parts. For optimization, we follow a top-down approach, replacing the current bottlenecks first. Coq's `Extract Constant` mechanism furthermore allows to replace parts of the algorithm with unverified code, therefore allowing us to test the efficiency of complicated algorithms prior to verifying them. We do this, for example, in our benchmark for our still unverified backreference algorithm in Section 9.3.

However, to this point, we are not trying to compete with the speed of the ZLib. While we give some digressions on how to improve efficiency, our goal is an implementation with *reasonable* space and runtime requirements. While our first implementations took weeks for several KiB, now we are at the magnitude of seconds for several MiB, and we believe that it is possible to continue this trend. However, the memory consumption is a more recent problem, which we address, for example, in Section 6.4.

## 2.2 Trusted Codebase

Coq allows for extraction to OCaml, Haskell and Scheme. The extraction mechanism itself is not verified. For example, we found a bug in the extraction mechanism during our studies, see [9]. The workaround required rewriting the `array_set` function in `Combi.v`. There are efforts to create a verified extraction mechanism [62], but currently, no such mechanism is available, and therefore the extraction mechanism has to be part of our trusted codebase. We began using extraction to Haskell, because it allows doing lazy I/O transparently, adding the GHC to our trusted codebase. We use singly-linked lists of bytes and even bits as intermediate values, as these are well-understood and easy. However, in absence of lazy evaluation, this results in having all the intermediate values fully loaded; lists are usually represented as cons cells of two machine words, and boxed booleans are represented as machine words – this means up to 1536 times the size of the original file. It would have been possible to use coinductive lists ("streams"), but strong decidability (which we define in Section 6) would not hold anymore on these, which we prove in Section 6.2. Our code relies on lazy evaluation, and it does lazy I/O, even though this is discouraged by the Haskell community [18]. Changing this would require a lot of efforts, which, for now, we leave as future work.

It is important to make sure to know the trusted codebase of verified software, since it is axiomatic, and a bug in it can invalidate all proven guarantees. In [49], such a case where programmers work with wrong assumptions is analyzed. We do not make any assumptions about the underlying operating system itself, but work at a higher level of abstraction, which seems more suitable for an implementation of a data format.

We will have a closer look at the trusted codebase and how it can be made smaller in Section 10.1.3.

## 2.3 Module Overview

This is a brief overview over the modules our project consists of. Some concepts will be introduced later, but we mention them here for later reference.

- `Backreferences.v`: Lemmata and definitions regarding the resolution of backreferences.
- `Combi.v`: Several lemmata and functions that are mostly combinatorial and are used multiple times throughout the project.
- `Compress.v`: An implementation of a simple compression algorithm.
- `DecompressWithPheap.v`: An unfinished implementation of our algorithm from Section 8.5. Finishing it is future work.

- `DeflateCoding.v`: Definition of deflate codings, proof of uniqueness and existence of codings for a sequence of code lengths.
- `DiffStack.v`: An implementation of the algorithm described in Section 8.4.
- `EncodingRelation.v`: The most of the encoding relation is specified here.
- `EncodingRelationProperties.v`: Proofs of strong decidability and strong uniqueness of the relations from `EncodingRelation.v`.
- `EncodingRelationProperties2.v`: Definition of streamable strong decidability, and some proofs.
- `ExpList.v`: Decompression using a queue of doom of two ExpLists for backreference resolution, see Section 8.3.
- `Extraction.v`: Definitions of constants for extraction.
- `HashTable.v`: An implementation of a hash table, using an `FMapAVL`. Mainly used in `Compress.v`.
- `Intervals.v`: Helper functions for the compression.
- `KraftList.v`: Kraft's inequality for sets (duplicate-free lists).
- `KraftVec.v`: Kraft's inequality for vectors and codings.
- `Lex.v`: Lemmata and functions about the lexicographical ordering.
- `LSB.v`: Several lemmata for converting numbers in least-significant-bit-first format into natural numbers, converting bytes to `Fin 256` and vice versa, proving several lemmata about them (uniqueness, etc).
- `NoRBR.v`: Extraction of the algorithm that decompresses, but does not resolve backreferences.
- `Pheap.v`: Verified implementation of a pairing heap.
- `Prefix.v`: Lemmata and functions about the prefix relation.
- `Quicksort.v`: An implementation of quicksort.
- `Repeat.v`: Lemmata and functions about repetitive lists.
- `Shorthand.v`: Lots of small definitions that make writing Coq code easier. For example, `Notation Vnth := Vector.nth`.
- `StrongDec.v`: Definitions of strong uniqueness and decidability, several lemmata.

- `Transports.v`: In many cases, it is necessary to transport types into other types. For example, `Definition vec_id A a b (eq : a = b) : vec A a -> vec A b`. Such definitions and lemmata are in this module.

# Chapter 3

# Program Extraction

We dedicate this section to a general introduction to and an overview of program extraction from proofs. For the largest part, it is not Coq-specific, and we will not provide a full formalization. For a complete specification of the calculus of inductive constructions, on which the Coq proof assistant bases, we refer to [93].

## 3.1 Motivation

As a motivation, let us look at a very simple proof:

**Lemma 1.** *For every $n \in \mathbb{N}$, there either exists an $m \in \mathbb{N}$ such that $n = m + m$ or an $m \in \mathbb{N}$ such that $n = m + m + 1$.*

*Proof.* We now prove this by induction.

- For $n = 0$ we have $m = 0$.
- For $n = n' + 1$, there are two cases:
  - Assume $n' = m' + m'$ for some $m'$. Then $n = n' + 1 = m' + m' + 1$, so $m = m'$.
  - Assume $n' = m' + m' + 1$. Then
    $$n = m' + m' + 1 + 1 = (m' + 1) + (m' + 1)$$
    and so $m = m' + 1$. □

This proof intuitively translates into the following algorithm:

```
div2 :: Integer -> Either Integer Integer
div2 0 = Left 0
div2 n = case div2 (n - 1) of
           Left m' -> Right m'
           Right m' -> Left $ m' + 1
```

The objective is to give this "intuition" a formal background. We will use this as a motivating example in the further sections.

## 3.2 Formalizing

The calculus of inductive constructions, with which Coq works, is a sophisticated calculus with scalability in mind. As the goal of this section is rather to give an idea of what program extraction is about, it is out of scope here, and we will only look at a few simple rules here, which resemble the calculus of natural deduction. A detailed introduction to the calculus of inductive constructions can be found in [93].

One central rule in mathematics is usually called *modus ponens*: If we have a proof for $A \to B$, and a proof for $A$, we also know $B$. In some calculi, this can be derived, but mostly, this is a basic rule itself, called *implication elimination*. Similarly, when having a term $a^{A\to B}$, and a term $b^A$, by application, we get a term $(ab)^B$.

$$\frac{A \to B \qquad A}{B} \qquad\qquad (a^{A\to B} b^A)^B$$

The opposite rule is *implication introduction*: If we have a proof of $B$ which uses $A$ as an assumption (which we give a name, say $u$), this is a proof of $A \to B$. The computational counterpart is *abstraction*: If we have a term $M^B$ containing a free variable $u^A$, we may create $(\lambda_u M)^{A\to B}$.

$$\frac{\begin{array}{c}[u : A] \\ \vdots \\ B\end{array}}{A \to B} \qquad\qquad (\lambda_{u^A} M^B)^{A\to B}$$

A rule similar to modus ponens is *forall-elimination*: If we have a proof of $\forall_{x^A} B$, and an object $o$ of type $A$, then we get $B[x := a]$. There is no computational counterpart in the simply typed $\lambda$-calculus anymore, the computational counterpart is *dependently typed*: We have a term $M^{\forall_{x^A} B}$ and then apply an object $o^A$, we get $(Mo)^{B[x:=o]}$.

$$\frac{\forall_{x^A} B \qquad o^A}{B[x := o]} \qquad\qquad (M^{\forall_x B} o)^{B[x:=o]}$$

Finally, *forall-introduction* says that if we can prove $B$ without any assumption regarding $x^A$, then we can prove $\forall_x B$. Computationally, this is again *abstraction*, but with a dependent type: $(\lambda_{x^A} M^B)^{\forall_x B}$.

$$\frac{\begin{array}{c}\vdots \\ B\end{array}}{\forall_{x^A} B} \qquad\qquad (\lambda_{x^A} M^B)^{\forall_x B}$$

These rules are already the basic rules of natural deduction. Type theory adds more intricate rules, but these rules are enough to transport the basic idea.

We could add rules for $\wedge$, $\vee$ and $\Sigma$, but we can as well give *axioms* for their introduction and elimination.

- $\vee_{+B} : A \to A \vee B$, $\vee_{A+} : B \to A \vee B$
  $\vee_{-} : A \vee B \to (A \to C) \to (B \to C) \to C$
- $\wedge_{+} : A \to B \to A \wedge B$
  $\wedge_{-} : A \wedge B \to (A \to B \to C) \to C$
- $\Sigma_{+} : \forall_x (A \to \Sigma_x A)$
  $\Sigma_{-} : \Sigma_x A \to (\forall_x A \to B) \to B$ (where $x$ does not occur freely in $B$)

The terms for $\wedge$ and $\vee$ can be interpreted as the constructors and recursion operators of the product and sum types (pairs and tagged unions), respectively. The interpretation of $\Sigma$ is a dependent product, where the first component is the object, and the second component is a proof of the proposition about this object. This duality of mathematical proofs and algorithms is often called *Curry-Howard isomorphism*.

Finally, to formalize our proof, we need rules for natural numbers $\mathbb{N}$. We have two axiomatic terms $0^{\mathbb{N}}$ and $S^{\mathbb{N}\to\mathbb{N}}$, and for induction we have the term

$$\mathrm{ind}_P^{P0\to\forall_m(Pm\to P(Sm))\to\forall_n Pn}$$

The induction term can be interpreted as recursion operator on natural numbers.

We can now formalize the proof. We leave out annotations where they are clear. The proposition we want to prove is

$$\forall_n.\Sigma_m(n = m + m) \vee \Sigma_m(n = m + m + 1)$$

and our proof will look like

$$\mathrm{ind}_{\lambda_n Q(n)}\, c_1^{Q(0)} c_2^{\forall_n.Q(n)\to Q(Sn)}$$

where $Q(n) = \Sigma_m(n = m + m) \vee \Sigma_m(n = m + m + 1)$. We have

$$c_1 = \vee_+ \Sigma_+ 0 \omega^{0=0+0}$$

In the following, the term $\omega(t_1, \ldots, t_k)$ proves an equality that can be calculated directly from the arguments $t_1, \ldots, t_k$. We will not elaborate on this simple kind of proof here, and just accept that it is provable – in Coq, the tactic for solving such simple equations is called `omega`, and uses Presburger Arithmetic.

For $c_2$, we have a case distinction on $Q(n)$:

$$c_2 = \lambda_{n^{\mathbb{N}}} \lambda_{q^{Q(n)}}\, \vee_- q d_1^{\Sigma_m(n=m+m)\to Q(n+1)} d_2^{\Sigma_m(n=m+m+1)\to Q(n+1)}$$

where

$$d_1 = \Sigma_-(\lambda_{m^{\mathbb{N}}}\lambda_{t^{n=m+m}} \vee_+ \Sigma_+ m(\omega(t))^{n+1=m+m+1})$$

and similarly

$$d_2 = \Sigma_-(\lambda_{m^{\mathbb{N}}}\lambda_{t^{n=m+m+1}} \vee_+ \Sigma_+ (m+1)(\omega(t))^{n+1=(m+1)+(m+1)})$$

So we have

$$\begin{array}{l}\text{ind} \\ \quad \vee_+ \Sigma_+ 0(\omega())^{0=0+0} \\ \quad (\lambda_n \lambda_q.\ \vee_-\ q \\ \qquad\qquad\qquad\qquad \Sigma_-(\lambda_m \lambda_t \vee_+ \Sigma_+ m(\omega(t)) \\ \qquad\qquad\qquad\qquad \Sigma_-(\lambda_m \lambda_t \vee_+ \Sigma_+ (m+1)(\omega(t)))))\end{array}$$

which resembles the above algorithm. We can formalize the proof in Coq, too:

```
Lemma div2 : forall (n : nat),
    {m | n = m + m} + {m | n = m + m + 1}.
Proof.
  induction n as [|n' IHn].
  + apply inl.
    exists 0.
    reflexivity.
  + destruct IHn as [[m' M] | [m' M]].
    − apply inr.
      exists m'.
      omega.
    − apply inl.
      exists (m' + 1).
      omega.
Qed.
```

The extracted Haskell-Code then looks like

```
div2 :: Nat -> Sum Nat Nat
div2 n =
  nat_rec (Inl O) (\_ iHn ->
    case iHn of {
     Inl s -> Inr s;
     Inr s -> Inl (add s (S O))}) n
```

Coq defines natural numbers in terms of `S` and `O`, if we do not override this manually. Coq furthermore removes the dependent pairs: The result of the extracted term is a `Sum Nat Nat`, the second component of the dependent pairs lies in the `Prop` universe, which is *computationally irrelevant*: It encodes invariants that we prove, but for the computation itself, it is not necessary, and hence erased. Instead of leaving it out, we can interpret it as the unit type, and we get the following algorithm, which resembles the proof term more directly:

```
div2 :: Integer -> Either (Integer, ()) (Integer, ())
div2 n_ = case n_ of
            0 -> Left (0, ())
            n -> case div2 (n - 1) of
                   Left q ->
                       case q of
                         (m', ()) -> Right (m', ())
                   Right q ->
                       case q of
                       (m', ()) -> Left (m' + 1, ())
```

However, it is clear that this algorithm can be automatically optimized to the above one.

## 3.3 Classical Reasoning

As soon as we introduce classical reasoning, things get more complicated. A standard example which can be found in many places throughout related literature is the following classical proof.

**Lemma 2.** *There exist two irrational numbers $p, q \in \mathbb{R}\backslash\mathbb{Q}$ such that $p^q \in \mathbb{Q}$.*

*Proof.* If $\sqrt{2}^{\sqrt{2}} \in \mathbb{Q}$. Then $p = q = \sqrt{2}$ is the solution. Otherwise we have $(\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^2 = 2 \in \mathbb{Q}$, so $p = \sqrt{2}^{\sqrt{2}}$ and $q = \sqrt{2}$ is the solution. □

We are using *tertium non datur* $t : \sqrt{2}^{\sqrt{2}} \in \mathbb{Q} \vee \sqrt{2}^{\sqrt{2}} \notin \mathbb{Q}$ in this proof. However, we have no realizer for this axiom: In general, it is not decidable whether a real number is rational, since no non-trivial subset of $\mathbb{R}$ is decidable.

However, in many cases, one can use tricks to extract algorithms from classical proofs. We analyzed a few examples in [86]. On the other hand, as long as only the computationally irrelevant parts require tertium non datur, we can in theory use it. In Coq, computationally irrelevant terms are of type `Prop`, and it is possible to add the tertium non datur for these terms. However, for the largest part, our own code is computationally relevant, and whenever classical reasoning would really make things easier, it would also break program extraction. Therefore, we do not use it.

## 3.4 Phases of Extraction

In our personal experience, there is some confusion about the concepts revolving around program extraction, starting with the simple question about what program extraction actually *is*. The problem is that *every* proof resembles *some* sort of algorithm: A witness of the theorem can be computed, as soon as there are realizers of all axioms that were used – it is just that in the case of non-constructive proofs, realizers of axioms might not exist.

A well-known example for an inherently non-constructive theorem is

**Theorem 1.** *Every infinite sequence $a : \mathbb{N} \to \mathbb{N}$ of natural numbers has an infinite non-descending subsequence.*

*Proof.* Assume $\forall_n \exists_m \forall_{k \geq m}. ak \neq n$. Then let $q$ be such that $\forall_n \forall_{k \geq qn}. ak \neq n$. We define $rn := \max\{q0, \ldots, qn\}$. Then, $\forall_n \forall_{k \geq rn} ak > n$. In particular, $a(rn) > n$. Now let $x0 = a0$ and $x(n+1) = a(r(xn))$. Then $x(n+1) = a(r(xn)) > xn$, and hence, $x$ is a strictly increasing subsequence of $a$.

Now assume $\neg \forall_n \exists_m \forall_{k \geq m}. ak \neq n$. That means, classically, $\exists_n \forall_m \exists_{k \geq m}. ak = n$, and henceforth, the constant sequence $\lambda\_.n$ is a non-decreasing subsequence of $a$. □

This proof is strictly non-constructive: Obviously, we cannot compute this subsequence, at least not without further knowledge about the sequence $a$. However, if we have an oracle that tells us that every $n \in \mathbb{N}$ occurs at most up to an index $k$, we can construct an infinitely ascending sequence. If the oracle tells us the opposite, and if it is also kind enough to tell us a counterexample, we can also construct a subsequence. Hence, it describes an algorithm. Therefore, from a certain point of view, talking about "program extraction from proofs" sounds wrong in the sense that every proof *is* a program. However, the usual mathematician will not regard proofs as algorithms, and it is arguable whether an algorithm that requires oracles should be regarded as a "program". We will distinguish programs and proofs in a more intuitive way: A proof is something that validates some lemma or theorem, and a program is something that can compute some desired output from given inputs. Both concepts are two sides of the same medal.

Starting from a proof in natural language, the first thing that usually exists is a formalization of that proof in some language that is designed to be easy to cope with. For example, Mizar's goal is to make machine proofs readable and writable in a manner that is close to natural language. It uses a very strong theory, such that Mizar proofs are not the easiest starting point for extracting programs, which is not a goal of Mizar [80]. In Coq and Minlog [76], you usually start with a sequence of *tactic* applications, which try to automatically break down the problem into smaller subproblems. This sequence of tactics can be converted into a proof tree. After such a proof is finished, the generated proof tree is checked by a kernel. Therefore, tactics can be written in an unverified manner, and might yield wrong proofs, but these proofs would be rejected by the kernel. This kind of proof checker is called LCF-Style, because the LCF proof checker used this model. We say, the proof term is *reconstructed* from this sequence of tactics. An example is our code from Section 3.2.

If it is a classical proof, some *Glivenko-Style theorem* is usually applied. Such theorems are in some form generalizations of Glivenko's Theorem:

**Theorem 2.** *Let $\Gamma$ be a set of propositional axioms, and $\gamma$ be some propositional formula following classically from $\Gamma$. Then $\neg\neg\gamma$ follows intuitionistically from $\{\neg\neg\vartheta \mid \vartheta \in \Gamma\}$.*

The proof does structural induction on proof terms. If we can classically prove some fact $\forall_x A(x)$, then we can intuitionistically prove $\neg\neg\forall_x A(x)$, which is equivalent to $\neg\exists_x \neg A(x)$, which relates it to the *no counterexample interpretation*:

Some systems, specifically Minlog [76], distinguish between proofs and programs, and therefore define *realizers* for terms. An early version which does this specifically for Peano arithmetic was Gödel's *Dialectica Interpretation* [27]. One usually uses $+$ and $\times$ instead of $\vee$ and $\wedge$ on the type-level in this context, and defines, as interpretation $\tau$:

$$\begin{aligned}
\tau(\bot) &:= \bot \\
\tau(A \wedge B) &:= \tau(A) \times \tau(B) \\
\tau(A \vee B) &:= \tau(A) + \tau(B) \\
\tau(A \to B) &:= \tau(A) \to \tau(B) \\
\tau(\forall_{x^\delta} Q) &:= \delta \to \tau(Q) \\
\tau(\exists_{x^\delta} Q) &:= \delta \times \tau(Q)
\end{aligned}$$

Specifically, $\tau(\neg Q) = \tau(Q) \to \bot$. In dependent type theory, proofs and programs need no distinction, and the type of a term is the same as the proved formula of a proof, because types are formulas and terms are proofs. However, for the moment, the distinction between proofs and terms makes things easier. In all systems that are suitable for program extraction, the existence of a closed term $t^{\tau(A)}$ is equivalent to the existence of a proof of $A$.

Now, we have $\tau(\neg\exists_{x^\delta}\neg A) = \tau(\exists_{x^\delta}\neg A) \to \bot$, meaning that a proof of $\exists_{x^\delta}\neg A$ would yield an element of the empty type, which is impossible. But $\tau(\exists_{x^\delta}\neg A) = \delta \times \tau(\neg A)$ is the type of counterexamples for $A$. Hence, the mixture of the negative translation and the dialectica interpretation is called no counterexample interpretation.

In [31], a classical consequence of our Theorem 1 for pairs of numbers is given:

**Theorem 3.** $\forall_{f,g:\mathbb{N}\to\mathbb{N}}\exists_{i,j:\mathbb{N}}. i < j \wedge f(j) \not< f(i) \wedge g(j) \not< g(i)$

It uses the following minimum principle:

**Lemma 3.** *Given a type $A$, a property $P : A \to \mathrm{Prop}$, and a measure function $m : A \to \mathbb{N}$, then $\exists_{x^A} P(x) \to \exists_{x^A}(P(x) \wedge \forall_y. m(y) < m(x) \to \neg P(y))$, that is, there exists an $m$-minimal $x^A$ which satisfies $P(x)$.*

This minimum principle can be proven by classical reasoning and strong induction:

*Proof.* By eliminating the first existential quantifier, we get some $x^A$. We now do strong induction on $m(x^A)$. If $m(x^A) = 0$, we are done, since nothing is smaller than 0. If $m(x^A) = n + 1$, then $x^A$ might already be $m$-minimal (this case distinction is strictly classical). Otherwise, there must be some $y^A$ and $q$ such that $m(y^A) = q \leq n$, and by induction hypothesis with $n := q$, we get a minimal element. □

The rest of the proof will be constructive:

*Proof of Theorem 3.* Let $M(x) := \forall_{y \geq x}. f(y) \not< f(x)$. Using $m = f$ and $P(x) = \top$ in the minimum principle, we get a global minimum $\delta$ of $f$, meaning $M(\delta)$. Therefore, $\exists_x M(x)$. Therefore, by the minimum principle with $M$ and $g$, we get a $g$-minimal element $i$ with $M(i)$, that is, $M(i) \wedge \forall_y(g(y) < g(i) \to \neg M(y))$. Now, again, we can apply the minimum principle with $\lambda_x.x > i$ and $f$, and we get a $j > i$ which is $f$-minimal, that is, $\forall_{y>i}. f(y) \not< f(j)$. Since $M(i)$, we now have $f(j) \not< f(i)$. On the other hand, since $j$ is the minimal value of $f$ such that $j > i$, we have $\forall_{y \geq j} f(y) \not< f(j)$, meaning $M(j)$. Since $i < j$, and by the minimality of $i$, it follows that $g(j) \not< g(i)$. Hence, $i$ and $j$ are as desired. □

Even though this proof is classical, and we cannot extract from the minimum principle in general, [31] uses negative translation and Dragalin-Friedman-translation to obtain a constructive proof (we discussed this procedure in [88]), and extracts an algorithm which is searching for such a pair $(i, j)$. However, the resulting algorithm is somewhat hard to grasp, a simplified version is given as

```
modifiedSolution f g = mod [] 0 1 where
  mod s i j = if g j < g i
              then mod (i : s) j (j + 1)
              else if f j < f i
                   then case s of
                          []       -> mod [] j $ j + 1
                          (x : xs) -> mod xs x j
                   else (i, j)
```

However, this shows that with this kind of program extraction, the resulting algorithms might be complicated. It is also difficult to estimate their runtime behavior without directly looking at them, because the correspondence between the existence proof and the resulting extracted term is much weaker. We do not use this kind of program extraction in our implementation.

As in this case, we will get a constructive proof of some refinement of the original theorem which still reflects its computational content. We call this phase *translation* of the proof. In some cases, not all classical applications are removed; for example, Coq has the type-universe `Prop` which may contain classical reasoning, but otherwise Coq proofs are usually constructive from the start, so no negative translation is needed. Notice

that it is not always possible to convert proofs into constructive proofs with the desired properties. Our above example shows this. In Coq, usually all proofs are constructive. It is possible to use classical reasoning, but only in the non-computational parts of the proof, which will be removed in the next phase.

After gaining a constructive proof term, it is *pruned*: The parts which are computationally irrelevant are removed. This is the part where we really "extract" something from it, it is the first time we do not care about the proof anymore and concentrate on the algorithm it resembles. Simple *dead-code-elimination* can be applied. This is an a posteriori approach. Coq, on the other hand, has an a priori approach, by distinguishing between computationally relevant types `Set`, and computationally irrelevant types `Prop`. Minlog uses a somewhat intermediate approach where proofs can be automatically *decorated*: Every $\to$ and $\forall$ gets an annotation $c$ or $nc$, which annotates whether this implication or quantification should be regarded as computationally relevant. The computational relevance of axioms must be given before.

After pruning the proof, we usually have a term which has another type than the original proof. Ideally, it is provable that the pruned term *realizes* the original proof term regarding some carefully crafted realizer relation. The property we usually want is that if $t : \forall_{x^A} \exists_{y^B} Rxy$ is proved, then the corresponding extracted program $t' : A \to B$ satisfies $\forall_{x^A} Rx(t'x)$.

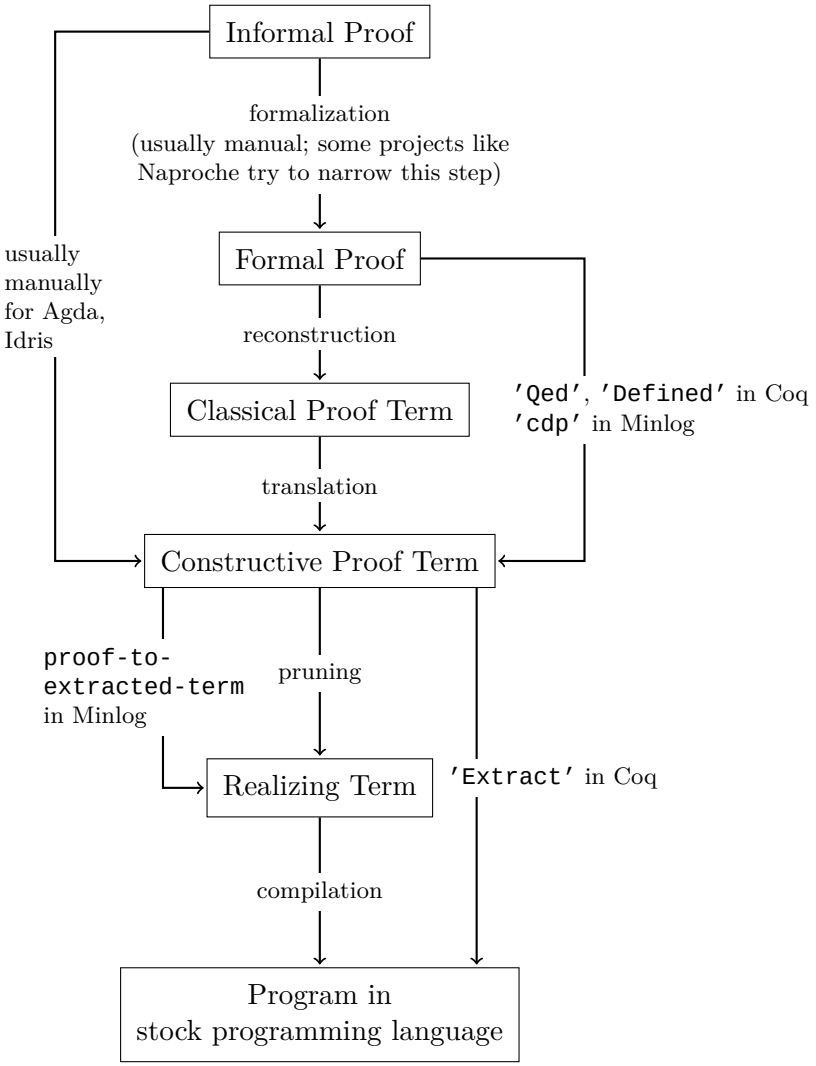
Getting a term inside the theory itself is called *internal extraction*. In the case of Coq, the extraction mechanism computes a term in some stock programming language, in our case this will usually be Haskell. This is called *external extraction*. Still, at least indirectly, there will be an internal term before external extraction. We call the transition from internal extraction to some stock programming language *compilation*, conforming with the common notion of programming languages being "compiled" into other programming languages, as is done in compiler backends like Emscripten [11], Chicken Scheme [7] or Stalin [14]. Sometimes, the readability of the compiled code from extracted algorithms is criticized. However, the aforementioned compilers do not explicitly have the goal of producing readable code, and we think that a compiler does not need to produce readable code, rather than correct and efficient code. However, besides the use of recursion operators instead of pattern matching, the extracted code is quite readable.

Often, in Coq, people are starting from having a function $t' : A \to B$ and proving its correctness by a proof about $t'$ rather than by extracting $t'$ from a proof about some existence, but still call the process of compiling it to some stock programming language "program extraction". We sometimes used this technique when it was more feasible; sometimes, especially when we had a complicated induction scheme, we even mixed both styles, using the `refine` statement, for example in `DeflateCoding.v`:

```
Lemma existence_lex_lemma : forall a m p, lex p m -> prefix a m ->
                                    ~ prefix a p -> lex p a.
Proof.
  refine (fix f a m p :=
            match (a, m, p) as R return ((a, m, p) = R -> _) with
              | (nil, _, _) => fun eq => _
              | (xa :: a', nil, _) => fun eq => _
              | (xa :: a', xm :: m', nil) => fun eq => _
              | (xa :: a', xm :: m', xp :: p') => fun eq => _
            end eq_refl).
```

How far these techniques should be regarded as program extraction is a discussion that is besides the point of our work.

In general, we can factorize "program extraction" in the following way:




Notice that this diagram only has the purpose of clarifying terminology,

and only the terminology we use in this work. Neither is it normative, nor is it a complete discussion of all details.

It can sometimes be hard to estimate the efficiency of an extracted program. The case study [81] points out some techniques to improve efficiency, which we mostly followed. In particular, these were

- to use expensive statements non-computationally, which we have done in large parts of the code.
- to use existential quantifiers as memory, which we did, for example, in our proofs regarding strong decidability (see Section 6).
- to calculate values in advance, which we did, for example, for the value `fixed_lit_code`.
- to turn loop invariants into induction statements, which is not directly applicable because Coq does not have imperative loops, but corresponds to Coq's induction measures, which give a clue about the computational complexity.

## 3.5 Moravec's Paradox

In the realm of artificial intelligence, there is a paradoxical fact, named after Hans Moravec, which states that it is often the case that things that are trivial for humans are complicated or even impossible for computers, but also the other way around. In [78], he states *"In hindsight, this dichotomy is not surprising. Since the first multicelled animals appeared about a billion years ago, survival in the fierce competition over such limited resources as space, food or mates has often been awarded to the animal that could most quickly produce a correct action from inconclusive perceptions. [...] Abstract thought, though, is a new trick, perhaps less than 100 thousand years old. We have not yet mastered it. It is not all that intrinsically difficult; it just seems so when we do it."* In formal methods, there is a similar dichotomy. If you are trying to formalize higher mathematics, it is often clear, or at least very plausible, that a theorem is formalizable, although actually doing so may require a significant amount of work. For example, proving that $\pi_1(S^1) \cong \mathbb{Z}$ uses an argument that uses a covering of $S^1$ and a lifting and several properties of the resulting curve which are intuitive but hard to formalize. In formal mathematics, the concrete formalization of a theorem can be important for the difficulty of its proof. To be able to formalize such theorems, Homotopy Type Theory uses another formalization of mathematics which makes it easy to prove this. On the other hand, making sure that in a large amount of case distinctions all cases are properly handled, like in the four-color-theorem [55], computers have a clear advantage.

In the same way, it is not always clear which parts of our project are difficult. A major peak in difficulty were Deflate codings, see Section 5. We had to deal with binary sequences, which were as well interpreted as numbers as they were as sequences. Though, the informal proofs are not hard to understand.

Furthermore, understanding simple algorithms, like the backreference algorithm with DiffArrays in Section 8.4, is easy for humans, but finding out the actual invariants and proving them formally is a lot harder.

Of course, there are situations where computers and humans have about the same level of difficulty: The `omega` strategy solves many simple equations which are also "clear" to humans. On the other hand, the purely functional algorithm for resolution of backreferences, which we presented in Section 8.5, is comparably difficult for both humans and computers.

It is, in any case, not always easy to estimate the difficulty of a formal proof in advance. One should keep this in mind when reading this work. We will try to point out the difficulties we found and how we coped with them.

## 3.6 Practical Applications

Every sophisticated algorithm should come with a proof. Therefore, in theory, program extraction should make it easier to produce sophisticated algorithms, because you only need to formalize the proof, and get the algorithm for free.

Unfortunately, reality looks different. While formal methods are becoming more and more widespread, program extraction from proofs is less popular, even though formally, there is no need to distinguish between dependently typed programs and proofs. The main difference is how you write these programs.

The following is a dependently typed proof of

**Theorem 4.** $0 + n = n$

where addition is defined recursively by

```
_+_ : Nat → Nat → Nat
a + O = a
a + (S n) = S (a + n)
```

and hence this equality must be proved. The informal proof is easy:

*Proof.* We prove this by induction: For $n = 0$, it holds by definition. Now consider $Sn$. We have $0 + Sn = S(0 + n)$, and by the induction hypothesis $0 + n = n$, we have $S(0 + n) = Sn$, and hence, $0 + Sn = Sn$. □

The proof in Agda is

```
n0 : (n : Nat) -> O + n ≡ n
n0 O = refl
n0 (S n) = cong S (n0 n)
```

which is an explicit proof term, with some implicit arguments. In Coq, one would proof this rather by

```
Goal forall n, O + n = n.
Proof.
  induction n.
  reflexivity.
  simpl.
  rewrite -> IHn.
  reflexivity.
Qed.
```

The second code feels more like a list of instructions, while the first one feels more like a program. The first one is much shorter in this case. However, this is not the case in general; for example, we can prove simple number theoretic equations using the `omega` tactic, a oneliner. In this case, the second one seems more readable. This is also not always the case: Using complicated tactics and tactic combinators can make proofs hardly readable. Generally speaking, in dependently typed programming, you build up proofs, while in tactic based program extraction, you break down propositions. However, Agda also has some tactics, for example, the `RingSolver` tactic [19].

We started the project with Agda. Our first specification of Deflate codings (see Secion 5) was motivated by trying to define code trees:

```
data DeflateTree {alpha : ℕ} : (range : Subset alpha)
                             → (shortHeight : ℕ)
                             → (shortBranchChar : Fin alpha)
                             → (longHeight : ℕ)
                             → (longBranchChar : Fin alpha)
                             → (notNonFork : Bool) → Set where
  leaf : (b : Fin alpha) → DeflateTree ｛ b ｝ 0 b 0 b true
  forkEq : {ld rd md : ℕ} → {f g : Subset alpha} →
           {lc mc1 mc2 rc : Fin alpha} → {tl : Bool}
           → DeflateTree f ld lc md mc1 true
           → DeflateTree g md mc2 rd rc tl
           → (disjoint : Empty (f ∩ g))
           → (toℕ mc1) < (toℕ mc2)
           → DeflateTree (f ∪ g) ld lc rd rc tl
  forkNeq : {ld md1 md2 rd : ℕ} → {f g : Subset alpha}
          → {lc mc1 mc2 rc : Fin alpha} → {tl : Bool}
          → DeflateTree f ld lc md1 mc1 true
          → DeflateTree g md2 mc2 rd rc tl
          → (disjoint : Empty (f ∩ g))
          → md1 < md2 → DeflateTree (f ∪ g) ld lc rd rc tl
  nonFork : {b : Bool} → {ld rd : ℕ} → {f : Subset alpha}
          → {lc rc : Fin alpha}
```

```
        → DeflateTree f ld lc rd rc b
        → DeflateTree f (suc ld) lc (suc rd) rc false
```

This directly and locally specifies the structure one can produce, and has a directly visible correspondence to what will be saved in the memory when the program runs. However, this also means that it is not possible to temporarily break these invariants, which is sometimes necessary.

At the time we began our project, Agda's standard library was not mature enough, which was the reason we switched to Coq. While Coq's proof mechanism focuses on tactic-based proving, Coq allows for both styles of programming, and we often mix both styles, depending on which is more apropriate for the specific problem.

While in Agda, we directly see the terms we programmed, in Coq, we have to trust the extraction algorithm, which enlargens the trusted codebase. There are, however, verified algorithms in work [62].

We think that especially for complicated type signatures, the break-down-approach of tactic based proving and program extraction are more beneficial, since breaking down large propositions into smaller ones has to be done in any case. Then, after profiling the resulting algorithms, one can replace bottlenecks with explicitly crafted proof terms.

In [31], some examples of realistic program extraction are given, but they are still no more than a case study. A more practical example is the DPLL-Solver we discussed in Section 1.4.3.

# Chapter 4

# An Introduction To Coq

As we use Coq for our project, we give a short introduction to Coq by examples. This introduction is by no means complete, and mainly for readers familiar with some other proof checker, to get an overview. For an example of program extraction, refer to Section 3.

## 4.1 Set and Prop

The Coq type system is documented in the Coq reference manual. A rule of thumb is that there is a `Type` universe (actually, there are infinitely many, but they are hidden by the language), which contains `Prop` and `Set`. Objects of type `Prop` are computationally irrelevant, while objects of type `Set` are relevant and can be extracted. For example, if we wanted to define our own type of natural numbers (though there is a type `nat` in the standard library with intrinsic decimal representation) one would define them as `Set`, like

```
Inductive Nat : Set :=
| O : Nat
| S : Nat -> Nat.
```

because we want to be able to use it in programs. The definition

```
Inductive NatEven : Nat -> Prop :=
| OEven : NatEven O
| SSEven : forall n, NatEven n -> NatEven (S (S n)).
```

is a `Prop`: We are usually not interested in the structure of the proofs of evenness of a natural number, we just need the fact that they are. We can then, of course, not eliminate an evenness proof inside a computationally relevant context.

## 4.2 Gauss formula

As a more sophisticated example, we now want to prove the Gauss formula

$$2\sum_{i=0}^{n} i = n(n+1)$$

Firstly, we need to import the packages we need, in our case, we only need the `omega` tactic

```
Require Import Omega.
```

It is commonly defined that

$$\sum_{i=0}^{k} w_i := \begin{cases} w_0 & \text{for } k = 0 \\ w_{n+1} + \sum_{i=0}^{n} w_i & \text{for } k = n+1 \end{cases}$$

In Coq, we do this using a `Fixpoint`:

```
Fixpoint sum (to : nat) (what : nat -> nat) :=
  match to with
    | 0 => what 0
    | S n => what (S n) + sum n what
  end.
```

The Gaussian sum is now for $w_i = i$, or `what = Id`:

```
Definition Id (x : nat) := x.
Definition GaussSum n := sum n Id.
```

We can now compute a Gaussian sum by

```
Eval compute in GaussSum 10.
```

which correctly yields 55. The proof itself starts with the statement and declaration.

```
Lemma GaussSumFormula : forall n, 2 * GaussSum n = n * (n + 1).
Proof.
```

We prove this by induction on $n$.

```
  induction n as [|n].
```

The `as` clause tells Coq the name of the destructured parameter: A natural number can either be 0, in which case we need no additional name, and the successor of something, which we call $n$, overloading the original meaning of $n$. We now have two cases, therefore, we can prefix our instructions with the bullet `+`. The first case is `2 * GaussSum 0 = 0 * (0 + 1)`, which we can just prove by `reflexivity`, since evaluating it results in the same term.

```
  + reflexivity.
```

The second goal is `2 * sum (S n) Id = S n * (S n + 1)`, but we can use the induction hypothesis, which Coq automatically called `IHn`, which is `2 * GaussSum n = n * (n + 1)`. We need to help Coq a little to find the right way of expanding a term: We need the equality `sum (S n) Id = (S n) + GaussSum n` which is trivial, but we need it in this form to replace a subterm in our goal.

```
  + unfold GaussSum.
    assert (X : sum (S n) Id = (S n) + GaussSum n).
    - reflexivity.
```

We can now use this equality to replace the part in our term with the equal term

```
    - rewrite -> X.
```

The goal becomes `2 * (S n + GaussSum n) = S n * (S n + 1)`, which we rewrite to `2 * S n + 2 * GaussSum n = S n * (S n + 1)` by distributivity:

```
      rewrite -> mult_plus_distr_l.
```

We finally can apply our induction hypothesis

```
      rewrite -> IHn.
```

yielding the new goal `2 * S n + n * (n + 1) = S n * (S n + 1)`. Now we do

```
      replace (S n) with (n + 1); [|omega].
```

This tells Coq to replace the given subterm with the new term, and prove the equality using the `omega` tactic. We apply commutativity and distributivity once again

```
      rewrite <- mult_plus_distr_r.
      rewrite -> mult_comm.
```

and gain the goal `(n + 1) * (2 + n) = (n + 1) * (n + 1 + 1)`. This is a goal of the form $f(a) = f(b)$, which can be reduced to the goal $a = b$ by the tactic

```
      f_equal.
```

yielding the goal `2 + n = n + 1 + 1` which can finally be solved by the tactic

```
      omega.
Qed.
```

This concludes the proof.

## 4.3 Square Pyramidal Numbers

A similar formula holds for the sum of squares, namely the formula for the $n$-th square pyramidal number,

$$6 \cdot \sum_{i=0}^{n} i^2 = n(n+1)(2n+1)$$

However, the equality we need in the induction is more complicated and can be automated. For this, there is the `ring` tactic, which allows us to define rings and semirings, and automatically solve equalities in them. We need to import two additional libraries, the first one for the ring theory itself, and the second one for the tactic.

```
Require Import Omega.
Require Import Coq.setoid_ring.Ring_theory.
Require Import Ring.
```

We now want to prove that the natural numbers form a semiring. The ring theory library declares a record that contains all the axioms, parametrized on the objects and operations for $0, 1, +, \cdot, =$:

```
  Record semi_ring_theory : Prop := mk_srt {
    SRadd_0_l : forall n, 0 + n == n;
    SRadd_comm : forall n m, n + m == m + n ;
    SRadd_assoc : forall n m p, n + (m + p) == (n + m) + p;
    SRmul_1_l : forall n, 1*n == n;
    SRmul_0_l : forall n, 0*n == 0;
    SRmul_comm : forall n m, n*m == m*n;
    SRmul_assoc : forall n m p, n*(m*p) == (n*m)*p;
    SRdistr_l : forall n m p, (n + m)*p == n*p + m*p
  }.
```

To realize this record, we prove a lemma:

```
Lemma SRNat : semi_ring_theory 0 1 plus mult eq.
Proof.
  constructor;
    intros;
    (omega ||
     apply mult_comm ||
     apply mult_assoc ||
     apply mult_plus_distr_r).
Qed.
```

This proof utilizes the tactic language of Coq. The call to `constructor` produces the goals, which are the fields of the `semi_ring_theory` record. Everything after the semicolon is applied to all goals. `intros` eliminates leading universal quantifiers and puts them into the antecedent. The `||` operator tells Coq to use the first tactic that solves the goal. For three

of the goals, this can just be done by the `omega` tactic, the rest are the respective laws proven directly in the standard library.

To use it, we do

```
Add Ring RNat : SRNat.
```

Now we can prove our formula.

```
Lemma PyramidFormula : forall n,
    6 * sum n (fun x => x * x) = n * (n + 1) * (2 * n + 1).
Proof.
```

The `ring` tactic has no notion of numbers other than 0 and 1, hence we replace these numbers, using the tactic `omega`:

```
  replace 6 with ((1+1)*(1+1+1)) by omega.
  replace 2 with (1+1) by omega.
```

We then need to do a bit of work, similar to our work for the Gaussian formula:

```
  induction n as [|n].
  - reflexivity.
  - replace (sum (S n) (fun x : nat => x * x))
    with (S n * S n + sum n (fun x : nat => x * x))
      by reflexivity.
    replace (S n) with (n + 1) by omega.
    rewrite -> mult_plus_distr_l.
    rewrite -> IHn.
```

After that, the goal will be `(1 + 1) * (1 + 1 + 1) * ((n + 1) * (n + 1)) + n * (n + 1) * ((1 + 1) * n + 1) = (n + 1) * (n + 1 + 1) * ((1 + 1) * (n + 1) + 1)`, which can be solved by the semiring solver

```
    ring.
Qed.
```

# Chapter 5

# Deflate Codings

Even though this section is about what we call "Deflate Codings", it is not specific to Deflate at all: The same mechanism is used, for example, in BZip2, too. It is more about coding theory in general, with an emphasis on Deflate.

It is a well-known result from [61] that for every string $A \in [\mathcal{A}]$ over a finite alphabet $\mathcal{A}$, there is a **Huffman coding** $h : \mathcal{A} \to [\{0,1\}]$, which is a prefix-free coding such that $\text{fold}_{(+\!+)}[\,](\text{map}_h\, A)$ has minimal length. In fact, this has already been formally proved [33]. While this is useful for compression, Deflate does not require codings to be optimal. Furthermore, there is more than one optimal coding; for example, if two symbols are encoded by equally long codes, we can swap their encodings without losing minimality. Deflate codings are special prefix-free codings which are uniquely determined by their code lengths. Therefore, instead of expensively saving a tree structure, it is sufficient to save the sequence of code lengths for the encoded characters. Other standards, like BZip2, have a similar way of saving codings, and therefore, our implementation might be usable for implementations of them, too. Optimal Deflate codings are also known as **canonical Huffman codings**. To our best knowledge, there is no verified formalization of Deflate codings so far.

In any practical case, there will be a canonical ordering on $\mathcal{A}$, so from now on, let us silently assume $\mathcal{A} = \{0, \ldots, n-1\}$.

**Definition 1.** *The* **lexicographical order** $\sqsubseteq$ *on* $[\{0,1\}]$ *is defined by*

$$\forall_a.[\,] \sqsubseteq a$$
$$\forall_{a,b}.0 :: a \sqsubseteq 1 :: b$$
$$\forall_{a,b,j}.a \sqsubseteq b \to j :: a \sqsubseteq j :: b$$

It is easy to show that the lexicographical order is in fact a decidable total ordering relation. This, and some additional lemmata that we need later, is proved in `Lex.v`.

**Definition 2.** *Let $a, b \in [\{0,1\}]$. We say $a$ is a* **prefix** *of $b$ and write $a \preccurlyeq b$, if there is a code $c \in [\{0,1\}]$ such that $a +\!\!+\ c = b$. Notice that $a \preccurlyeq a$.*

Decidability of the prefix relation on Boolean lists, transitivity and some other small lemmata are proved in `Prefix.v`. In practice, the code [] is used to denote that the corresponding codepoint does not occur, and we will therefore exclude this special case from our definition of prefix-freeness.

**Definition 3.** *A* **Deflate coding** *is a coding $\lceil\cdot\rceil : \mathcal{A} \to [\{0,1\}]$ which satisfies the following conditions:*

1. *$\lceil\cdot\rceil$ is prefix-free, except that there may be codes of length zero:*

$$\forall_{a,b}.(a \neq b \wedge \lceil a\rceil \neq [\,]) \to \lceil a\rceil \not\preccurlyeq \lceil b\rceil$$

2. *Shorter codes lexicographically precede longer codes:*

$$\forall_{a,b}.\,\mathrm{len}\lceil a\rceil < \mathrm{len}\lceil b\rceil \to \lceil a\rceil \sqsubseteq \lceil b\rceil$$

3. *Codes of the same length are ordered lexicographically according to the order of the characters they encode:*

$$\forall_{a,b}.(\mathrm{len}\lceil a\rceil = \mathrm{len}\lceil b\rceil \wedge a \leq b) \to \lceil a\rceil \sqsubseteq \lceil b\rceil$$

4. *For every code, all lexicographically smaller bit sequences of the same length are prefixed by some code:*

$$\forall_{a\in\mathcal{A},l\in\{0,1\}^+}.(l \sqsubseteq \lceil a\rceil \wedge \mathrm{len}\, l = \mathrm{len}\lceil a\rceil) \to \exists_b.\lceil b\rceil \neq [\,] \wedge \lceil b\rceil \preccurlyeq l$$

The corresponding Coq definition is the record `deflateCoding` which can be found in `DeflateCoding.v`. These Axioms are our proposed formalization of the informal specification in [41], which states: *The Huffman codes used for each alphabet in the "deflate" format have two additional rules:*

- *All codes of a given bit length have lexicographically consecutive values, in the same order as the symbols they represent;*
- *Shorter codes lexicographically precede longer codes.*

[41] claims that with these two rules, Huffman codings are uniquely determined by their code lengths. Even though this might be true, Huffman codings are defined for strings – from a given coding, you cannot tell whether it is a Huffman coding. However, of course, some codings are clearly not Huffman codings, for example the coding

$$0 \to [0], 1 \to [1,0,0], 2 \to [1,0,1], 3 \to [1,1,0]$$

is clearly not a Huffman coding, since for every case it would apply to, we could also use

$$0 \to [0], 1 \to [1,0], 2 \to [1,1,1], 3 \to [1,1,0]$$

which will always be better. We tried to introduce the concept of "optimal codings", but as it appeared to be harder to use and less general than the definition we actually used, and all the reference algorithms in [41] work for non-optimal codings too, we decided to follow Postel's Robustness Principle [82] and accept them.

Axiom 3 is weaker than the first axiom from [41], as it does not postulate the consecutivity of the values, which is ensured by axiom 4: Assuming you have characters $a < b$ such that $\text{len}\lceil a \rceil = \text{len}\lceil b \rceil$, and there is a $l \in \{0,1\}^{\text{len}\lceil a \rceil}$ such that $\lceil a \rceil \sqsubseteq l \sqsubseteq \lceil b \rceil$, then there is a $d$ such that $\lceil d \rceil \preccurlyeq l$, and therefore $\lceil d \rceil = l$ and $a \leq d \leq b$, so the values are consecutive.

Furthermore, consider our non-optimal coding from above: It has the lengths $0 \to 1, 1 \to 3, 2 \to 3, 3 \to 3$, and satisfies our axioms 1-3, and additionally, the codes of the same length have lexicographically consecutive values. But the same holds for the coding

$$0 \to [0], 1 \to [1,0,1], 2 \to [1,1,0], 3 \to [1,1,1]$$

However, in this coding, there is a "gap" between the codes of different lengths, namely between $[0]$ and $[1,0,1]$, and that is why it violates our axiom 4: The list $[1,0,0]$ is lexicographically smaller than $[1,0,1]$, but it has no prefix.

We can show that Deflate codings are uniquely determined by their code lengths:

**Theorem 5** (uniqueness)**.** *Let* $\lceil \cdot \rceil, \lfloor \cdot \rfloor : \mathcal{A} \to [\{0,1\}]$ *be two Deflate codings, such that* $\text{len}\lceil \cdot \rceil = \text{len}\lfloor \cdot \rfloor$*. Then* $\lceil \cdot \rceil = \lfloor \cdot \rfloor$*.*

*Proof.* Besides the fact that equality is computationally irrelevant, equality of codings is obviously decidable, therefore we can do a proof by contradiction. So assume there were two distinct deflate codings $\lceil \cdot \rceil$ and $\lfloor \cdot \rfloor$ with $\text{len}\lceil \cdot \rceil = \text{len}\lfloor \cdot \rfloor$. Then there must exist $n, m$ such that $\lceil n \rceil = \min_{\sqsubseteq}\{\lceil x \rceil \mid \lceil x \rceil \neq \lfloor x \rfloor\}$ and $\lfloor m \rfloor = \min_{\sqsubseteq}\{\lfloor x \rfloor \mid \lceil x \rceil \neq \lfloor x \rfloor\}$.

If $\text{len}\lceil n \rceil > \text{len}\lfloor m \rfloor$, then also $\text{len}\lceil n \rceil > \text{len}\lceil m \rceil$, and by our axiom 2, $\lceil m \rceil \sqsubseteq \lceil n \rceil$. But $n$ was chosen minimally. Symmetric for $\text{len}\lceil n \rceil > \text{len}\lfloor m \rfloor$. Therefore, $\text{len}\lceil n \rceil = \text{len}\lfloor m \rfloor$.

Assuming $n < m$, by axiom 3, $\lfloor n \rfloor \sqsubseteq \lfloor m \rfloor$, but $m$ was chosen minimally. Symmetric for $n > m$. Hence, $n = m$.

$\lfloor m \rfloor \neq [\,]$, because otherwise $0 = \text{len}\lfloor m \rfloor = \text{len}\lceil n \rceil$, so $\lceil n \rceil = [\,]$, and so $\lceil m \rceil = \lfloor m \rfloor$, which contradicts our assumption on the choice of $m$. Analogously, $\lceil n \rceil \neq [\,]$.

By totality of $\sqsubseteq$, we know that $\lceil n \rceil \sqsubseteq \lfloor m \rfloor \vee \sqsubseteq \lfloor m \rfloor \sqsubseteq \lceil n \rceil$. Both cases are symmetric, so without loss of generality assume $\lceil n \rceil \sqsubseteq \lfloor m \rfloor$. Now, by axiom 4, we know that some $b$ exists, such that $\lfloor b \rfloor \preccurlyeq \lceil n \rceil$, therefore by axiom 2, $\lfloor b \rfloor \sqsubseteq \lfloor m \rfloor$, and thus, by the minimality of $m$, either $b = m$ or $\lfloor b \rfloor = \lceil b \rceil$.

But since $n = m$, $b = m$ would imply $b = n$, and hence, $\lfloor m \rfloor \preccurlyeq \lceil n \rceil$, and therefore, since the lengths are equal, $\lfloor m \rfloor = \lceil m \rceil$, which contradicts our choice of $m$.

But $\lfloor b \rfloor = \lceil b \rceil$ would imply $\lceil b \rceil \preccurlyeq \lceil n \rceil$, which contradicts our axiom 1. □

This theorem is proved as Lemma `uniqueness` in `DeflateCoding.v`. While uniqueness is a desirable property, it does not give us the guarantee that, for every configuration of lengths, there actually is a Deflate coding. And in fact, there isn't. Trivially, for the sequence of code lengths $1, 1, 1$, there is no Deflate coding. It is desirable to have a simple criterion on the list of code lengths, that can be efficiently checked, before creating the actual coding.

**Theorem 6** (Kraft's inequality for sets of codes)**.** *Let $k_1, \ldots, k_N$ be non-negative integers. Then the following propositions are equivalent:*

1. *There is a prefix-free $S = \{s_1, \ldots, s_N\} \subseteq [\{0,1\}]$ such that $k_i = \operatorname{len} s_i$ for all $i \in \{1, \ldots, N\}$*
2. $\sum_{i=1}^{N} 2^{-k_i} \leq 1$

*Equality in 2. holds if and only if there is no $a \in [\{0,1\}] \setminus S$ such that $S \cup \{a\}$ is prefix-free.*

This is a well-known result, see e.g. [75]. We only prove 1. ⇒ 2. directly, since we need the stronger proposition for the other direction from Theorem 7. Notice that we are talking about sets of codes, and not about codings, now. In our implementation, we use duplicate- and prefix-free lists. The implementation is mainly in `KraftList.v`. The reason why we do not directly use the codings is that we allow our codings' images to contain `[]`, as this is closer to the standard [41]. However, Kraft's inequality also holds for the set $\{[]\}$, and here it is even sharp, because `[]` is a prefix of all lists, and this is the base-case for our induction. The theorem is proved in `Lemma kraft_pflist`, the sharp version in `Lemma kraft_pflist_sharp`, in `KraftList.v`. To then apply it to our Deflate coding data structures, there are several lemmata in `KraftVec.v`.

In the following, let $\aleph S := \sum_{i \in S} 2^{-\operatorname{len} i}$, $i :: S := \{i :: x \mid x \in S\}$ and $S_i := \{x \mid i :: x \in S\}$. Notice that $S = 0 :: S_0 \cup 1 :: S_1$.

**Lemma 4** (`pflist_splittable`)**.** *If $S$ is prefix-free, then so is $S_i$ for $i \in \{0,1\}$.*

*Proof.* As prefix-freedom is a negative property, we can use a proof by contradiction. Assume $S_i$ contained a prefix $a \preccurlyeq b$. Then $i :: a \preccurlyeq i :: b$. Contradiction. □

**Lemma 5** (`maxlen_split`)**.** *Let* $\operatorname{maxlen} S := \max\{\operatorname{len} x \mid x \in S\}$ *and* $\operatorname{maxlen} \emptyset := 0$. *If* $\operatorname{maxlen} S \neq 0$, *then* $\operatorname{maxlen} S_i < \operatorname{maxlen} S$ *for* $i \in \{0, 1\}$.

*Proof.* We do this proof in a more complicated way than necessary, to be closer to the formal proof in Coq (which is about lists rather than sets). We do an induction on the size of $S$. If $S = \emptyset$ we are done. Now assume $S = \{a\} \dot\cup S'$, and the proposition holds for $S'$ already. If $a = (1 - i) :: a'$, then $S_i = S'_i$, and since then trivially $\operatorname{maxlen} S \geq \operatorname{maxlen} S'$, the proposition holds. If $a = i :: a'$, then $\operatorname{len} a \leq \operatorname{maxlen} S$. If $\operatorname{len} a = \operatorname{maxlen} S$, then trivially, $\operatorname{len} a' = \operatorname{maxlen} S_i$, because it must be greater than or equal any elements of any subset of $S$ as well. But $\operatorname{len} a' = \operatorname{len} a - 1 < \operatorname{len} a$. Now assume the case $\operatorname{len} a < \operatorname{maxlen} S$. If $\operatorname{len} a' = \operatorname{maxlen} S_i$, we are done, since $\operatorname{len} a' = \operatorname{len} a - 1$. If $\operatorname{len} a' < \operatorname{maxlen} S_i$, then $\operatorname{maxlen} S_i = \operatorname{maxlen} S'_i$, and also $\operatorname{maxlen} S = \operatorname{maxlen} S'$, and the claim follows by induction. □

**Lemma 6** (`kraft_pflist_split`)**.** *Let* $S = \{s_1, \ldots, s_N\} \subseteq [\{0, 1\}]$ *be prefix-free. Then* $[\,] \in S$ *or* $\mathfrak{K} S = 2(\mathfrak{K} S_0 + \mathfrak{K} S_1)$.

*Proof.* Let $\{s_1, \ldots, s_N\}$ be prefix-free, and $[\,] \neq s_k$ for all $k$. Trivially,

$$\mathfrak{K} S = \mathfrak{K}(S \cap \{0 :: l \mid l \in [\{0, 1\}]\}) + \mathfrak{K}(S \cap \{1 :: l \mid l \in [\{0, 1\}]\})$$

But

$$\begin{aligned}
\mathfrak{K}(S \cap \{j :: l \mid l \in [\{0, 1\}]\}) &= \sum_{(j::i) \in S} 2^{-\operatorname{len} j::i} \\
&= \sum_{(j::i) \in S} 2^{-(1+\operatorname{len} i)} \\
&= \frac{1}{2} \sum_{(j::i) \in S} 2^{-\operatorname{len} i} \\
&= \frac{1}{2} \sum_{i \in \{l \mid j::l \in S\}} 2^{-\operatorname{len} i} \\
&= \frac{1}{2} \mathfrak{K}\{l \mid j :: l \in S\}
\end{aligned}$$

for $j \in \{0, 1\}$. The claim immediately follows. □

*Proof of Theorem 6.* We do induction on the maximum code length of $S$. If it is 0, the claim can be calculated directly. If it is $n + 1$, then the maximum

code lengths of $\{x \mid i :: x \in S\}$ for $i \in \{0,1\}$ are less then or equal $n$, so by induction, we know that $\sum_{i::x\in S} 2^{-\operatorname{len} x} \leq 1$ for $i \in \{0,1\}$, and thus

$$\sum_{x\in S} 2^{-\operatorname{len} x} = \frac{1}{2} \sum_{i\in\{0,1\}} \sum_{i::x\in S} 2^{-\operatorname{len} x} \leq 1$$

Assume $\sum_{x\in S} 2^{-\operatorname{len} x} = 1$ and an $a$ existed such that $S \cup \{a\}$ is prefix-free. Then clearly $\sum_{x\in S} 2^{-\operatorname{len} x} > 1$, but we just proved $\sum_{x\in S} 2^{-\operatorname{len} x} \leq 1$. Contradiction.

Assume there was no $a$ such that $S \cup \{a\}$ is prefix-free. If the maximum code length of $S$ is 0, the claim can be calculated directly. Assume it is $n+1$. If there existed an $a_i$ such that $\{x \mid i :: x \in S\} \cup \{a_i\}$ was prefix-free, then $S \cup \{i :: a_i\}$ would also be prefix-free. Thus, such an $a_i$ does not exist for $i \in \{0,1\}$, and thus, by induction,

$$\sum_{x\in S} 2^{-\operatorname{len} x} = \frac{1}{2} \sum_{i\in\{0,1\}} \sum_{i::x\in S} 2^{-\operatorname{len} x} = \frac{1}{2} \sum_{i\in\{0,1\}} 1 = 1 \qquad \square$$

We furthermore show that a Deflate coding exists if and only if its length sequence satisfies Kraft's inequality. (From this, the direction 2. $\Rightarrow$ 1. follows immediately.)

**Theorem 7** (`extended_kraft_ineq`)**.** *Let $\lceil\cdot\rceil : \mathcal{A} \to [\{0,1\}]$ be a Deflate coding. Then*

$$\sum_{\substack{i\in\mathcal{A} \\ \lceil i\rceil \neq []}} 2^{-\operatorname{len}\lceil i\rceil} \leq 1$$

*Equality holds if and only if there is some $k \in \mathcal{A}$ such that $\lceil k\rceil \in \{1\}^+$.*

*Proof.* The first claim follows directly from Theorem 6, since $(\operatorname{img}\lceil\cdot\rceil)\backslash\{[]\}$ is a prefix-free set. For the second claim, it is sufficient to show that such a $k$ exists if and only if no $a \notin \operatorname{img}\lceil\cdot\rceil$ exists such that $((\operatorname{img}\lceil\cdot\rceil)\backslash\{[]\}) \cup \{a\}$ is prefix-free. So assume such a $k$ exists, but also such an $a$ existed. If $\lceil k\rceil \sqsubseteq a$, then $\lceil k\rceil \preccurlyeq a$, since $\lceil k\rceil$ is a sequence of 1's. If $a \sqsubseteq \lceil k\rceil$, then by axiom 4, a prefix of $a$ must already be in $\operatorname{img}\lceil\cdot\rceil$. This is a contradiction. For the other direction, assume equality holds, but no sequence of 1's is contained in $\operatorname{img}\lceil\cdot\rceil$. Let $m$ be the maximum code length of $\operatorname{img}\lceil\cdot\rceil$. Then all prefixes of $[1]^{m+1}$ are sequences of 1's, so it is not prefixed by any of the codes in $\operatorname{img}\lceil\cdot\rceil$. This contradicts Theorem 6. $\square$

This is proved in theorem `kraft_ineq` and theorem `extended_kraft_ineq` in the file `DeflateCoding.v`.

**Lemma 7.** *Let $d \in [\{0,1\}]$ and $d \neq [1]^{\operatorname{len} d}$. Then there is a $\sqsubseteq$-minimal $\overline{d} \in [\{0,1\}]$ with $\operatorname{len}\overline{d} = \operatorname{len} d$ and $d \lesssim \overline{d}$.*

Regarding $d$ as a binary number, $\overline{d} = d + 1$ does the job. Formally, this is proved as `Lemma lex_inc` in `Combi.v`. The most important theorem regarding Deflate codings is:

**Theorem 8.** *Let $l : \mathcal{A} \to \mathbb{N}$ be such that*

$$\sum_{\substack{i \in \mathcal{A} \\ l(i) \neq 0}} 2^{-l(i)} \leq 1$$

*Then there is a Deflate coding $\lceil \cdot \rceil : \mathcal{A} \to [\{0,1\}]$ such that $l = \lambda_x \operatorname{len} \lceil x \rceil$.*

*Proof.* Let $\lesssim$ be a binary relation on $\mathbb{N}^2$, defined by

$$\forall_{mqo}. q < o \to (q, m) \lesssim (o, m)$$

$$\forall_{m_1 m_2 n_1 n_2}. m_1 < m_2 \to (n_1, m_1) \lesssim (n_2, m_2)$$

It is easy to show that $\lesssim$ is a transitive decidable antisymmetric ordering relation. Now let $R = L := \operatorname{sort}_{\lesssim} \operatorname{map}_{\lambda_k(k,lk)}[0, \ldots, n-1]$, $S = [\,]$, $c = \lambda_x[\,]$.

We will do a recursion on tuples $(S, c, R)$, maintaining the invariants

$$\forall_q.(q, \operatorname{len}(c(q))) \notin S \to (c(q) = [\,] \wedge (q, l(q)) \in R) \tag{5.1}$$

$$(\operatorname{rev} S) +\!\!+ R = L \tag{5.2}$$

$$S = [\,] \vee \forall_q. c(q) \sqsubseteq c(\pi_1(\operatorname{car} S)) \tag{5.3}$$

Furthermore, $c$ will be a deflate coding at every step. The decreasing element will be $R$, which will become shorter at every step.

We first handle the simple cases:

- For the initial values $([\,], \lambda_x[\,], L)$, the invariants are easy to prove.
- For $R = [\,]$, we have $\operatorname{rev} S = L$ by 5.2 and therefore, either $c = \lambda_x[\,]$ if $L = [\,]$, or $\forall_q.(q, \operatorname{len}(c(q))) \in L$ by 5.1, and therefore, $c$ is the desired coding.
- For $R = (q, 0) :: R'$, $S$ can only contain elements of the form $(\_, 0)$. We proceed with $((q, 0) :: S, \lambda_x[\,], R')$. All invariants are trivially preserved.
- For $R = (q, 1 + l) :: R'$ and $S = [\,]$ or $S = (r, 0) :: S'$, we set $c'(x) = [0]^{1+l}$ for $x = q$, and $c'(x) = [\,]$ otherwise. We proceed with $((q, 1+l) :: S, c', R')$. The invariants are easy to show. It is easy to show that $c'$ is a Deflate coding.

Now the most general case is $R = (q, 1 + l) :: R'$ and $S = (r, 1 + m) :: S'$, and let the intermediate Deflate coding $c$ be given. We have

$$\sum_{\substack{i \in \mathcal{A} \\ c(i) \neq [\,]}} 2^{-\operatorname{len}(c(i))} < 2^{-l-1} + \sum_{\substack{i \in \mathcal{A} \\ c(i) \neq [\,]}} 2^{-\operatorname{len}(c(i))} \leq \sum_{\substack{i \in \mathcal{A} \\ li \neq 0}} 2^{-l(i)} \leq 1$$

By Theorem 7, $[1]^{1+m} \notin \operatorname{img} c$, and therefore, by Lemma 7, we can find a fresh code $d'$ of length $1+m$. Let $d = d' +\!\!+ [0]^{l-m}$ and set

$$c'(x) := \begin{cases} d & \text{for } x = q \\ c(x) & \text{otherwise} \end{cases}$$

We have to show that $c'$ is a Deflate coding. The axioms 2 and 3 are easy. For axiom 4, assume $x \neq [\,]$ and $x \sqsubseteq c'(q)$. If $x \sqsubseteq c'(r)$, the claim follows by axiom 4 for $r$. Otherwise, by totality $c'(r) \sqsubseteq x$. If $x \sqsubseteq d'$, by the minimality of $d'$ follows $x = c'(r)$. If $d' \sqsubseteq x$, trivially, $d' \preccurlyeq c'(q)$. Axiom 4 holds. For axiom 2, it is sufficient to show that no other non-$[\,]$ code prefixes $d$. Consider a code $e \preccurlyeq d$. As all codes are shorter or of equal length than $d'$, $e \preccurlyeq d'$. But then, either $e \preccurlyeq c(r)$, or $c(r) \sqsubseteq e$. Contradiction. Therefore, we can proceed with $((q, 1+l) :: S, c', r')$. □

The formal proof can be found as Theorem `existence` in `Deflate-Coding.v`. For a better understanding of the algorithm proposed here, we consider the following length function as an example:

$$l : 0 \to 2; 1 \to 1; 2 \to 3; 3 \to 3; 4 \to 0$$

We first have to sort the graph of this function according to the $\lesssim$ ordering.

$$[(4,0), (1,1), (0,2), (2,3), (3,3)]$$

Then, the following six steps are necessary to generate the coding (for the lack of space, we leave out the column stating that c(4) is always $[\,]$).

| Step | R | S | c(0) | c(1) | c(2) | c(3) |
|---|---|---|---|---|---|---|
| 0 | [(4, 0),<br>(1, 1),<br>(0, 2),<br>(2, 3),<br>(3, 3)] | [] | [] | [] | [] | [] |
| 1 | [(1, 1),<br>(0, 2),<br>(2, 3),<br>(3, 3)] | [(4, 0)] | [] | [] | [] | [] |
| 2 | [(0, 2),<br>(2, 3),<br>(3, 3)] | [(1, 1),<br>(4, 0)] | [] | [0] | [] | [] |
| 3 | [(2, 3),<br>(3, 3)] | [(0, 2),<br>(1, 1),<br>(4, 0)] | [1,0] | [0] | [] | [] |
| 4 | [(3, 3)] | [(2, 3),<br>(0, 2),<br>(1, 1),<br>(4, 0)] | [1,0] | [0] | [1,1,0] | [] |
| 5 | [] | [(3, 3),<br>(2, 3),<br>(0, 2),<br>(1, 1),<br>(4, 0)] | [1,0] | [0] | [1,1,0] | [1,1,1] |

The final values of $c$ are, in fact, a Deflate coding. The main difference to the algorithm in the standard [41] is that we sort the character/length pairs and then incrementally generate the coding, while the proposed algorithm counts the occurrences of every non-zero code length first, determines their lexicographically smallest code, and then increases these codes by one for each occurring character. In our case, that means that it would first generate the function $a : 1 \to 1; 2 \to 1; 3 \to 2$ and 0 otherwise, which counts the lengths, and then define

$$b(m) = \sum_{j=0}^{m-1} 2^j a(j)$$

which gets the numerical value for the lexicographically smallest code of every length when viewed as binary number with the most significant bit being the leftmost bit. In our case, this is $1 \to 0; 2 \to 2; 3 \to 6$. Then

$$c(n) = b(l(n)) + |\{r < n \mid l(r) = l(n)\}|$$

meaning $c(0) = b(2) = 10_{(2)}$, $c(1) = b(1) = 0_{(2)}$, $c(2) = b(3) = 110_{(2)}$, $c(3) = b(3) + 1 = 111_{(2)}$ which is consistent with the algorithm presented here.

The algorithm described in the standard [41] is more desirable for practical purposes, as it can make use of low-level machine instructions like bit shifting:

```
static int[] get_canonical_code (int[] lengths)                    {

    // check for kraft's inequality
    double kraft = 0                                               ;
    for (int i = 0; i < lengths.length; ++i)
        kraft += (1.0d / (1 << lengths[i]))                        ;
    if (kraft > 1) return null                                     ;

    // get the maximum code length
    int MAX_BITS = 0                                               ;
    for (int i = 0; i < lengths.length; ++i)
        if (MAX_BITS < lengths[i]) MAX_BITS = lengths[i]           ;

    // Count the number of codes for each code length except 0
    int[] bl_count = new int[MAX_BITS+1]                           ;
    for (int i = 0; i < lengths.length; ++i)
        bl_count[lengths[i]]++                                     ;
    bl_count[0] = 0                                                ;

    // Find the numerical value of the smallest code for each code
    // length
    int[] next_code = new int[MAX_BITS+1]                          ;
    int code = 0                                                   ;
    for (int bits = 1; bits <= MAX_BITS; ++bits)                   {
        code = (code + bl_count[bits-1]) << 1                      ;
        next_code[bits] = code                                    ;}

    // Assign numerical values to all codes
    int[] tree_code = new int[lengths.length]                      ;
    for (int n = 0; n < lengths.length; ++n)                       {
        int len = lengths[n]                                       ;
        if (len != 0)                                              {
            tree_code[n] = next_code[len]                          ;
            next_code[len]++                                      ;}}
    return tree_code                                               ;}
```

# Chapter 6

# Parsers from Constructive Proofs

Before we get into the actual Deflate-specific parts, we introduce the several concepts we created to be able to get a nice, usable and modular framework for specifying and using grammars, which we think will also be useful for further work on other data formats.

We specify a part of the format through a relation that relates the output to the input. Usually, output will be a list of bytes, and input will be a list of bits, but this is no requirement. The relations can have other parameters, but these parameters will always be before the output and input parameter.

## 6.1 Strong Uniqueness

In Deflate – and in most other formats – there is more than one way to represent the same thing. More specific, in [41], only recommendations for compression algorithms are given, but it is not required to use any specific algorithm – by design, Deflate is flexible, and a byte sequence can be compressed in several ways. However, for the other direction, it is different: We want, for every dataset in our format, a unique original dataset, meaning the guarantee that for any given data $d$, $\text{decompress}(\text{compress}\, d) = d$. While most container formats have some checksum or error correction code, Deflate itself does not have mechanisms to cope with data corruption due to hardware failures and transcription errors, therefore a formal discussion of these is outside the scope of this work, and we will focus on the correctness of the algorithms themselves in absence of external memory corruption. However, the concepts defined here should be applicable for such other formats, too. In summary, we want *left-uniqueness*.

Left-Uniqueness ("injectivity") can be formalized as $\forall_{a,b,l}.R\, a\, l \to R\, b\, l \to a = b$. However, when reading from a stream, it must always be clear when to "stop" reading, which essentially means that given an input $l$ such that

$R\,a\,l$, it cannot be extended: $\forall_{a,b,l,l'}.R\,a\,l \to R\,b\,(l +\!\!+ l') \to l' = [\,]$. In `StrongUniqueLemma` in the file `StrongDec.v`, we proved that these two properties are equivalent to the following property, which we call *strong uniqueness*:

$$\begin{aligned}\text{StrongUnique}\,R \quad :\Leftrightarrow \forall_{a,b,l_a,l'_a,l_b,l'_b}.\quad & l_a +\!\!+ l'_a = l_b +\!\!+ l'_b \to \\ & R\,a\,l_a \to R\,b\,l_b \to \\ & a = b \wedge l_a = l_b\end{aligned}$$

**Lemma 8.** $((\forall_{a,b,l,l'}.R\,a\,l \to R\,b\,(l +\!\!+ l') \to l' = [\,]) \wedge \forall_{a,b,l}.R\,a\,l \to R\,b\,l \to a = b) \leftrightarrow \text{StrongUnique}\,R$.

*Proof.* "$\to$": We have $l_a +\!\!+ l'_a = l_b +\!\!+ l'_b$, $R\,a\,l_a$ and $R\,b\,l_b$. Without loss of generality assume $\text{len}\,l_a \leq \text{len}\,l_b$. Then $l_b = l_a +\!\!+ l''_b$ for some $l''_b$. Therefore, $l''_b = [\,]$. Thus, $l_a = l_b$ and by uniqueness $a = b$. "$\leftarrow$": We have $R\,a\,l$ and $R\,b\,(l +\!\!+ l')$. Setting $l_a = l$, $l'_a = l'$, $l_b = l +\!\!+ l'$, $l'_b = [\,]$, strong uniqueness yields $a = b$ and $l = l +\!\!+ l'$, therefore $l' = [\,]$. □

## 6.2 Strong Decidability

While strong uniqueness gives us uniqueness of a prefix, provided that it exists, we need an additional property that states that it is actually decidable whether such a prefix exists, which we call *strong decidability*:

$$\begin{aligned}&\text{StrongDec}_E\,R :\Leftrightarrow \\ &\forall_l.(\Sigma_{a,x,y}.l = x +\!\!+ y \wedge R\,a\,x) + (E \times \neg(\exists_{a,x,y}.l = x +\!\!+ y \wedge R\,a\,x))\end{aligned}$$

This is a decidability predicate with the negated part tagged with some type $E$. This tag is formally not necessary, and is there for error messages; it is a string in our implementation, to get information of errors in the given compressed data, and as an additional documentation. The existential quantifiers in our implementation are strong: If a prefix exists, then we can compute it. This is the obvious type of a verified decoder. It resembles Haskell's Exception-monad. The extracted terms from strong decidability proofs are parsers.

**Lemma 9.** *Let* $R : A \to [B] \to \text{Prop}$. *If* $\text{StrongDec}\,R$ *and* $\text{StrongUnique}\,R$ *and* $A$ *has decidable equality, then* $R$ *is decidable.*

*Proof.* Let $a : A$ and $b : [B]$ be given. If $\neg(\exists_{c,x,y}.b = x +\!\!+ y \wedge R\,c\,x)$, then by uniqueness we know $\neg R\,a\,b$. Otherwise, we can check whether $y = [\,]$. If not, then by uniqueness we also know $\neg R\,a\,b$. Otherwise, we have to decide whether $a = c$. □

However, strong decidability is actually stronger than decidability: Let the inputs $[B]$ encode turing machines, and let $R : \mathbb{N} \to [B] \to \text{Prop}$ be the relation such that $R\,n\,b$ denotes "The turing machine encoded in $b$ stops after $n$ steps". We clearly have $\forall_{n,b}.R\,n\,b \vee \neg R\,n\,b$. However, strong decidability would imply solving the halting problem.

One drawback of strong decidability is that combining them consumes stack space, the same way Exception monads do: You usually have to recursively call the next sub-parser to know whether it succeeds, before you can return anything. Furthermore, it prevents lazy evaluation to some extent. A possible solution is proposed in Section 6.4.

A further drawback is that we cannot use cototal lists. This is for the simple reason that parsing potentially infinite lists is not strongly decidable anymore. For example, the relation $R : 1 \to [\text{bool}] \to \text{Prop}$ defined by

$$R\ \text{tt}\ [\text{true}]$$
$$\forall_{a,b,c}.R\,a\,b \to R\,a\,(c :: b)$$

holds if and only if the list contains true. This is strongly decidable for total lists, but clearly not for cototal lists: We can encode the halting problem in this problem: For an arbitrary machine $M$, define the list which is false in the $n$-th place if $M$ does not halt after $n$ steps, and true otherwise. Despite these drawbacks, strong decidability proved useful for writing our implementation of Deflate.

## 6.3 Relational Combinators

If a relation satisfies both properties, it is well-suited for parsing. We can combine such relations in a way similar to parser monads. When some of the relations we defined became very complicated and hard to read, we decided to use such monadic combinators. The advantage, besides readability, is that we could prove lemmata about those combinators which greatly simplified proving properties of our relations. We provide the functions `Combine` and `nTimes` and the corresponding lemmata in `StrongDec.v`. The main difference is that we do not have a disjunctive connective: There must be at most one possibility to decode a bit sequence, therefore, branches of a grammar need to be mutually exclusive. Proving mutual exclusivity is the main issue. Otherwise, we tried to use combinations of other relations as far as possible.

```
Inductive Combine {A BQ BR BS}
 (Q : BQ -> list A -> Prop)
 (R : BQ -> BR -> list A -> Prop)
 (c : BQ -> BR -> BS) : BS -> list A -> Prop :=
| doCombine : forall bq br aq ar,
   Q bq aq -> R bq br ar ->
     Combine Q R c (c bq br) (aq ++ ar).
```

```
Notation "A >>[ B ]= C" := (Combine A C B) (at level 0).
Notation "A >>= B" := (Combine A B pi2) (at level 0).
```

We try to make the intuition behind this relations clearer. Assume you already have functions `q : list A -> option (list A * BQ)` and `r : BQ -> list A -> option (list A * BR)`, which realize strong decidability for `Q` and `R`, respectively. Then the following function realizes strong decidability in the same way for `Q >>[ c ]= R`:

```
Function qr (l : list A) : option (list A * BS) :=
  match q l with
  | None => None
  | Some (lq, bq) => match r bq lq with
                     | None => None
                     | Some (lr, br) =>
                       Some (lr, c bq br)
                     end
  end.
```

The function `pi2` is the projection on the second argument. The `Q` argument is a relation that takes some input of type `list A`, and relates it to some output of type `BQ`, much like the relations we defined in Section 7.2. `R` is a function that maps some `BQ` to a relation between `BR` and `list A` – it will take the output of `Q` as a parameter, and produce itself some output of type `BR`. Finally, `c` combines these intermediate outputs to some common output type `BS`. Usually, `c = pi2` and `BR = BS`, that is, the intermediate result of type `BQ` is just dropped. The resulting relation first consumes some input using the first relation `Q`, then passes its output to the function `R`, which in turn consumes more of the input and produces some output `BR`. The final relation will be between the concatenation of the consumed lists and the intermediate results combined by `c`.

Admittedly, this relation is not trivial. But the pattern to read a small portion of the input, relate it to something, and make the rest dependent on that, frequently occurs, especially when reading format headers. So it seemed reasonable to abstract it out, so at least it only has to be understood once.

In many cases, it will occur that one just has to read something $n$ times, and combine the results in some way, that is, iteratively use the `Combine` function. In `StrongDec.v` we therefore define

```
Fixpoint nTimes {A B C} (n : nat) (null : C)
                        (comb : A -> C -> C)
                        (rel : A -> list B -> Prop)
: C -> list B -> Prop :=
  match n with
    | 0 => fun c L => c = null /\ L = nil
    | (S n') => rel >>[ comb ]= fun _ =>
                (nTimes n' null comb rel)
```

```
  end.
```

We usually do not need the full generality of this function, but defining it once in a most general manner makes us only having to prove its properties once. Special cases we use are appending or consing the outputs.

```
Definition nTimesApp {A B} (n : nat)
  (rel : list A -> list B -> Prop) :=
  nTimes n nil (@app A) rel.

Definition nTimesCons {A B} (n : nat)
  (rel : A -> list B -> Prop) :=
  nTimes n nil (@cons A) rel.
```

For definitions like `nTimes`, we need a base case, a "null" relation. While the output would not really matter, and we could just require `L = @nil B`, the resulting relation would not be unique anymore. To make it strong unique, we must define a default value `(fun n L => n = null /\ L = @nil B)`.

We also need a relation that applies a function to the first argument of the relation, prior to relating it:

```
Definition AppCombine {A BQ BR : Set }
  (Q : BQ -> list A -> Prop)
  (f : BQ -> BR) : BR -> list A -> Prop :=
  Combine Q (fun n (m : unit) L => m = () /\ L = @nil A)
            (fun a b => f a).
```

Since we can prove several properties about strong uniqueness and strong decidability, and since these combinators reflect common patterns that frequently occur, they are universal and should be usable for formally specifying other data formats, too.

It should be noted that we do not define a disjunctive connective. In unverified parser grammars, one can just interpret such a disjunctive connective by using the first branch that matches. However, in a formally verified implementation, we need the property that therer can be at most one branch that matches. This must be proved in every single case, and therefore we cannot lift it to this abstraction level. Usually, such branches correspond directly to the several constructors of a relation.

## 6.4 Streamable Strong Decidability

It is inevitable to consume stack if we want to have a function that returns an error if the input stream is malformed but not allow side effects. For example, it is also possible to write a program in continuation passing style, as it is done in [77], but this is just a transformation from the exception monad to the continuation monad, with the same problems. Most parser combinator libraries do it this way, since mostly the consumed stack is not

relevant for the size of the data which is parsed. For a format like Deflate, you would normally not use a parser combinator.

A general solution to stack overflows of such libraries, for example in Java, is to use an own stack-like structure to track the state. For example, the following is a recursive implementation of Flood Fill.

```
static void floodFill1 (int[][] image, int x, int y, int color)   {
    int orig = image[x][y]                                         ;
    if (orig == color) return                                      ;
    floodFill1_ (image, x, y, color, orig)                        ;}

static void floodFill1_ (int[][] image, int x, int y,
                         int color, int orig)                      {
    if ((x >= 0) && (y >= 0) &&
        (x < image.length) && (y < image[0].length) &&
        image[x][y] == orig)                                       {
        image[x][y] = color                                        ;
        floodFill1_(image, x-1, y, color, orig)                    ;
        floodFill1_(image, x+1, y, color, orig)                    ;
        floodFill1_(image, x, y-1, color, orig)                    ;
        floodFill1_(image, x, y+1, color, orig)                   ;}}
```

This will make result in a stack overflow for larger images. However, it is not trivially possible to not save the return path. In this case, the iterative way will use a `switch` statement:

```
static class XY                                                    {
    public int x, y, state                                         ;
    public XY(int _x, int _y, int _state)                          {
        x = _x; y = _y; state = _state                            ;}}

static void floodFill2 (int[][] image, int x, int y, int color)   {
    int orig = image[x][y]                                         ;
    if (orig == color) return                                      ;

    Stack<XY> st = new Stack<XY>()                                 ;
    st.push(new XY(x, y, 0))                                       ;

    while (true)                                                   {
        if (st.empty()) return                                     ;
        XY c = st.peek()                                           ;
        switch (c.state)                                           {
        case 0                                                     :
            if ((c.x >= 0) && (c.y >= 0) &&
                (c.x < image.length) && (c.y < image[0].length) &&
                image[c.x][c.y] == orig)                           {
                image[c.x][c.y] = color                            ;
                c.state = 1                                       ;}
            else                                                   {
                st.pop()                                  ;}break;
        case 1                                                     :
            st.push(new XY(c.x-1, c.y, 0))                         ;
            c.state = 2                                    ;break;
```

```
        case 2                                                  :
            st.push(new XY(c.x+1, c.y, 0))                      ;
            c.state = 3                                  ;break;
        case 3                                                  :
            st.push(new XY(c.x, c.y-1, 0))                      ;
            c.state = 4                                  ;break;
        case 4                                                  :
            st.push(new XY(c.x, c.y+1, 0))                      ;
            c.state = 5                                  ;break;
        case 5                                                  :
            st.pop()                                ;break;};}}
```

Clearly, this is just a way to put the stack to a higher abstraction level, which is slower but does not impose us the limitations of the machine-level stack. It cannot really overflow in the same sense as the machine-level stack does, but it can overflow in the sense that the machine runs out of memory. Therefore, this is the same as increasing the stack size.

Another possibility to avoid stack overflows is using exceptions. In Haskell, you could do the following:

```
data IncorrectDigit = NewIncorrectDigit deriving Show

instance Exception IncorrectDigit

ternary :: [Bool] -> Integer
ternary [] = 0
ternary (False : False : r) = 3 * ternary r
ternary (False : True : r) = 1 + 3 * ternary r
ternary (True : False : r) = 2 + 3 * ternary r
ternary (True : True : _) = throw NewIncorrectDigit
```

However, exceptions are impure: The result of a computation depends on the order of evaluation. Assume we defined an unsafe catch function using `unsafePerformIO`:

```
unsafeCatch :: Exception b => a -> (b -> a) -> a
unsafeCatch d q = unsafePerformIO $
  do z <- catch (do x <- evaluate d
                    return x)
                (\e -> do return (q e))
     return z
```

The `seq` function was made to enforce a certain evaluation order in cases where it is necessary due to limitations; specifically, the `foldl'` function, which enforces immediate folding of the given list instead of lazy folding which would potentially waste memory, can be implemented using `seq`:

```
foldl' f z []     = z
foldl' f z (x:xs) = let z_ = f z x
                    in z_ `seq` (foldl' f z_ xs)
```

However, in pure code, the order of evaluation must not matter, and therefore, `seq` is not allowed to change the outcome of any function. Now, if we define

```
e_x = let x = True
          y = error "Fail"
      in unsafeCatch (x `seq` Just (x || y))
                     ((\b -> Nothing) :: ErrorCall -> Maybe Bool)
```

it should be equivalent to

```
e_y = let x = True
          y = error "Fail"
      in unsafeCatch (y `seq` Just (x || y))
                     ((\b -> Nothing) :: ErrorCall -> Maybe Bool)
```

However, it is not. `e_x` returns `Just True`, and `e_y` returns `Nothing`. This shows that catching exceptions is impure. Exceptions in dependently typed programming languages are still under research. In [85] the following example is given to point out the problem:

$$Px^{\mathbb{N}} : \begin{cases} \mathbb{N} \to \mathbb{N} & \text{if } (\text{try } x \text{ catch } \lambda_\varepsilon.S0) = 0 \\ \mathbb{N} & \text{otherwise} \end{cases}$$

Now we have the reduction

$$P00 \to 0$$

however, we have the reduction

$$P(\text{throw}\,\varepsilon)0 \to 00$$

which is not well-typed. Until these problems are resolved, we mainly see two possibilities.

One possibility would be to allow for calling a given function if an exception occurs. This function would formally return a list which would be the tail of the returned stream, so the type checker is satisfied. On the other hand, we will mostly embed our verified functions into less strict languages, so we could make the passed function throw an exception. This would be possible, but it is not really satisfactory.

Another obvious solution would be to specify "erroneous" results that a parser can produce. For example, as we produce lists, we could define an alternative list type which has an additional constructor which indicates an error:

```
Inductive EList (A E : Set) : Set :=
  NilOK : EList A E
| NilError : E -> EList A E
| ECons : A -> EList A E -> EList A E.
```

This allows us to write an algorithm that can produce an error at every time, but makes it necessary to read the list entirely before we know whether there was an error. The following will consume much stack:

```
Fixpoint EListToExc (A E : Set) (l : EList A E) : (list A + E) :=
  match l with
  | NilOK _ _ => inl []
  | NilError _ _ x => inr x
  | ECons _ _ a r =>
    match EListToExc A E r with
    | inl l_ => inl (a :: l_)
    | inr e => inr e
    end
  end.
```

This is essentially what we did with strong decidability. What we actually want to do is to postpone the evaluation of the error as far as possible, to preserve laziness. We notice that `EList A E` is isomorphic to `list A * option E`:

```
Fixpoint EListToPair (A E : Set) (e : EList A E)
  : (list A * option E) :=
  match e with
  | NilOK _ _ => ([], None)
  | NilError _ _ x => ([], Some x)
  | ECons _ _ a r =>
    let (l, x) := EListToPair A E r
    in (a :: l, x)
  end.
```

To convince ourselves that this code will not break laziness in Haskell, we wrote the following Haskell-Code:

```
import System.Environment

lrec :: [Maybe a] -> ([a], Int)
lrec [] = ([], 0)
lrec (Nothing : l) = ([], -1)
lrec (Just a : l) = let (q, r) = lrec l
                    in (a : q, r + 1)

evald = lrec (map Just [1..])

lazytest :: [Int]
lazytest = let (r, _) = evald
           in take 10 r

lazytest2 :: Int
lazytest2 = let (r, b) = evald
           in b

main :: IO ()
main =
    do args <- getArgs
```

```
putStr $ show $
 [Left lazytest, Right lazytest2] !!
   (read $ args !! 0)
```

Calling this program with 0 as argument shows a finite list, while calling it with 1 as argument will result in an endless recursion, meaning that for computation of the list, the error itself is not evaluated.

We therefore define $R : A \to [B] \to \text{Prop}$ to be *streamably strongly decidable* by

$$\begin{aligned}\text{SSD}_E\, R := \quad & \forall_{l^{[B]}} \Sigma_{a^A, l_1^{[B]}, l_2^{[B]}, e^{1+E}}. \\ & l = l_1 +\!\!+ l_2 \wedge (e = \text{inl}() \to R\, a\, l_1) \wedge \\ & (e \neq \text{inl}() \to \forall_{b,k_1,k_2}. l = k_1 +\!\!+ k_2 \to R\, b\, k_1 \to \bot)\end{aligned}$$

**Lemma 10.** *Formally, strong decidability and streamable strong decidability are equivalent.*

*Proof.* Let $R$ be streamably strong decidable and an input $l^{[B]}$ be given. By eliminating $\text{SSD}_E\, R\, l$, we get $a^A$, $l_1^{[B]}$, $l_2^{[B]}$ and $e^{1+E}$. We eliminate $e$. Assuming $e = \text{inl}()$, we know that $R\, a\, l_1$ and $l = l_1 +\!\!+ l_2$. Thus, by re-introducing the quantors, we get $\text{StrongDec}\, R$. Assuming $e = \text{inr}\, f$, we know that $\forall_{b,k_1,k_2}. l = k_1 +\!\!+ k_2 \to R\, b\, k_1 \to \bot$, and thus, $\text{StrongDec}_E\, R$.

Now let $R$ be strongly decidable, and an input $l^{[B]}$ be given. We eliminate $\text{StrongDec}_E\, R\, l$. If the left side is satisfied, we get $a^A$, $l_1^{[B]}$, $l_2^{[B]}$ and $l = l_1 +\!\!+ l_2 \wedge R\, a\, l_1$, so we can re-introduce the quantors and get $\text{SSD}_E\, R$, with $e = \text{inl}()$. If the right side is satisfied, we get the error $f^E$, and so we set $e = \text{inr}\, f$. We furthermore can set $l_1 = [\,]$ and $l_2 = l$ to get the guarantee that $l = l_1 +\!\!+ l_2$. We can then re-introduce the quantors and get $\text{SSD}_E\, R$. □

As we proved strong decidability for all relations already, for the simple cases, we will just apply Lemma 10 to our old proofs. This shows how our flexible and modular approach pays off when changing definitions later. We proved the directions of this lemma separately in the file `EncodingRelationProperties2.v` as `SD_imp_SSD` and `SSD_imp_SD`.

For example, for `OneBitSSD`, the definition is

```
Lemma OneBitSSD : StreamableStrongDec string OneBit.
Proof.
  apply (SD_imp_SSD false).
  apply OneBitStrongDec.
Defined.
```

It does only require a neglegible amount of stack to parse, which is why it is sufficient to use this lemma instead of proving it directly.

Due to the lack of time, we did not finish proving all relevant streamable strong decidabilities. Our implementation therefore still consumes lots of memory. We regard this as further work.

# Chapter 7

# The Encoding Relation

In this section, we want to present the relation we gave in Coq, which is the core of our implementation. Our implementation is verified against it, and it is part of the trusted codebase: It is impossible to verify that our relation is "correct" – it is not even clear what that means, since the standard [41] is informally stated, and therefore subject to interpretation. However, we tested our implementation, and it is unlikely that there is a problem.

## 7.1 Overview

We give a short informal overview of correct Deflate streams, to show you the complexity of the format, and in the hope that it will make it easier to follow our definitions and relations. Notice that we are describing an existing and widespread standard here. Especially, this standard was not made by us. We are giving this overview so you do not have to read the actual standard. There are many parts which seem overcomplicated, but that is probably due to the fact that this is a grown standard.

We start with a small informal pseudogrammar for Deflate streams for some clarity:

```
Deflate             ::=  ('0' Block)*
                         '1' Block ( '0' | '1' )*
Block               ::=  '00' UncompressedBlock |
                         '01' DynamicallyCompBl |
                         '10' StaticallyCompBl
UncompressedBlock   ::=  length ~length bytes
StaticallyCompBl    ::=  CompressedBlock(standard coding)
DynamicallyCompBl   ::=  header coding
                         CompressedBlock(coding)
CompressedBlock(c)  ::=  [^256]* 256 (encoded by c)
```

A first thing to notice is that a compressed block ends with the number 256 – which means that compressed blocks' alphabet does not only contain bytes. This is possible because it is encoded by a prefix-free coding in which characters may be encoded by more than 8 bits. Notice furthermore that

this grammar does not in any way give information on how the data is to be interpreted. More specifically, the whole machinery of resolving backreferences and encoding codings is not reflected in this informal grammar.

To clarify our terminology, we say a *character* is an element from an alphabet, a *codepoint* is a number that is encoded in some dataset and may stand for either a character or some instructional control structure, a *coding* is a function that assigns bit sequences to codepoints, and a *code* is a bit sequence which is associated with some codepoint through a coding.

Deflate streams can make use of three techniques of compression: prefix-free coding (as in Huffman codes), run length encoding and backreferences as found in Lempel-Ziv-compression. The latter two use the same mechanism, as described in Section 7.4. Furthermore, Deflate streams are byte streams, which are streams of values from 0 to 255. With such byte streams, one associates bit streams by concatenating the bytes LSB (least significant bit first), regardless of how they are actually sent. This is necessary, because most Deflate modes operate conceptually on the bit level.

On top of this bit stream, the data is sliced into blocks which may be compressed. A block starts with a bit that indicates whether it is the last block, and two further bits indicating whether the block is "statically" compressed, that is, with fixed codings defined in the standard, or "dynamically" compressed, where the codings must be saved, or uncompressed.

For an uncompressed block, the bits up to the next byte boundary are ignored, then a 16 bit integer followed by its bitwise complement are saved byte aligned. It denotes the number of bytes the block contains. Uncompressed blocks cannot contain backreferences. The advantage of the byte aligned layout of uncompressed blocks is that it allows for the use of byte-wise access, e.g. `sendfile(2)`. On the formal level this brings the extra difficulty that Deflate streams cannot be described as a formal grammar on a bit sequence without knowing the byte boundaries.

Compressed blocks start immediately after the three header bits. Statically compressed blocks have predefined codings, and therefore, the compressed data immediately follows the header bits. Even when the actual compression does not utilize Huffman codings to save memory directly (which will usually be the case for statically compressed blocks), two prefix-free codings are needed to encode backreferences: A coding does not only encode the 256 byte-values, but up to 286 (288 with 2 forbidden) characters, of which one, 256, is used as end mark, and the values above 256 are used to denote backreferences. If the decoder encounters a code for such a character, a certain number of additional bits is read, from which the length of this backreference is calculated. Then, using another coding, a value from 0 to 29 is read, and additional bits, which determine the distance of that backreference. The numbers of actual bits for characters can be looked up in a table specified in the standard [41].

Dynamically compressed blocks get another header defining the two De-

flate codings. The codings are saved as sequences of numbers, as formalized in Section 5. This way is similar to other compression standards that utilize prefix-free codings, like BZip2. These sequences are themselves compressed, and another header is needed to save their coding.

For clarity, let us consider a small example. As we have to deal with three layers of compression, it is not always clear what a code, a coding and a character is. For this example, we add indices to the words to denote which layer they are from. A $\text{code}_n$ is a sequence of bits for a $\text{codepoint}_n$. A $\text{codepoint}_n$ is a number assigned to either a $\text{character}_n$ or some special instruction on that level. A $\text{coding}_n$ is a deflate coding for $\text{codepoints}_n$. Raw bytes are $\text{characters}_0$. We want to compress the string

$$ananas\_banana\_batata$$

Firstly, as we want to compress, we need an end sign (which gets the $\text{codepoint}_0$ 256), which we will denote as $\emptyset$. Since this string has a lot of repetitions, we can use backreferences. A backreference is a pair $\langle l, d\rangle$ of length and distance, which can be seen as an instruction to copy $l$ bytes from a backbuffer of decompressed data, beginning at the $d$-th last position, to the front, in ascending order, such that the copied bytes may be bytes that are written during the resolution of this backreference, hence allowing for both deduplication and run length encoding. In our case, we can add two backreferences.

$$an\ \langle 3,2\rangle\ s\_b\ \langle 5,8\rangle\ \langle 3,7\rangle\ t\ \langle 3,2\rangle\ \emptyset$$

The $\text{codepoint}_0$ for length 3 is 257, and for 5 it is 259. They do not have suffixes. The $\text{codepoint}_0$ for the distance 2 is 1 with no suffix, for 7 and 8 it is 5, and it has a single bit as suffix, which indicates whether it stands for 7 or 8. We write $a^l$ to denote that $a$ is a literal/length $\text{codepoint}_0$, with an index denoting the corresponding $\text{character}_0$ if any, and $a^d$ to denote that it is a distance $\text{codepoint}_0$. We furthermore put suffix bit sequences in brackets. Then we get

$$97^l_a\ 110^l_n\ 257^l 1^d\ 115^l_s\ 95^l_{\_}\ 98^l_b\ 259^l 5^d(1)\ 257^l\ 5^d(0)\ 116^l_t\ 257^l 1^d\ 256^l_\emptyset$$

The frequencies of literal/length $\text{codepoints}_0$ are

$$95\times 1; 97\times 1; 98\times 1; 110\times 1; 115\times 1; 116\times 1; 256\times 1; 257\times 3; 259\times 1$$

The frequencies of distance $\text{codepoints}_0$ are

$$1\times 2; 5\times 2$$

The optimal deflate $\text{codings}_0$ (as defined in Section 5) are

$$95\to 1100; 97\to 010; 98\to 011; 110\to 100; 115\to 1101$$

$$116 \to 101; 256 \to 1110; 257 \to 00; 259 \to 1111$$

and

$$1 \to 0; 5 \to 1$$

To clarify the terminology, note that e.g. character$_0$ `a` has codepoint$_0$ 010 under the given coding$_0$. The reason for introducing the concept of "codepoints$_0$" is that the alphabets for lengths and characters$_0$ are merged: Every character$_0$ has an assigned codepoint$_0$, but not every codepoint$_0$ has a character$_0$, e.g. the codepoint$_0$ 257 indicates a backreference, but still has the code$_0$ 00. Our message can therefore be encoded by the following sequence of bits (spaces are included for clarity):

```
010 100 00 0 1101 1100 011 1111 1 1 00 1 0 101 00 0 1110
```

As proved in Section 5, it is sufficient to save the lengths, which is done as a run length encoded list, where length 0 means that the corresponding codepoint$_0$ does not occur. We use a semicolon to separate the literal/length coding$_0$ from the distance coding$_0$. Both lists are not separated in the actual file, and it is even allowed that run-length-encoding-instructions spread across their border. What part of the unfolded list belongs to which coding is specified in another header defined later.

$$\underbrace{0, \ldots, 0}_{95\times}, 4, 0, 3, 3, \underbrace{0, \ldots, 0}_{11\times}, 3, 0, 0, 0, 4, 3, \underbrace{0, \ldots, 0}_{138\times}, 0, 4, 2, 0, 4; 0, 1, 0, 0, 0, 1$$

This list will itself be compressed, thus, the lengths of codes$_0$ become characters$_1$. Notice that due to a header described later, we can cut off all characters$_1$ after the last nonzero character$_1$ of both sequences. The maximum length that is allowed for a code$_0$ in deflate is 15. Deflate uses the codepoints$_1$ $16, 17, 18$ for its run length encoding. Specifically, 17 and 18 are for repeated zeroes. 17 gets a 3 bit suffix ranging from 3 to 10, and 18 gets a 7 bit suffix, ranging from 11 to 138. These suffixes are least-significant-bit first. The former sequence therefore becomes

$$18(0010101), 4, 0, 3, 3, 18(0000000), 3, 17(100),$$

$$4, 3, 18(1111111), 0, 4, 2, 0, 4; 0, 1, 17(000), 1$$

Now, the frequencies of codepoints$_1$ are

$$0 \times 4; 1 \times 2; 2 \times 1; 3 \times 4; 4 \times 4; 17 \times 1; 18 \times 2$$

Therefore, the optimal coding$_1$ is

$$0 \to 100; 1 \to 1110; 2 \to 1111; 3 \to 00; 4 \to 01; 17 \to 101; 18 \to 110$$

The sequence of code$_0$ lengths can therefore be saved as

```
110 0010101 01 100 00 00 110 0000000 00 101 100 01 00
110 1111111 100 01 1111 100 01 100 1110 101 000 1110
```

We now have to save the $\text{coding}_1$, and again, it is sufficient to save the $\text{code}_1$ lengths. These $\text{code}_1$ lengths for the 19 $\text{codepoints}_1$ are saved as 3 bit least-significant-bit first numbers, but in the following order: $16, 17, 18, 0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15$. Again, the $\text{codepoint}_2$ 0 denotes that the corresponding $\text{codepoint}_1$ does not occur. We can furthermore cut off the $\text{codepoint}_2$ for the last $\text{code}_1$ length (in the given order), 15, which is 0 in our example, due to a header described later. The sequence of $\text{codepoints}_2$ therefore becomes

```
000 110 110 110 000 000 000 000 000
000 000 010 000 010 000 001 000 001
```

We now come to the aforementioned header that in particular allows us to economize trailing zeroes. We need the number of literal/length $\text{codepoints}_0$ and distance $\text{codepoints}_0$ saved in the former sequence, and the number of saved $\text{codepoints}_1$. These are 260, 6 and 18, respectively. The first one is saved as a 5 bit number ranging from 257 to 286 (the values 287 and 288 are forbidden), the second one is saved as a 5 bit number ranging from 1 to 32, the third one is saved as a 4 bit number ranging from 4 to 19. Therefore, this header becomes

```
11000 10100 0111
```

With three additional header bits, denoting that what follows is the last block, and that it is a dynamically compressed block, (and with 7 additional bits to fill up the byte in the last line) we get

```
1 0 1
11000 10100 0111

000 110 110 110 000 000 000 000 000
000 000 010 000 010 000 001 000 001

110 0010101 01 100 00 00 110 0000000 00 101 100 01 00
110 1111111 100 01 1111 100 01 100 1110 101 000 1110

010 100 00 0 1101 1100 011 1111 1 1 00 1 0 101 00 0 1110

0000000
```

Of course, this example is constructed for instructional purposes, and the compressed message is longer than the original text. However, Deflate also supports statically compressed blocks, which are good for repetitive files. Those use a fixed $\text{coding}_0$ which is completely described in the standard [41].

Its relevant part for our example is the following:

$$95^l \to 10001111; 97^l \to 10010001; 98^l \to 10010010; 110^l \to 10011110;$$
$$115^l \to 10100011; 116^l \to 10100100; 256^l \to 0000000; 257^l \to 0000001;$$
$$259^l \to 0000011; 1^d \to 00001; 5^d \to 00101$$

With the three header bits, and 4 additional padding bits to fill the byte, the compressed file is

```
1 1 0

10010001 10011110 0000001 00001 10100011 10001111 10010010
0000011 00101 1 0000001 00101 0 10100100 0000001 00001
0000000

0000
```

which is, in fact, slightly shorter than the original string.

## 7.2 The Toplevel Relation

The toplevel relation is

```
Inductive DeflateEncodes (out : LByte) (inp : LB) : Prop :=
| deflateEncodes : forall swbr,
                     ManyBlocks 0 swbr inp ->
                     ResolveBackReferences swbr out ->
                     DeflateEncodes out inp.
```

where `inp` is a list of bits, `out` is a list of output bytes. This relation relates a compressed data stream to its cleartext. Firstly, it refers to `ManyBlocks`, which has a bit list as an input, and a `SequenceWithBackRefs` as its output: All other block relations produce such sequences with backreferences, because the resolution of backreferences is not bound to block boundaries. These Backreferences must be resolved, and then one gets the correct output.

```
Inductive ManyBlocks : nat -> SequenceWithBackRefs Byte
                                          -> LB -> Prop :=
| lastBlock : forall n inp out,
                OneBlockWithPadding out (n + 1) inp ->
                ManyBlocks n out (true :: inp)
| middleBlock : forall n inp1 inp2 out1 out2,
    OneBlockWithPadding out1 (n + 1) inp1 ->
    ManyBlocks (n + 1 + ll inp1) out2 inp2 ->
    ManyBlocks n (out1 ++ out2) (false :: inp1 ++ inp2).
```

This relation has an additional first argument. It is a natural number. It tracks the number of bits that have been read from the compressed input. It is increased at every recursive call by the respective number of bits that have

been consumed. The reason is that while the two kinds of compressed blocks can start at any bit position, uncompressed blocks cannot: They have to start at a byte boundary. Our way of reflecting this is tracking the number of consumed bits and use this information in the respective relation to only allow it at multiples of 8. This relation is called by `DeflateEncodes` with the argument 0, obviously. Besides that, this relation distinguishes between `lastBlock` and `middleBlock` and adds a header bit accordingly. This is the bit called `BFINAL` in [41].

```
Inductive OneBlockWithPadding
  (out : SequenceWithBackRefs Byte) : nat -> LB -> Prop :=
| obwpDCB : forall dcb n,
  DynamicallyCompressedBlock out dcb ->
   OneBlockWithPadding out n (false :: true :: dcb)
| obwpSCB : forall scb n, StaticallyCompressedBlock out scb ->
   OneBlockWithPadding out n (true :: false :: scb)
| obwpUCB : forall ucb n m pad,
   UncompressedBlockDirect out ucb ->
   n + ll (false :: false :: pad) = 8 * m ->
   ll pad < 8 ->
   OneBlockWithPadding out n (false :: false :: pad ++ ucb).
```

All the branches of this relation add the two header bits that indicate the type of block that follows, according to [41]. Furthermore, like `ManyBlocks`, this relation has a natural argument which is the number of bits that have been consumed. Only the branch `obwpUCB`, which is for uncompressed blocks, actually uses this argument, and takes care that the data begins at a byte-boundary, by adding an arbitrarily choosable padding string (which will usually consist of zeroes).

## 7.3 Uncompressed Blocks

It might seem trivial to read uncompressed blocks, but the input argument of our relations is – for practical reasons – always a bit list, while the output is a sequence of backreferences and bytes. That is, we disassemble the bytes into a bit list first, and for the case of uncompressed blocks, we have to combine the bits into bytes again. We originally tried to solve this problem differently, by having a byte list or some more sophisticated structure as input.

For example, one such idea was to utilize lazyness and graph reduction and define a structure

```
data BitsAndBytes =
  Bit Bool Word8 BitsAndBytes BitsAndBytes |
  End deriving Show

toBitsAndBytes :: [Word8] -> BitsAndBytes
toBitsAndBytes [] = End
```

```
toBitsAndBytes (byte : rest) =
  let toBits :: Word8 -> [Bool]
      toBits w = map (testBit w) [0..7]
      toBB :: [Bool] -> Word8 -> BitsAndBytes -> BitsAndBytes
      toBB [] _ bb = bb
      toBB (b : r) w bb = Bit b w (toBB r w bb) bb
  in toBB (toBits byte) byte (toBitsAndBytes rest)
```

therefore, saving a pointer to the next byte at every cell. This would probably have been an elegant way for a Haskell implementation doing Lazy I/O. However, it is formally hard to capture the semantics of this structure, and it will result in a massive blowup in the absence of graph reduction. In the end, we decided to have a bit list as input, and count the number of bits that have been consumed, as shown in Section 7.2. This leaves us with some extra work for uncompressed blocks, but it greatly simplifies the relations that do work on the bit level.

The problem when reading a byte is that we defined bytes to be bit-vectors. We need to assure Coq that these vectors are indeed of length 8, which is a bit tedious.

For reading a single bit and just return it, we have the relation

```
Inductive OneBit : bool -> LB -> Prop :=
| oneBit : forall b, OneBit b [b].
```

We then simply use our combinator function to read `n` bits into a list

```
Definition nBits (n : nat) : LB -> LB -> Prop :=
  nTimesCons n OneBit.
```

Using this relation, we can define a relation that reads `n` bits into a vector

```
Definition nBitsVec (n : nat) (vb : vec bool n) (l : LB)
: Prop := nBits n (to_list vb) l.
```

We use `vec bool 8` as representation for bytes:

```
Notation Byte := (vec bool 8).
```

and can then define

```
Definition OneByte : (Byte + nat*nat)%type -> LB -> Prop :=
  AppCombine (nBitsVec 8) inl.
```

We can now specify reading multiple bytes:

```
Definition nBytesDirect (n : nat)
: SequenceWithBackRefs Byte -> LB -> Prop :=
  nTimesCons n OneByte.
```

Uncompressed blocks start with two 16 bit numbers, where the second one is the complement of the first one, and the first one denotes the length. We can therefore define:

```
Definition UncompressedBlockDirect
  : SequenceWithBackRefs Byte -> LB -> Prop :=
  (readBitsLSB 16) >>=
  fun len  => (readBitsLSB 16) >>=
  fun nlen =>
    (fun swbr lb => len + nlen = 2 ^ 16 - 1 /\
                    nBytesDirect len swbr lb).
```

## 7.4 Backreferences

Compressed blocks can make use of run length encoding and backreferences as found in Lempel-Ziv-compression, being merged into one mechanism, which we will refer to as backreferences. In `Backreferences.v` we made the definition

```
Function SequenceWithBackRefs A := (list (A+(nat*nat))%type).
```

An element is therefore either an element of some abstract alphabet `A`, or a backreference, which is a pair $\langle l, d \rangle$ of a length $l$ and a distance $d$. The length is the number of bytes to be copied, the distance is the number of bytes in the backbuffer that has to be skipped before copying. Assuming we wanted to compress the string

$$ananas\_banana\_batata \tag{7.1}$$

we could shorten it with backreferences to

$$ananas\_b\,\langle 5, 8\rangle\,\langle 3, 7\rangle\,tata \tag{7.2}$$

An intuitive algorithm to resolve such a backreference would be a loop that decreases the length and copies one byte from the backbuffer to the front each time (the example is written in Java):

```
  int resolve (int len, int dist, int pointer, byte[] output)    {
    while (len > 0)                                              {
      output[pointer] = output[pointer-dist]                     ;
      pointer = pointer + 1                                      ;
      len = len - 1                                             ;}
    return pointer                                              ;}
```

This intuitive algorithm would also work when $l > d$, and result in a repetition of already written bytes – which is what run length encoding would do. Therefore, Deflate explicitly allows $l > d$, allowing us to shorten (7.2) even further:

$$an\,\langle 3, 2\rangle\,s\_b\,\langle 5, 8\rangle\,\langle 3, 7\rangle\,t\,\langle 3, 2\rangle \tag{7.3}$$

More directly, the string $aaaaaaaargh!$ can be compressed as $a\,\langle 7, 1\rangle\,rgh!$, which essentially is run length encoding.

For resolution, we inductively define

```
Inductive nthLast {A : Set}
 : forall (n : nat) (L : list A) (a : A), Prop :=
| makeNthLast : forall L M a, nthLast (ll (a :: M))(L ++ a :: M) a.
```

It is easy to see that this relation formalizes a step of our above algorithm (`ll` stands for list length). The relation `ResolveBackReference` uses it to resolve a single backreference, and is used by `ResolveBackReferences` for complete resolution of backreferences. As a Lemma, we proved an example from [41] in `Backreferences.v`:

```
(** Example from RFC Page 10 with X = 1 and Y = 2*)
Goal ResolveBackReference 5 2 [1; 2] [1; 2;  1; 2; 1; 2; 1]
                          (** = [1; 2] ++ [1; 2; 1; 2; 1] *).
```

## 7.5 Compressed Blocks

There are two types of compressed blocks: One with fixed codings, and one where the coding is given in an additional header. They share a lot of definitions, and we put some patterns that frequently occur in common definitions.

### 7.5.1 Compressed Code with Extra Bits

Having some element of a coding being followed by a sequence of bits with a defined length is a pattern that occurs in multiple places. This pattern is hard to formalize in a readable fashion. So we decided to define one relation that can handle this pattern in every case that occurs, which made it even less readable, but is only one relation – we decided that one less readable definition is better than four of them. We give the relation first, and then explain its various parts:

```
1  Inductive CompressedWithExtraBits
2         {m : nat} (coding : deflateCoding m)
3         (mincode : nat) (xbitnums bases maxs : list nat)
4         : nat -> LB -> Prop :=
5  | complength
6    : forall (base extra code max xbitnum : nat) (bbits xbits : LB),
7       dc_enc coding (mincode + code) bbits -> (* code >= mincode *)
8       nth_error xbitnums code = Some xbitnum->(* # of addit. bits *)
9       nth_error bases code = Some base -> (* base *)
10      nth_error maxs code = Some max -> (* maximum *)
11      ll xbits = xbitnum -> (* addit. bits have specified length *)
12      LSBnat xbits extra -> (* binary number made by xbits *)
13      base + extra <= max ->
14      CompressedWithExtraBits coding mincode xbitnums
15                      bases maxs (base + extra) (bbits ++ xbits).
```

During usual compressed blocks, the alphabets of bytes and backreference instructions are merged. Our relation only wants to handle the codepoints that can have a suffix and are not just encoding an alphabet. That is, there is a minimal code that we want to accept. It is the `mincode` parameter of our relation. In line 7, you can see the `dc_enc` relation which says that the given coding encodes the given codepoint with the given bits. It uses `mincode + code`, which makes it impossible to match a code less than `mincode`. This also explains the `code` variable: It is the difference between the actual codepoint and the minimal code.

It is used as an index for the lists `xbitnums`, `bases` and `maxs`. The `(n - mincode)`-th element of the list `xbitnums` contains the number of extra bits that the given codepoint `n` gets. In line 8, this number is read into `xbitnum`, which then contains the number of extra bits for the given code that was encoded by `bbits`. The actual bits that are in the suffix are in the list `xbits`, and in line 11, it is assured that it has apropriate length.

The relation `LSBnat` relates a bit list to a natural number, by interpreting it as binary least-significant-bit-first number. In line 12, it is used to define `extra` to be the number encoded by the additional bits. This number is usually not the actual number that one wants to encode: There is a `base` for every codepoint, defined in the list `bases`, which is read. The actual number that is encoded is then `base + extra`, as it is specified in line 15. As an additional gotcha, it is not always allowed to use the entire range of values: The codepoint 284 in the usual literal alphabet, see [41], page 11, gets 5 additional bits, which would make it possible to encode 32 values; however, it can only encode 30 values. As we need to be able to forbid the other values, we have the `maxs` list, which saves the upper bounds of encoded numbers. The maximum is looked up in line 10, and checked in line 13.

This relation, though complicated, reflects all the needs for this pattern in any place it occurs. We will use it multiple times in the following, so make sure you understood it.

### 7.5.2 Compressed Data

If you look at the RFC [41], Page 11, you will find the following table:

| Code | Extra Bits | Length(s) | Code | Extra Bits | Lengths | Code | Extra Bits | Length(s) |
|---|---|---|---|---|---|---|---|---|
| 257 | 0 | 3 | 267 | 1 | 15,16 | 277 | 4 | 67-82 |
| 258 | 0 | 4 | 268 | 1 | 17,18 | 278 | 4 | 83-98 |
| 259 | 0 | 5 | 269 | 2 | 19-22 | 279 | 4 | 99-114 |
| 260 | 0 | 6 | 270 | 2 | 23-26 | 280 | 4 | 115-130 |
| 261 | 0 | 7 | 271 | 2 | 27-30 | 281 | 5 | 131-162 |
| 262 | 0 | 8 | 272 | 2 | 31-34 | 282 | 5 | 163-194 |
| 263 | 0 | 9 | 273 | 3 | 35-42 | 283 | 5 | 195-226 |
| 264 | 0 | 10 | 274 | 3 | 43-50 | 284 | 5 | 227-257 |
| 265 | 1 | 11,12 | 275 | 3 | 51-58 | 285 | 0 | 258 |
| 266 | 1 | 13,14 | 276 | 3 | 59-66 | | | |

After a brief look, one could think that the range of encoded lengths is always 2 to the power of the number of bits. Therefore, one might have the idea to use some clever function to calculate these bounds. However, a closer look at code 284 shows that here, for some reason we do not know, only 30 encoded lengths are allowed. We therefore refrained from using any formula to directly calculate the value, and just copied the min- and max-values from the tables. We initially did not have such a max-value that limits the allowed encoded lengths, because we did not notice this non-continuity. This is one example of a possible source of bugs and the strangeness of the standard.

We now look at the data part of the compressed blocks, that is, everything besides the headers. This part is common to all compressed blocks.

```
Function CompressedLength (litlen : deflateCoding 288) :=
 CompressedWithExtraBits
   litlen 257 repeatCodeExtraBits repeatCodeBase repeatCodeMax.
Function CompressedDist (dist : deflateCoding 32) :=
 CompressedWithExtraBits
  dist 0 distCodeExtraBits distCodeBase distCodeMax.
```

These are helper relations that define when a length and a distance are compressed. The lists of extra bit numbers, bases and maximally allowed encoded numbers are defined in `EncodingRelation.v`, they were directly taken from the lists in [41].

```
Inductive CompressedSWBR
 (litlen : deflateCoding 288) (dist : deflateCoding 32)
 : SequenceWithBackRefs Byte -> LB -> Prop :=
| cswbr_end : forall l, dc_enc litlen 256 l ->
                         CompressedSWBR litlen dist [] l
| cswbr_direct : forall prev_swbr prev_lb l n,
                   dc_enc litlen (ByteToNat n) l ->
                   CompressedSWBR litlen dist prev_swbr prev_lb ->
                   CompressedSWBR litlen dist ((inl n):: prev_swbr)
                                              (l ++ prev_lb)
| cswbr_backref : forall prev_swbr prev_lb l d lbits dbits,
                    CompressedSWBR litlen dist prev_swbr prev_lb ->
                    CompressedLength litlen l lbits ->
```

```
                    CompressedDist dist d dbits ->
                    CompressedSWBR litlen dist
                                   ((inr (l, d)):: prev_swbr)
                                   (lbits ++ dbits ++ prev_lb).
```

Two codings, a literal/length coding and a distance coding, are associated with this relation. The first constructor `cswbr_end` just expresses that if the input encodes 256 according to the literal/length coding, the encoded list is just the empty list, because the codepoint 256 means that the block ends here. The second constructor `cswbr_direct` says that if the literal/length coding encodes a number below 256, that is, it encodes a byte `n`, we have no backreference at this point, and can add `inl n` to the list. The third constructor `cswbr_backref` uses our above helper relation to express a length/distance-pair. This relation may be long, but it should be straightforward.

### 7.5.3 Statically Compressed Blocks

Statically compressed blocks have fixed literal/length and distance codes. We define these according to [41],

```
(** See RFC 1951, section 3.2.6. *)
Definition vector_for_fixed_lit_code : vec nat 288 :=
  of_list ((repeat 144 8) ++ (repeat (255 - 143) 9) ++
     (repeat (279 - 255) 7) ++ (repeat (287 - 279) 8)).

(** See RFC 1951, section 3.2.6. *)
Definition vector_for_fixed_dist_code : vec nat 32 :=
   of_list (repeat 32 5).
```

and using our constructive existence proof from Section 5 we get codings from them, `fixed_lit_code` and `fixed_dist_code`. See `Encoding-Relation.v` for details. With these, defining the parsing relation for statically compressed blocks is straightforward:

```
Inductive StaticallyCompressedBlock
             (output : SequenceWithBackRefs Byte) : LB -> Prop :=
| makeSCB :
    forall input,
      CompressedSWBR fixed_lit_code fixed_dist_code output input ->
      StaticallyCompressedBlock output input.
```

### 7.5.4 Dynamically Compressed Blocks

One of the most complicated parts of the standard are the headers of dynamically compressed blocks. The codings of the compressed data are themselves compressed by a third coding, and can additionally contain instructions for a crude form of run length encoding. The third coding is a (possibly truncated) list of bit-triples in a strange order specified in [41]. An additional

header of three numbers is used to indicate how many elements every coding has. This is the problem when trying to give a readable specification: Everything is interconnected.

**The Code-length-code Header** We start with the "raw" form of the first coding, which is just a relation that gets a numeric parameter `hclen` and reads that number of bit triples into a list of natural numbers.

```
Inductive CLCHeaderRaw
  : forall (hclen : nat) (input : LB) (output : list nat), Prop :=
| zeroCLCHeaderRaw : CLCHeaderRaw 0 nil nil
| succCLCHeaderRaw : forall n i o j m, CLCHeaderRaw n i o ->
   ll m = 3 -> LSBnat m j -> CLCHeaderRaw (S n) (m ++ i) (j :: o).
```

This is the header that is actually read from the file. It can be at most of length 19, but if `hclen` $< 19$, it is truncated, and the remaining codepoints are regarded as 0. We therefore add a relation that pads the list we get from the former relation:

```
Inductive CLCHeaderPadded
  (hclen : nat) (input : LB) (output : list nat) : Prop :=
| makeCLCHeaderPadded : forall m output1,
                         CLCHeaderRaw hclen input output1 ->
                         output = output1 ++ repeat m 0 ->
                         ll output = 19 ->
                         CLCHeaderPadded hclen input output.
```

For the next relation, we need the permutation that is specified in [41]:

```
(** See RFC 1951, section 3.2.7. *)
Definition HCLensNat :=
  [16; 17; 18; 0;  8; 7;  9; 6; 10; 5;
   11;  4; 12; 3; 13; 2; 14; 1; 15].
```

With it, we can define a relation that permutes the sequence into the right order:

```
Inductive CLCHeaderPermuted
    (hclen : nat) (input : LB) (output : list nat) : Prop :=
| makeCLCHeaderPermuted :
   forall output1,
        CLCHeaderPadded hclen input output1 ->
        (forall m, nth_error output (nth m HCLensNat 19) =
                   nth_error output1 m) ->
         CLCHeaderPermuted hclen input output.
```

Finally, we can give a relation between the input and a Deflate coding, the code-length-coding, in which the two other codings are then encoded:

```
Inductive CLCHeader
  (hclen : nat) (output : deflateCoding 19) (input : LB) : Prop :=
| makeCLCHeader : forall cooked,
                    CLCHeaderPermuted hclen input cooked ->
```

```
                    CodingOfSequence cooked output ->
                    CLCHeader hclen output input.
```

where the relation `CodingOfSequence` relates a list of code lengths to a coding with these lengths:

```
Inductive CodingOfSequence {n : nat} (l : list nat)
          (dc : deflateCoding n) :=
| makeCodingOfSequence : forall (eq : ll l = n),
    Vmap lb (C _ dc) = vec_id eq (of_list l) ->
    CodingOfSequence l dc.
```

Of course, this relation is only satisfiable under the conditions elaborated in Section 5, and it is enforced by the dependent record `deflateCoding`. Our `CLCHeader` relation has a parameter `hclen` which comes from another header we describe later.

Having read that code, we use it to read the combined code lengths of the literal/length and distance coding, which are encoded by the code length coding. The problem is that the codepoints 16, 17 and 18 have a special meaning. Firstly, they come with suffix bits, for which we use the relation from Section 7.5.1 to realize this. Secondly, they stand for repetitions of either zeroes, for 17 and 18, or the last given byte, for 16. The codepoints 17 and 18 can therefore be at the beginning of the sequence, but 16 cannot, as it depends on a previously given code length. Instead of controlling this directly, we generate a sequence with backreferences, and then use our already given mechanism of backreference resolution from Section 7.4, where we insert trailing zeroes for 17 and 18, and therefore decrease the length by 1. Furthermore, a parameter to the relation counts the number of code length codes.

```
Inductive CommonCodeLengthsSWBR (clc : deflateCoding 19)
  : nat -> SequenceWithBackRefs nat -> LB -> Prop :=
| cswbr0 : CommonCodeLengthsSWBR clc 0 [] []
| cswbrc :
    forall m n brs lb1 input,
      CommonCodeLengthsSWBR clc n brs lb1 ->
      m < 16 ->
      dc_enc clc m input ->
      CommonCodeLengthsSWBR clc (n + 1) (inl m :: brs)
                            (input ++ lb1)
| cswbr16 :
    forall m n brs lb1 input,
      CommonCodeLengthsSWBR clc n brs lb1 ->
      CompressedWithExtraBits clc 16 [2] [3] [6] m input ->
      CommonCodeLengthsSWBR clc (n + m) (inr (m, 1) :: brs)
                            (input ++ lb1)
| cswbr17 :
    forall m n brs lb1 input,
      CommonCodeLengthsSWBR clc n brs lb1 ->
      CompressedWithExtraBits clc 17 [3] [3 — 1] [10 — 1]
                              m input ->
```

```
      CommonCodeLengthsSWBR clc (n + m + 1)
                            (inl 0 :: inr (m, 1) :: brs)
                            (input ++ lb1)
| cswbr18 :
    forall m n brs lb1 input,
      CommonCodeLengthsSWBR clc n brs lb1 ->
      CompressedWithExtraBits clc 18 [7] [11 - 1] [138 - 1]
                              m input ->
      CommonCodeLengthsSWBR clc (n + m + 1)
                            (inl 0 :: inr (m, 1) :: brs)
                            (input ++ lb1).
```

This might be seen controversial, because it is less direct than other relations. On the other hand, we decided to do it this way, so we can rely on the correctness of the already given relations, rather than rewriting them.

```
Inductive CommonCodeLengthsN (clc : deflateCoding 19) (n : nat)
          (B : list nat) (A : LB) : Prop :=
| ccl : forall C, CommonCodeLengthsSWBR clc n C A ->
                  ResolveBackReferences C B ->
                  CommonCodeLengthsN clc n B A.
```

Now, the given code lengths must be split into the code lengths of the literal/length coding, and the code lengths of the distance coding. Additional headers which we will introduce later will define the numbers of code lengths belonging to each of them. The rest must be padded by zeroes. Hence, the following relation:

```
Inductive SplitCodeLengths (clc : deflateCoding 19)
           (hlit hdist : nat) (litlen : vec nat 288)
           (dist : vec nat 32) (input : LB)
  : Prop :=
| makeSplitCodeLengths :
    forall litlenL distL lm ld,
      ll litlenL = hlit ->
      ll distL = hdist ->
      to_list litlen = litlenL ++ repeat lm 0 ->
      to_list dist = distL ++ repeat ld 0 ->
      CommonCodeLengthsN
          clc (hlit + hdist) (litlenL ++ distL) input ->
      SplitCodeLengths clc hlit hdist litlen dist input.
```

This part is an example for a source of confusion: Though the standard states it, it is easy to overread that the repetition codes may define repetitions that go over the boundary between the literal/length coding and the code length coding.

Finally, we relate the inputs to their respective codings:

```
Inductive LitLenDist (clc : deflateCoding 19) (hlit hdist : nat)
          (litlen : deflateCoding 288) (dist : deflateCoding 32)
          (input : LB) : Prop :=
| makeLitLenDist :
    SplitCodeLengths clc hlit hdist
```

```
                    (Vmap lb (C 288 litlen))
                    (Vmap lb (C 32 dist)) input —>
   LitLenDist clc hlit hdist litlen dist input.
```

Now, we add the three headers determining the respective lengths of the particular codings, using our monadic combinator from Section 6.3:

```
Definition DynamicallyCompressedHeader
  : (deflateCoding 288 * deflateCoding 32) —> LB —> Prop :=
  (readBitsLSB 5)
    >>= fun hlit => (readBitsLSB 5)
    >>= fun hdist => (readBitsLSB 4)
    >>= fun hclen => (CLCHeader (hclen + 4))
    >>= fun clc lld =>
          LitLenDist
            clc (hlit + 257) (hdist + 1) (fst lld) (snd lld).
```

As a final step, we can use our `CompressedSWBR` relation, which we already used for the statically compressed blocks:

```
Definition DynamicallyCompressedBlock
  : SequenceWithBackRefs Byte —> LB —> Prop :=
  DynamicallyCompressedHeader
    >>= fun lld => CompressedSWBR (fst lld) (snd lld).
```

## 7.6 Refactoring

By adding new abstractions, we could later change some of the definitions we already made. The problem with changing the specification is, of course, that it might introduce new bugs. That is why we keep the old definitions (and suffix them `_old`), and prove their equivalence with the new definition. This way, we cannot introduce new bugs. Some examples can be found in the file `EncodingRelationProperties.v`.

This is not just theory: We accidentally defined the `OneBit` relation in the following way:

```
Inductive OneBit : bool —> LB —> Prop :=
| oneBit : forall l b, OneBit b (b :: l).
```

This definition is wrong. We noticed this when we tried to prove the newer definition of `nBytesDirect` to be equivalent with the older definition, and it was not possible.

# Chapter 8

# Efficiency

Our primary goal was to make the specification as simple as possible. We use very simple data structures in our definitions. More sophisticated structures would have to be proven correct first.

## 8.1 Natural Numbers

We mostly used the `nat` type for numbers, with recursively defined arithmetic functions. They are natively supported by Coq and are easy to work with. Unfortunately, Coq does not provide builtins to map the `nat` type to bigints internally, as for example Agda does:

```
data Nat : Set where
  zero : Nat
  suc  : (n : Nat) -> Nat

_+_ : Nat -> Nat -> Nat
zero  + m = m
suc n + m = suc (n + m)

_*_ : Nat -> Nat -> Nat
zero  * m = zero
suc n * m = m + (n * m)

{-# BUILTIN NATURAL Nat #-}
{-# BUILTIN NATPLUS _+_ #-}
{-# BUILTIN NATTIMES _*_ #-}
```

Of course, such optimizations enlarge the trusted codebase. Other types like `N` which define binary numbers are more efficient, but still not as efficient as Haskell's bigints. And while tactics like `omega` can handle them, they can be tedious to work with. A generalization that might make things similar to builtins possible would be to allow substitution of "compatible" subexpressions in proof trees and recheck them, for some suitable concept of "compatibility". One possibility would be to use the typeclass mechanism of

Coq: Instead of using an algebraic datatype `nat`, we could define our own typeclass that defines methods `zero`, `succ` and `recur` and some axioms, in which we then could embed any definition of natural numbers. But then again, the standard library contains theorems and functions for raw lists and natural numbers only, and one would have to rewrite large parts of it. Another approach might be using Voevodsky's univalence axiom [100], as considered in homotopy type theory [97], which makes it possible to replace a type by an isomorphic one. To this point, however, there is no elegant way of doing this. Two problems arise from this.

The first problem is that when Coq encounters a large natural number, it crashes. The following line will crash Coq:

```
Definition A := 65535%nat.
```

However, the following line will not:

```
Definition A := (2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2)%nat.
```

Our solution uses a technique described in [25]; we define a function that parses a decimal string into a natural number if possible, and raise a type error otherwise:

```
Definition forceOption A Err (o : option A) (err : Err) :
  match o with
    | Some _ => A
    | None => Err
  end :=
  match o with
    | Some a => a
    | None => err
  end.

Inductive parseError := ParseError.

Fixpoint strToNat (strn : string) : option nat :=
  (fix rc str n :=
   match str with
     | EmptyString => Some n
     | String "0" str => rc str ((10 * n) + 0)
     | String "1" str => rc str ((10 * n) + 1)
     | String "2" str => rc str ((10 * n) + 2)
     | String "3" str => rc str ((10 * n) + 3)
     | String "4" str => rc str ((10 * n) + 4)
     | String "5" str => rc str ((10 * n) + 5)
     | String "6" str => rc str ((10 * n) + 6)
     | String "7" str => rc str ((10 * n) + 7)
     | String "8" str => rc str ((10 * n) + 8)
     | String "9" str => rc str ((10 * n) + 9)
     | _ => None
   end) strn 0.

Definition d (s : string) :=
```

```
  forceOption nat parseError (strToNat s) ParseError.

Example e1 : (d"22" : nat) = 22.
Proof. reflexivity. Qed.

Example e3 : (d"1x" : parseError) = ParseError.
Proof. reflexivity. Qed.
```

The second problem is that in extracted code, the inefficient representation is reflected. We can, however, give our own interpretations for extracted types via the `Extract Inductive` command. For example

```
Extract Inductive nat =>
 "Prelude.Int" [ "0" "(1 Prelude.+)" ]
  "(let r z s n = case n of { 0 -> z 0 ; " ++
                            "q -> s (q Prelude.- 1)} in r)".
```

However, the arithmetic functions will still be extracted to their recursive definitions, and there does not appear to be a designated way of overloading them. It is possible to abuse another mechanism which realizes axioms to overload our definition, the `Extract Constant` command:

```
Extract Constant plus => "(Prelude.+)".
Extract Constant mult => "(Prelude.*)".
```

We use several such commands to optimize our extracted code. In most cases, it is just a simple optimization; however, it enlarges the trusted codebase.

Newer versions of Coq have a native `int31` type. However, it is only "native" in Coq: It is defined as a binary number with 31 bits and some rules, and we would still have to manually add `Extract Constant` instructions. We think that extracting from `nat` and trusting the efficient big-integer implementations from the target language is a better design choice.

## 8.2 Singly-linked Lists

Lists are well-understood and simple, therefore we use them in our specification. Our first extracted algorithm used concatenation a lot, which is particularly slow with singly-linked lists. Since we thought that concatenation was a major bottleneck, we tried to optimize this using fast catenable deques [65].

In a purely functional environment like Coq, it is very convenient to use lazy lists for streamed file-i/o. With Haskell, it is no problem to load a file into a bit list, because it will lazily build this list, consume this list, consume the output list and write the output list to the output file. When we overloaded lists with catenable deques using `Extract Constant`, it broke laziness, resulting in the whole file being loaded bitwise into a deque,

consuming a lot of memory, while concatenation turned out not to be the major bottleneck.

## 8.3 Backreferences

The resolution of backreferences turned out to be the major bottleneck in our implementation. Our benchmarks in Section 9.3 show that even though we did not really optimize our implementation much, most of the decompression algorithm performs well, and gets the job done in seconds.

In an imperative language, one possible intuitive algorithm uses a 32 KiB array ring buffer, in which it saves the last 32 KiB that have been decompressed:

```
static class BackRef                                                {
    public int length, distance                                     ;
    public BackRef()                                               {}}

public static ArrayDeque<Character>
resolve (Deque<Object> input, int buflen)                           {
    ArrayDeque<Character> ret = new ArrayDeque<Character>()         ;
    Character[] buffer = new Character[buflen]                      ;
    int cptr = 0                                                    ;
    Object c                                                        ;
    while ((c = input.pollFirst()) != null)                         {
        if (c instanceof BackRef)                                   {
            int length = ((BackRef)c).length                        ;
            int distance = ((BackRef)c).distance                    ;
            while (length-- > 0)                                    {
                Character q =
                    buffer[ (buflen + cptr - distance) % buflen ]   ;
                buffer[cptr] = q                                    ;
                cptr = (cptr + 1) % buflen                          ;
                ret.addLast(q)                                     ;}}
        else if (c instanceof Character)                            {
            buffer[cptr] = (Character) c                            ;
            cptr = (cptr + 1) % buflen                              ;
            ret.addLast((Character)c)                              ;}}
    return ret                                                      ;}
```

This algorithm is fast. However, its invariants are rather complicated. Furthermore, verifying imperative algorithms with Coq is still an active research topic. We tried to formalize simple state monads with arrays, but it turned out that this quickly results in a reproduction of separation logic, making it not easier than using some implementation of it directly [26].

In the presence of uniqueness types, one could reproduce the algorithm recursively, using a unique array. Neither Haskell nor Coq has uniqueness types. However, something similar can be done with DiffArrays [10]. We implemented a function that uses DiffArrays and tested it, see Section 8.4. Of course, this uses a trick, and increases the trusted codebase. Thoughts on

how to make this kind of trick typesafe can be found in [98]. Still, having a really purely functional backreference resolver, without using tricks, seemed like a desirable goal.

The naïve way of resolving backreferences in a purely functional way is to save a reverse list of the decompressed data and access it:

```
resolve :: [Either a (Int, Int)] -> [a] -> [a]
resolve [] _ = []
resolve ((Left b) : r) x = b : resolve r (b : x)
resolve (Right (0, _) : r) x = resolve r x
resolve (Right (l, d) : r) x = (x !! (d - 1)) :
    resolve (Right (l-1, d) : r) ((x !! (d - 1)) : x)
```

This algorithm has two disadvantages. On the one hand, accessing the $n$-th element of a list, which it frequently does, takes $O(n)$ time. We can solve this problem using **ExpList**s instead of lists:

```
Inductive ExpList (A : Set) : Set :=
| Enil : ExpList A
| Econs1 : A -> ExpList (A * A) -> ExpList A
| Econs2 : A -> A -> ExpList (A * A) -> ExpList A.
```

The access time of the $n$-th element is $O(\log n)$; specifically the elements that have a small distance can be accessed faster, which takes into account that – in our experience – most backreferences tend to be "near", that is, have small distances, and such elements can be accessed faster.

On the other hand, this algorithm will save a reversed version of all of the decompressed data, so it will waste memory, because it only needs to save 32 KiB. We use another technique which we call **Queue of Doom**: We save two `ExpLists` and memorize how many elements are in them. The front `ExpList` is filled until it contains 32 KiB. If a backreference is resolved, and its distance is larger than the amount of bytes saved in the front `ExpList`, it is looked up in the back `ExpList`. Now, if the front `ExpList` is 32 KiB large, the front `ExpList` becomes the new back `ExpList`, a new empty front `ExpList` is allocated, and the former back `ExpList` will be doomed to be eaten by the garbage collector. The following is an illustration of filling such a queue of doom, the `ExpLists` are denoted as lists, and their size is

– for illustration – only 3:

| | | | |
|---|---|---|---|
| start | [] | [] | |
| push 1 | [1] | [] | |
| push 2 | [2; 1] | [] | |
| push 3 | [3; 2; 1] | [] | |
| push 4 | [4] | [3; 2; 1] | [] → ☠ |
| push 5 | [5; 4] | [3; 2; 1] | |
| push 6 | [6; 5; 4] | [3; 2; 1] | |
| push 7 | [7] | [6; 5; 4] | [3; 2; 1] → ☠ |

The advantage of this algorithm is that we have a fully verified implementation in `ExpList.v`. The disadvantage is that while it does not perform badly, it still does not have satisfactory performance, taking several minutes.

## 8.4 Using DiffArrays

Our main bottleneck is the resolution of backreferences. While in Section 8.5, we show how to do it efficiently in a purely functional manner, this specific problem appears to be inherently imperative: it is extremely easy and fast using a mutable array as a ring buffer. Mutable arrays have the advantage of $O(1)$ access and modification.

There are several possibilities to utilize arrays in a purely functional environment, the most popular currently being state monads. A way of easily getting stateful operations is to use adjustable references [99]. In some sense, this appears to be the most natural thing to do, since in the end, every runtime just abstracts away the stateful operations on the actual hardware. We use our own definition of adjustable arrays in Coq.

We adapted the code from our purely functional implementation with ExpLists from Section 8.3, rewrote some functions to use the structure in a more linear fashion, and defined the following axioms the structure needs to satisfy (we call the structure `DiffStack` for reasons that will become apparent later in this chapter):

```
Axiom DiffStack : Set -> Set.
```

The structure has a nil value:

```
Axiom DSNil : forall (A : Set), DiffStack A.
Arguments DSNil [_].
```

The structure can be converted into a list:

```
Axiom DStoL : forall (A : Set) (ds : DiffStack A),
                list A * DiffStack A.
Arguments DStoL [_].
```

The structure has a push-operation:

```
Axiom DSPush : forall (A : Set) (a : A) (ds : DiffStack A),
                  DiffStack A.
Arguments DSPush [_].
```

The structure has an $n$-th operation:

```
Axiom DSNth : forall (A : Set) (n : nat)
              (ds : DiffStack A) (default : A), A * DiffStack A.
Arguments DSNth [_].
```

The nil-, push- and $n$-th-operations are compatible with the conversion to a list:

```
Axiom NilNil : forall (A : Set), DStoL (@DSNil A) = ([], DSNil).

Axiom PushLst : forall (A : Set) (a : A) b,
    fst (DStoL (DSPush a b)) = a :: fst (DStoL b).

Axiom DSNth_nth : forall {A : Set} (x : DiffStack A)
                         (a : A) (n : nat),
    fst (DSNth n x a) = nth n (fst (DStoL x)) a.
```

These are the most important operations. They can be realized using a DiffArray and a few simple operations on top. Using these axioms looks like using a linear type, but the operations are just "faked" linear operations, so we add add `FakeLinear` axioms, which tell that the returned structure is unchanged:

```
Axiom DSNthFakeLinear : forall {A : Set} n ds d,
    snd (@DSNth A n ds d) = ds.
```

For efficiency, we add an operation to return a reversed list, and a reset-operation. This is optional, but it should save memory. To extract an algorithm, we need to add extraction directives:

```
Extract Constant DiffStack "q" => "DiffStackT.DiffStack q".
Extract Constant DSPush => "DiffStackT.push".
Extract Constant DSNth => "DiffStackT.nth".
Extract Constant DSNil => "DiffStackT.newDiffStack".
Extract Constant DStoL => "DiffStackT.toList".
Extract Constant DStoR => "DiffStackT.toReverseList".
Extract Constant ResetDS => "DiffStackT.reset".
```

The definition is the structure

```
data DiffStack a = DiffStack
    { size :: Int
```

```
    , sp :: Int
    , array :: D.DiffArray Int (Maybe a)
    }
```

As an example, we give the definition of the conversion to list, which is the most complicated function we need in our trusted codebase:

```
readArray arr index = (arr D.! index, arr)

toList' :: D.DiffArray Int (Maybe a) -> Int -> Int ->
           [a] -> ([a], D.DiffArray Int (Maybe a))
toList' ds n sp l =
    if n < sp
    then case DiffStackT.readArray ds n of
           (Nothing, ds_) -> toList' ds_ (n+1) sp l
           (Just x, ds_) -> toList' ds_ (n+1) sp (x : l)
    else (l, ds)

toList :: DiffStack a -> ([a], DiffStack a)
toList ds = let (l, narr) = toList' (array ds) 0 (sp ds) []
                    in (l, DiffStack { size = size ds
                                     , sp = sp ds
                                     , array = narr
                                     })
```

## 8.5 A Purely Functional, Efficient Backreference-resolver

All approaches so far save the actually produced output in some structure which they read afterwards. However, we can use the fact that our window of backreferences is limited by 32 KiB. The algorithm we present now uses a "look ahead" approach and reads the sequence in advance, memorizing the possible backreferences. It uses priority queues to memorize them in the proper order. We will show this algorithm in multiple steps, where we refine the intermediate algorithms - which might not be efficient at all - until we get a purely functional and efficient algorithm.

Due to the lack of time, we did not produce a verified implementation of this algorithm. However, we tested an unverified implementation in our benchmarks, see Section 9.3. The part of the implementation we have so far can be found in `DecompressWithPheap.v`. However, we will give an informal proof with a Coq implementation in mind. It should be possible to translate it to Coq.

The input of the algorithm is a sequence with backreferences, that is, of type $[A + \mathbb{N} \times \mathbb{N}]$, where the pair of natural numbers encodes a length and a distance. A trivial transformation we can do is to replace a subsequence $[(Sl, d)]$ by $[(1, d); (l, d)]$. Repeating this, and removing possible backreferences of length 0, we can reduce such a sequence to a sequence in which the

length of every backreference is 1. We can then just save it in a sequence of type $[A + \mathbb{N}]$, only saving the actual distance of this backreference.

```
Fixpoint BackRefsLengthOne {A : Set}
         (swbr : SequenceWithBackRefs A) :=
  match swbr with
    | [] => []
    | (inl x :: r) => inl x :: BackRefsLengthOne r
    | (inr (l, d) :: r) => repeat l (inr d) ++
                                   (BackRefsLengthOne r)
  end.
```

This can be done lazily, thus not requiring us to use additional memory. Notice that after transforming our sequence this way entirely, the input sequence and the output sequence have the same length.

In the following, we will regard this transformed sequence as our input, and a sequence with backreferences will be a sequence of type $[A+\mathbb{N}]$ rather than $[A + \mathbb{N} \times \mathbb{N}]$ as before. For the sake of simplicity, we will call the elements of $A$ "bytes". Our motivating example would become

$$\text{an}\overleftarrow{2\,2\,2}\text{s\_b}\overleftarrow{8\,8\,8\,8\,8\,7\,7\,7}\text{t}\overleftarrow{2\,2\,2}$$

### 8.5.1 Pairing Heaps

Pairing heaps [16] are purely functional priority queues which are – except for `decrease-key` – easy to implement, and have good amortized runtime behavior. We do not need the `decrease-key` operation, so we do not implement it.

Our implementation can be found in `Pheap.v`. Since we want to be able to temporarily violate invariants, we decided not to define pairing heaps as a dependent data structure, but define external properties about it. The definition of the structure is, besides the declaration of implicit arguments, the same as in Wikipedia [16].

```
Inductive pheap A : Type :=
| Empty : pheap A
| Heap : A -> list (pheap A) -> pheap A.

Arguments Empty [_].
Arguments Heap [_] _ _.
```

We then define some set-theoretic predicates `pheap_in`, `pheap_subseteq`, `pheap_subsetneq`. Since both `Heap 1 [Heap 2 [], Heap 3 []]` and `Heap 1 [Heap 2 [Heap 3 []]]` contain the same elements $\{1, 2, 3\}$, but are clearly not equal, we define an *extensional equality*

```
Inductive pheap_in {A} : A -> pheap A -> Prop :=
| H_in : forall a l, pheap_in a (Heap a l)
| L_in : forall a b h l, In h l -> pheap_in a h
                                -> pheap_in a (Heap b l).
```

```
Definition pheap_subseteq {A} a b :=
 forall (x : A), pheap_in x a -> pheap_in x b.

Definition pheap_subsetneq {A} a b :=
 @pheap_subseteq A a b /\ exists x, pheap_in x b /\ ~ pheap_in x a.

Definition pheap_ext_eq {A} a b :=
    @pheap_subseteq A a b /\ @pheap_subseteq A b a.
```

which states that both heaps contain the same elements. The main invariant for pairing heaps is defined as

```
Inductive pheap_correct {A} (cmp : A -> A -> bool) : pheap A ->
   Prop :=
| E_correct : pheap_correct cmp Empty
| H_correct : forall b l, Forall (pheap_correct cmp) l ->
                          (forall c, pheap_in c (Heap b l)
                                     -> cmp b c = true) ->
                          pheap_correct cmp (Heap b l).
```

which states that all subheaps are correct, and that the first element is smaller, with respect to `cmp`, than the later elements. Of course, at this point, we do not have any constraints on `cmp`. We define this externally, and only use it when we need it:

```
Definition cmp_ordering {A} (cmp : A -> A -> bool) :=
  (forall a, cmp a a = true) /\
  (forall a b c, cmp a b = true -> cmp b c = true
                                -> cmp a c = true) /\
  (forall a b, cmp a b = true -> cmp b a = true -> a = b) /\
  (forall a b, cmp a b = true \/ cmp b a = true).
```

We implement the `find_min` function and prove its specification:

```
Lemma find_min_spec : forall {A} (b : A) cmp h,
                          cmp_ordering cmp ->
                          pheap_correct cmp h ->
                          pheap_in b h ->
                          exists a,
                            Some a = find_min h /\
                            cmp a b = true.
```

Similar for `merge`

```
Lemma merge_spec : forall {A} cmp (b : A) g h,
                     (pheap_in b g \/ pheap_in b h) <->
                     pheap_in b (merge cmp g h).
```

and for `insert`

```
Lemma insert_spec :
  forall {A} (cmp : A -> A -> bool) (a b : A) (h : pheap A),
    cmp_ordering cmp ->
    (pheap_in a (insert cmp b h) <-> (a = b \/ pheap_in a h)).
```

For `delete_min`, the specifications are a little more complicated, as we need an additional helper function `merge_pairs`. We will not discuss it here. We then define a function `pheap_num` which counts the number of elements of a heap, and several properties about it, like subadditivity for merging. We need this to apply well-founded recursion over the number of elements in a heap, which we use in our algorithms and lemmata.

### 8.5.2 General Idea

We will first introduce a simpler algorithm to motivate the ideas. The most important idea of the algorithm is to work with **absolute** positions of characters in the list, rather than relative positions, that is, regarding the input- and output-list as function $\mathbb{N} \to \mathcal{A}$. This way, we can talk about a *specific* backreference which has a unique absolute **source** and **destination**. Obviously, if a backreference $\overleftarrow{n}$ is at position $t$, then $t$ is its destination, and $n - d$ is its source position.

As a first step of of our simpler algorithm, we collect all the backreferences in our input stream into a list of source-destination-pairs, and sort this list lexicographically:

```
collect :: SWBR1 a -> [(Int, Int)]
collect s = sort [ (n-d, n) | (Backref_ d, n) <- zip s [0..] ]
```

Notice that now, the backreferences are sorted according to their **source** position. We can resolve the backreferences from such a list with the following algorithm:

Let $m$ be some generic map structure, initially empty. The current absolute position in the input list is saved in a variable $n$, initially 0.

1. If the sorted backreference list is not empty, remove its first element and store it as $(s, d)$. Otherwise, proceed at step 4.
2. If $s \neq n$, proceed at step 4.
3. If $s = n$, there is a backreference to the current position $n$. Peek an element from the input.
   - 3a. If it is Char_ $c$, then set $m[d] = c$, and recur at step 1.
   - 3b. If it is Backref_ _, then set $m[d] = m[n]$, and recur at step 1.
4. Read an element from the input.
   - 4a. If we are at the end of the input, end.
   - 4b. If we read a character Char_ $c$, write $c$ to the output.
   - 4c. If we read a backreference Backref_ _, write $m[n]$ to the output.
5. Set $n = n + 1$ and recur at step 1.

In Haskell, we could implement it in the following way:

```
resolve_ :: [BR_ a] -> [(Int, Int)] -> Int -> Map Int a -> [a]
resolve_ l r n m =
    let res l r n m =
            case l of
              [] -> []
              (Char_ c : l') -> c : resolve_ l' r (n + 1) m
              (Backref_ _ : l') -> (m ! n) : resolve_ l' r (n+1) m
    in case r of
         [] -> res l r n m
         ((s, d) : r') ->
          if (s == n)
           then
            case l of
             (Char_ c : l') -> resolve_ l r' n (insert d c m)
             (Backref_ _ : l')->resolve_ l r' n(insert d (m ! n) m)
           else res l r n m
```

As can be seen in the highlighted parts of the code, we only ever use the table to look up the current position. We never look at anything smaller than the current position afterwards. Therefore, the structure of choice is a priority queue of destination-character-pairs sorted according to their destination:

```
resolve_ :: Ord a => [BR_ a] -> [(Int, Int)] -> Int ->
                     MinQueue (Int, a) -> [a]
resolve_ l r n m =
    let res l r n m =
            case l of
              [] -> []
              (Char_ c : l') -> c : resolve_ l' r (n + 1) m
              (Backref_ _ : l') ->
                  let (_, nm) = findMin m
                  in nm : resolve_ l' r (n + 1) (deleteMin m)
    in case r of
         [] -> res l r n m
         ((s, d) : r') ->
          if (s == n)
           then
            case l of
             (Char_ c : l') -> resolve_ l r' n (insert (d, c) m)
             (Backref_ _ : l') ->
                 let (_, nm) = findMin m
                 in  resolve_ l r' n (insert (d, nm) m)
           else res l r n m
```

We sorted the list of pairs in advance. We could as well replace it by a priority queue which returns the pairs in the right order:

```
resolve_ l r n m =
    let res l r n m =
            case l of
```

```
            [] -> []
            (Char_ c : l') -> c : resolve_ l' r (n + 1) m
            (Backref_ _ : l') ->
                let (_, nm) = findMin m
                in nm : resolve_ l' r (n + 1) (deleteMin m)
in case minView r of
      Nothing -> res l r n m
      Just ((s, d), r') ->
      if (s == n)
       then
        case l of
         (Char_ c : l') -> resolve_ l r' n (insert (d, c) m)
         (Backref_ _ : l') ->
             let (_, nm) = findMin m
             in  resolve_ l r' n (insert (d, nm) m)
       else res l r n m
```

In the highlighted pair, notice that $s - d \leq 32768$. Since only pairs with the current source position are needed, it is sufficient to read 32768 input characters in advance and make sure that all references are in.

### 8.5.3 A Formal Proof

This is a proof which is given with the actual verified algorithm in mind, that is, a proof that is given in a way that should be reproducible in Coq. Due to the lack of time, we did not manage to implement it in Coq entirely.

Let us call the front position we have $m$, and the back position $n$. We call the two queues $b$ and $c$. We call the sets of elements of $b$ and $c$ respectively $\overline{b}$ and $\overline{c}$ (so it will be easier to talk about it informally). Let us call $i$ the input sequence with backreferences of length 1, and $r$ the result with the resolved backreferences. As we proved strong uniqueness of our relation, $r$ is unique, so this notion is well-defined. We write $o_2 \mapsto o_1$ to express that $i \mathbin{!!} o_2$ is a backreference to $o_1$, where $o_1, o_2$ are absolute positions in $i$.

Firstly, we have the following two invariants for our priority queues:

$$\overline{b} = \{(o_2, r \mathbin{!!} o_1) \mid o_1 < n \leq o_2 \wedge o_2 \mapsto o_1\} \tag{8.1}$$

$$\overline{c} = \{(o_1, o_2) \mid n \leq o_1 < o_2 < m \wedge o_2 \mapsto o_1\} \tag{8.2}$$

We furthermore want that $n$ and $m$ have at least distance $D$ if possible:

$$m - n \geq D \vee n = 0 \vee m = \operatorname{len} i \tag{8.3}$$

The algorithm has essentially three phases:

- The start phase, where $n = 0$ but $m - n < D$ and $m \neq \operatorname{len} i$.
- The interim phase, where $m \neq \operatorname{len} i$, and $m - n \geq D$.

- The end phase, where $m = \operatorname{len} i$.

It is a further invariant that:

$$\text{we will always be in one of these phases} \tag{8.4}$$

Notice that if the data size is $\leq D$, then there will be no interim phase.

We have intermediate lists $l_m$ and $l_n$, which are truncations of $i$, respectively, with the invariants

$$l_m = \operatorname{drop}_m i \tag{8.5}$$
$$l_n = \operatorname{drop}_n i \tag{8.6}$$

where

$$\operatorname{drop}_k l = \begin{cases} l & \text{for } k = 0 \\ [\,] & \text{for } l = [\,] \\ \operatorname{drop}_{k'} l' & \text{for } k = k' + 1 \text{ and } l = \_ :: l' \end{cases}$$

Therefore, $i \mathbin{!!} n = l_n \mathbin{!!} 0$ and $i \mathbin{!!} m = l_m \mathbin{!!} 0$, which means that we can easily destructure $l_m$ and $l_n$ at every step.

The state is entirely described by the tuple $(n, m, l_n, l_m, b, c)$.

In the beginning, $n = 0$ and $m = 0$, $l_m = l_n = i$, and the priority queues are empty. The invariants are trivially satisfied at this point, and we are in the start phase.

As a first subprocedure `m_inc`, we show that given a tuple $(n, m, l_n, l_m, b, c)$, if $l_m = x :: l_{m+1}$, and all the invariants hold, we can increase $m$ consistently, and modify the other values appropriately, such that the resulting tuple $(n, m + 1, l_n, l_{m+1}, b, c')$ still satisfies all the invariants.

1. If $l_m = \operatorname{inl} \_ :: l_{m+1}$, we can just return the tuple $(n, m + 1, l_n, l_m, b, c)$. Invariant (8.2) is preserved: All former elements are still in $c$, and the only possible additional element is not a backreference. Invariant (8.3) is also preserved, since $m - n \geq D$ implies $m + 1 - n \geq D$, and $m = \operatorname{len} i$ cannot hold if $l_m = \operatorname{inl} \_ :: l_{m+1}$. The other invariants do not talk about $m$ and can therefore not be violated.

2. If $l_m = \operatorname{inr} d :: l_{m+1}$, we first check whether $d > m$.

   (a) If so, this is a format error: A backreference that goes too far.

   (b) Otherwise, we have to add $(m - d, m)$ to $c$. Invariant (8.2) is preserved: All former elements are still in $c$, and there can only be an additional one, which we just added. As in the first case, replacing $m$ by $m + 1$ will not violate any invariant.

We proved this part in `DecompressWithPheap.v`. Notice that if initially $m - n \geq D$, after the `m_inc` we have the sharp inequality $m - n > D$.

We need this for the second subprocedure `n_inc`: We show that if $m-n > D$ or $m = \operatorname{len} i$ and $n \neq \operatorname{len} i$, and all the invariants hold, we can increase $n$ consistently, and the resulting tuple will also satisfy all invariants. Trivially, $l_n \neq [\,]$, therefore, $l_n = x :: l_{n+1}$.

1. If $x = \operatorname{inl} c$, we know that $r \mathbin{!!} n = c$.

2. If $x = \operatorname{inr} d$, by (8.1), we know that $(n, r \mathbin{!!} n) \in \overline{b}$, and furthermore, this must be the lexicographically smallest element in $b$, because there can only be one element with $n$ in the first component, and there can be no pair with first element $< n$ due to (8.1). Therefore, we can use `find-min` and `delete-min` on $b$ to find this element. We now know $r \mathbin{!!} n$. Invariant (8.1) may be violated for $b'$ and $n+1$, but we restore it in the next step.

We now need to determine all elements of $c$ which have $n$ as their first component. In this intermediate step, invariant (8.2) will temporarily be violated. Instead, the following invariant holds:

$$o_2 \mapsto n \to (o_2, r \mathbin{!!} n) \in \overline{b} \vee (n, o_2) \in \overline{c} \tag{8.7}$$

In the beginning, the right side of the disjunction will always hold. We check with `find-min` for the minimal element of $c$ and whether it has $n$ as first component. As long as this is true, we use `delete-min`, and add $(o_2, r \mathbin{!!} n)$ to $b$. At every step, (8.7) will hold. At every step, $c$ gets smaller, therefore, at some point, this algorithm terminates. Since then, $(n, o_2) \in \overline{c}$ cannot hold anymore for any $o_2$, we know that for every $o_2$, $(o_2, r \mathbin{!!} n) \in \overline{b}$. Therefore, our new $b$ satisfies the invariant (8.1) for $n+1$. Our new $c$ also trivially satisfies the invariant (8.2) for $n+1$. Also trivially, after applying this, either $m = \operatorname{len} i$, or $m - n \geq D$, since before we had $m - n > D$ and $m$ increased by one, so invariant (8.3) holds.

Now that we have `m_inc` and `n_inc`, we can write the actual algorithm:

- If $m = \operatorname{len} i$
  - If $n < m$, we are in the end phase, and apply `n_inc` until $n = m$.
  - If $n = m$, we are done.
- If $m \neq \operatorname{len} i$
  - If $m - n < D$, then we are in the start phase. Hence, we know that $n = 0$ by (8.4). We apply `m_inc` repeatedly, until $m-n \geq D$ or $m = \operatorname{len} i$.

– If $m - n \geq D$ we are in the interim phase. We want to increase both $m$ and $n$. Since $m \geq n$, we know that both $l_n = x_n :: l_{n+1}$ and $l_m = x_m :: l_{m+1}$. We first apply `m_inc`, and then `n_inc`. As $(m+1) - (n+1) = m - n \geq D$, this is allowed. We do this, until $m = \operatorname{len} i$.

# Chapter 9

# Extraction and Testing

We depend on some specific Haskell libraries and the specific Coq version 8.6. To make our results reproducable in the future, we decided to provide a `Dockerfile` which only depends on Debian Stretch packages. We expect the Debian Stretch repositories to be archived in the future, as current old stable versions are, and therefore, our work should be runnable in the future, even when the software is deprecated.

## 9.1 Extraction

We chose to name our project "DampFnudeL". You can download the source from the GitHub-repository https://github.com/dasuxullebt/DampFnudeL. This work is about revision `6b04 96d0 ed5a f231 99bf 0a03 e04d bb9b b037 3873`, which you can download via

```
git clone https://github.com/dasuxullebt/DampFnudeL
cd DampFnudeL
git reset ––hard 6b0496d0ed5af23199bf0a03e04dbb9bb0373873
```

Notice that it depends on CpdtTactics [38].

In `Extraction.v`, we define some constants for extraction, like mapping Coq strings to Haskell strings. We furthermore overload arithmetic operations with Haskell's native operations, which are part of the trusted codebase. Instead of inlining, we wrote a Haskell module `Extraction` which can be found in `Extraction.hs`, which defines functions that we use.

We define a function `DeflateTest` that essentially uses our strong decidability proof to decide whether a given dataset can be decompressed. It is then called with the given files and command line arguments by the main method of the respective Haskell module.

To make it easier to reproduce our benchmarks, as well as install the several dependencies, we define a Docker container which bases on Debian Stretch.

### 9.1.1 Compatibility

Originally, we aimed to be compatible with a range of Coq versions. However, Coq changes its library names and syntax throughout the versions we like to support. For example, version 8.7 requires a package `FunInd` to be imported to use functional induction, but this package is not recognized by former versions.

One way of coping with this is given in [39]: Using the eRuby template engine, it is possible to match on the current Coq version, and insert code depending on it. We tried this, but as the tool support is not yet given, we decided to only support Coq version 8.6.1 for now.

### 9.1.2 Makefile

We require GNU Make for building. We define tests with names of the form `NoRBR-alice29.txt`, which contain the test name, and the name of the file.

The build process involves compiling the Coq files, for which we wrote a simple Ruby script which detects the dependencies of our Coq files, and generates a file `coqfiles.mak`, which we include in our Makefile. We had to manually delete the entries that actually create extracted Haskell-files, and put them in the main makefile. Then, the extracted Haskell-files are patched with AWK, so they include our `Extraction` package and several dependencies. This is necessary, since the program extraction facility of Coq does not allow adding dependencies directly.

## 9.2 Testing Unverified Algorithms

Since writing a verified algorithm is usually a lot harder than writing an algorithm without verification, it is desirable to know whether the approach of that algorithm is worthwile. Using our modular approach, it was possible to test different algorithms, before actually verifying them.

The bottleneck of our current implementation, namely the resolution of backreferences, was a natural candidate for applying this technique. In Section 9.3.4, we will discuss an example.

## 9.3 Benchmarks

While we verified the correctness of our program, we did not verify its space and time requirements. To get exact boundaries, we would require a clear order of evaluation, and the speed of all elementary evaluations, while the asymptotic behavior of the algorithms is clear in most cases. There are approaches to get such boundaries automatically, like Resource Aware ML [58]. We chose to test the requirements directly.

Since decompression usually means doing the same thing very often, there is lots of potential for optimization at almost every level. We followed the usual top-down approach, and profiled our implementation, and it turned out that the resolution of backreferences (see Section 7.4) was the major bottleneck, lifting the time of execution from seconds to minutes (See Section 9.3).

We did not optimize for space yet, and our implementation uses lots of memory, but in Section 6.4, we suggest how to lower the space requirements (we mainly use stack space because of exception propagation).

We wrote a tool that uses `time(1)` to measure the runtime and memory consumption, and save it to an SQLite database. We use the Canterbury Corpus [5]. As most of our examples are about decompression, we compress it with gzip beforehand, except for our compression algorithm.

Some common functions, like converting the internal representation of bytes to Haskell's bytes, can be found in `DecompressHelper.hs`.

The benchmarks have been made on an `Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz` with 2 cores á 2 threads.

### 9.3.1 No Backreferences

To test the bottleneck of our implementation, we removed several parts and tested how fast the rest is. Leaving out the resolution of backreferences shows that this is actually the part that takes most time, and while the rest is still several magnitudes worse than the ZLib, it only takes seconds, which is satisfactory at this point.

| Memory (KiB) | Time (s) | File |
|---|---|---|
| 62900 | 0.96 | alice29.txt |
| 53224 | 0.96 | asyoulik.txt |
| 17540 | 0.17 | cp.html |
| 11332 | 0.1 | fields.c |
| 9268 | 0.06 | grammar.lsp |
| 273524 | 5.93 | kennedy.xls |
| 168032 | 2.94 | lcet10.txt |
| 210048 | 4.09 | plrabn12.txt |
| 81048 | 1.66 | ptt5 |
| 21560 | 0.4 | sum |
| 10220 | 0.05 | xargs.1 |

### 9.3.2 With ExpLists

These are the benchmarks for the implementation we described in Section 8.3, which uses exponential lists. It is comparably slow. The memory footprint is, however, comparable to the other implementations. Still, this is a

purely functional and fully verified decompression mechanism that does not use any additional axioms whatsoever.

| Memory (KiB) | Time (s) | File |
|---|---|---|
| 93276 | 727.25 | alice29.txt |
| 76952 | 628.5 | asyoulik.txt |
| 18488 | 86.73 | cp.html |
| 11368 | 4.22 | fields.c |
| 9268 | 0.27 | grammar.lsp |
| 590924 | 3540.27 | kennedy.xls |
| 251992 | 1958.08 | lcet10.txt |
| 284760 | 2177.95 | plrabn12.txt |
| 296036 | 2241.7 | ptt5 |
| 24672 | 177.54 | sum |
| 10352 | 0.42 | xargs.1 |

### 9.3.3 With DiffArrays

Our implementation with DiffArrays, as described in Section 8.4, takes more memory than expected. We assume that some optimization process destroys linearity, and makes it necessary to copy the array at some point. However, the runtime is satisfactory.

| Memory (KiB) | Time (s) | File |
|---|---|---|
| 62936 | 1.69 | alice29.txt |
| 57812 | 1.52 | asyoulik.txt |
| 17868 | 0.31 | cp.html |
| 12756 | 0.15 | fields.c |
| 9232 | 0.08 | grammar.lsp |
| 422320 | 9.96 | kennedy.xls |
| 166312 | 4.65 | lcet10.txt |
| 210352 | 6.12 | plrabn12.txt |
| 186848 | 3.96 | ptt5 |
| 24912 | 0.63 | sum |
| 9644 | 0.09 | xargs.1 |

### 9.3.4 Unverified Functional Resolver

Here, we used the algorithm described in Section 8.5, but did not verify it. This implementation can compete with the former implementation with DiffArrays regarding runtime, but the memory footprint is much larger.

| Memory (KiB) | Time (s) | File |
|---|---|---|
| 149652 | 4.05 | alice29.txt |
| 122012 | 3.43 | asyoulik.txt |
| 28796 | 0.56 | cp.html |
| 14448 | 0.25 | fields.c |
| 9348 | 0.11 | grammar.lsp |
| 949392 | 17.42 | kennedy.xls |
| 423100 | 10.72 | lcet10.txt |
| 474220 | 12.78 | plrabn12.txt |
| 476308 | 7.5 | ptt5 |
| 42136 | 0.85 | sum |
| 10272 | 0.14 | xargs.1 |

### 9.3.5 Compression

For compression, we wrote a helper function `gzclad` which adds a GZip header and the required CRC32 checksum to the compressed deflate stream. Our compression algorithm is purely functional, and only finds repetitions for possible backreferences, but does not utilize dynamic Huffman codings. In the following table, the memory sizes are given in KiB:

| File | Original | Compressed | Time (s) | Memory |
|---|---|---|---|---|
| alice29.txt | 152 | 116 | 458.91 | 187640 |
| asyoulik.txt | 124 | 96 | 311.93 | 150748 |
| cp.html | 28 | 16 | 9.1 | 40124 |
| fields.c | 12 | 8 | 1.76 | 21592 |
| grammar.lsp | 4 | 4 | 0.31 | 10316 |
| kennedy.xls | 1008 | 432 | 4483.02 | 1004784 |
| lcet10.txt | 420 | 320 | 2478.75 | 479324 |
| plrabn12.txt | 472 | 384 | 3045.61 | 568528 |
| ptt5 | 504 | 264 | 3444.06 | 612544 |
| sum | 40 | 24 | 21.71 | 51384 |
| xargs.1 | 8 | 4 | 0.28 | 10404 |

The best compression ratio is gained for `kennedy.xls`, since this file contains lots of repetitions. However, this comes at the cost of memory and runtime. Though the runtime and memory costs of our compression program are not good, they are satisfactory, since we did not optimize the compression algorithm yet, and it only uses purely functional structures. In this thesis, we focused on decompression; writing a better compression algorithm is further work.

## 9.4 Building and Running

In this section we describe how to build the Docker container which contains our project, and run the benchmarks. Another thing we had to test was whether our specification is conformant with what other implementations understand as "Deflate" – the specification is axiomatic, and while it definitely describes a data compression format, there is no way of formally verifying that it really conforms to the informal specification. We therefore also describe how to test our algorithms with real-life data, using `bind` mounts.

We hope that in the future, it will still be possible to build the Docker container this way. However, there might be adaptations necessary, which we cannot foresee at the time of writing.

To build the Docker container, `cd` into the software directory, and run

```
$ docker build .
```

This should result in a prompt of a successfully built container, which has an ID that might be different from the ID we use here, so just substitute the ID from your output prompt

```
Successfully built 66ef0ea29d8d
```

We assume that we have a directory `/tmp/gztest` in which there is a file `decomp.gz` and a file `comp.txt` which we want to decompress and compress. We bindmount this to `/mnt`, and start the container with the argument `-it` to get a command prompt:

```
$ docker run –v /tmp/gztest:/mnt –it 66ef0ea29d8d
```

The contents of our project are in the directory `/var/deflate`, and the benchmark process can be run using `make Benchmarks`; you can pass `-j $(nproc)` to utilize multiple cores.

```
# cd /var/deflate
# make –j $(nproc) Benchmarks
```

The benchmarks will take a while. The command builds the executables from our code, and runs the benchmarks on the Canterbury Corpus, and generates an SQLite file `benchs.db` which can be accessed using SQLite3:

```
# sqlite3 benchs.db
sqlite> .mode column
sqlite> .header on
sqlite> select max_mem_ki_bs,runtime_secs,arguments from benchmark
   ...> where command = './CompressMain';
```

gives an overview over the benchmarks for the compression, for example. Notice that the argument strings start with "s", which is a type indicator for the Haskell database framework.

To decompress the file we bind-mounted using the DiffArray method, use

```
# ./DiffStackMain /mnt/decomp.gz /mnt/decomp
```

To compress, use

```
# ./CompressMain /mnt/comp.txt /mnt/comp.txt.gz
```

# Chapter 10

# Conclusion

Our contribution is a complete mathematical formalization of Deflate. We formalized the proofs in Coq, such that an implementation of a decompression algorithm in Haskell can be extracted. We tested this implementation against some datasets, and observed that it is compatible with other implementations of Deflate.

## 10.1 Further Work

### 10.1.1 Streamable Strong Decidability

As we showed in Section 6.4, it should be possible to rapidly decrease memory consumption by switching from strong decidability to streamable strong decidability. This seems to be a worthwile next step for making the project more useful.

### 10.1.2 Fast Compression

The compression algorithm is still slow. We hope to be able to make it faster, with the knowledge we gained by optimizing the other algorithms. Probably, using a DiffArray-based hashtable as a heuristic for backreferences instead of an AVL-based hashtable which we did in our implementation will bring a lot of speedup.

### 10.1.3 Trusted Codebase

The trusted codebase of our project is still large. In this section, we will discuss to what extent we could reduce it, and how this could be achieved. Of course, to this point, it is inevitable to trust the underlying hardware and operating system, so we will focus on things above that level.

**Coq** We have to trust Coq itself. Coq is an LCF-Style theorem prover, meaning that we do not have to trust its tactics, but only the Kernel which checks proofs. This kernel is a short Ocaml program, but it might still contain bugs. In one case, a bug was introduced by an optimization, and it was possible to derive `False` [44].

However, this bug required explicit crafting. Of the proofs we give, only trivial parts are computer generated, and the main ideas are thought through. Hence, it seems extremely unlikely, that such a bug would invalidate the derived guarantees.

Another danger comes from the theory itself, which might be inconsistent. However, we only use a small fragment of the actual type universe, and our relations and proofs should be portable into any other commonly used theory.

Hence, this part of the codebase is probably the most trustworthy we will work with. It is not possible to remove this part of the trusted codebase. There are efforts like the Coq in Coq project [29], that try to verify Coq inside itself. However, this is not entirely possible, which follows from Gödel's incompleteness theorem.

**Extraction** Besides Coq itself, we trust the program extraction mechanism which is itself not verified. It might be possible that extracted programs do not reflect the given proofs. To prove this, we would need a semantic of the language we extract to. There are efforts for a verified extraction mechanism [62], but these are outside the scope of this work. Program extraction is – for the largest part – straightforward, and we do not believe that there are major bugs that can invalidate our results.

**Ocaml** As Coq is written in Ocaml, we have to trust the Ocaml compiler and runtime. Ocaml is a widely used industrial quality programming language, and the Coq proofchecker's code does only use simple terminology every other sophisticated program uses, and hence, is trustworthy in the sense that all relevant bugs have probably been found. There are projects like CakeML [4] which try to cope with this problem, but at the time we wrote our project, these were not developed far enough to be used.

**GHC, big integers** Finally, we trust the Glasgow Haskell Compiler. This is probably the most critical part, as GHC does a lot of optimization and has a sophisticated runtime.

We rely on its big integer implementation. This dependency could be removed, by rewriting parts of the implementation using Coq's binary integers or native integers. However, we do not think that this is worthwile, since big integers and the parts we use – mostly addition and comparison – are not complicated or hard to implement.

We could at least replace these parts with Ocaml, so we would not have to trust two programming languages. However, we also rely on lazy evaluation.

**Lazy Evaluation** As we explicitly rely on lazy evaluation for I/O, we cannot just switch to Ocaml, which uses eager evaluation. An alternative that looks suitable are iteratees and enumeratees as defined in [68]; these are one possible alternative that would also work in Ocaml. However, inside Coq, there is no ready to use library for them. At this point, we do not have time to implement one ourselves.

**Memory** We trust the automatic memory management of GHC. For example, our queues of doom from Section 8.3 require them. In a purely functional setting, it seems impossible not to rely on it.

### 10.1.4 Imperative Implementation

In Section 1 we already pointed out an easy way to get a verified compression algorithm from an unverified compression algorithm. While we also pointed out why this is not the way to go, it is a benchmark that an implementation should always keep in mind. Though our implementation behaves well and we could get the runtime behavior down to a useful level, it is still slower than this implementation applied to the Zlib would be, and it is probably possible to optimize the code even further in a purely functional fashion, which would be interesting for its own sake.

However, for improvements into that direction, one should note that the Deflate standard was probably made with imperative low-level programming techniques rather than functional programming in mind. To give an imperative algorithm and verify it in Coq, there are several possibilities. One is the technique of using a Hoare state monad [92], and axiomatizing Haskell's ST monads or using the stateful operations from OCaml. This is probably the least intrusive technique, but the trusted codebase will remain large. The same goes for Ynot [22], which is a framework for simple imperative programs, which are then extracted to Ocaml.

The usual tool being used for the desired level of efficiency is the C programming language. And in fact, it has a complete toolchain which is already verified, the CompCert compiler, as well as the Verified Software Toolchain [26], everything already formalized in Coq. It should therefore be possible to use our specification to verify a C implementation of Deflate.

Another System is Frama C [12] with its Jessie plugin that allows the use of the Ansi/C specification language (ACSL). It can utilize many automatic provers and proof assistants, also Coq. However, it is not formalized in Coq itself.

While these tools are still quite complicated, they make the implementation of a fast verified ZLib replacement realistic, especially since it has to be

programmed only once, and could then be used everywhere. Such an implementation would be both of practical use, and it would show the strengths and weaknesses of the said tools, and could lead to an improvement of those.

As a more general approach, one could use the Verified Software Toolchain to write a formally verified program extraction mechanism to C.

### 10.1.5 Usage In Other Projects

After the aforementioned natural next steps, like the formalization to gain a complete implementation of a verified compression and expansion tool with competitive runtime and memory requirements, we could furthermore think of embedding it into other projects, as Deflate is used in many other standards; for example, the Quark browser [63] or MiTLS [32].

### 10.1.6 Other System Components

Of course, besides optimizing this specific project, there are other middleware system components that can be verified in a similar manner. Firstly, there are the many other compression formats like BZip2, Zip and XZ. Then, there are archive formats like tar and zip. Several formats that are using Deflate internally could be verified too, for example, the PNG format encodes graphics. Formats like PNG and FLAC produce lossless compression of graphics and sound, which means that one should be able to give a relation between a pixel matrix or sample sequence and the data stream. An interesting other problem would be using lossy compression schemes, where one would have to find apropriate relaxations of this strict scheme. Furthermore, one could think of a verified parser for more high-level formats like XML, which allows for validation, and on which many other formats base. Then, similarly to lossy compression, tag soup parsers, which are not exact, but relaxed, are an interesting problem for verification.

Then, there are several protocols that can be verified, like SMTP, HTTP, XMPP. Also, a verified content management system would be useful, which guarantees that only authenticated persons can access respective content. In that context, a verified implementation of the CommonMark standard [8] sounds interesting. A verified revision control system which guarantees not to lose old versions and gives some guarantees for merging also sounds like a reasonable goal.

## 10.2 Lessions Learned

While we tried to use low-level direct proving in the beginning, with as few tactics as possible, we quickly learned that the use of tactics improves the readability of proofs and often makes them self-documenting. Documentation on parts that are less clear helps, especially in cases where Coq version

changes break compatibility. While program extraction was a very good way of prototyping, this special kind of problem appears to be better solved imperatively, especially the resolution of backreferences. Our successful use of DiffArrays shows that linear types would be a nice feature that would enable us to do such a thing directly and without tricks.

# Bibliography


[1] Agda, http://wiki.portal.chalmers.se/agda/pmwiki.php, accessed: 2017-11-17

[2] Alte und neue Blindenschriften, http://www.fakoo.de/blindenschriften.html#zeitleiste, accessed: 2017-11-15

[3] Brailleschrift, https://de.wikipedia.org/wiki/Brailleschrift, accessed: 2017-11-15

[4] Cakeml – a verified implementation of ml, https://cakeml.org, accessed: 2017-11-17

[5] The canterbury corpus, http://corpus.canterbury.ac.nz/g, accessed: 2017-11-17

[6] Certicrypt: Computer-aided cryptographic proofs in coq, http://certicrypt.gforge.inria.fr/

[7] Chicken scheme, http://www.call-cc.org/, accessed: 2017-11-17

[8] Commonmark – a strongly defined, highly compatible specification of markdown, http://commonmark.org/, accessed: 2017-11-27

[9] Data structure is extracted twice, https://coq.inria.fr/bugs/show_bug.cgi?id=5754, accessed: 2017-11-17

[10] The diffarray package, https://hackage.haskell.org/package/diffarray, accessed: 2017-11-17

[11] Emscripten, https://en.wikipedia.org/wiki/Emscripten, accessed: 2017-11-17

[12] Frama C, http://frama-c.com/, accessed: 2017-11-17

[13] Idris | a language with dependent types, http://www.idris-lang.org/, accessed: 2017-11-17

[14] Jeffrey mark siskind's software, https://engineering.purdue.edu/~qobi/software.html, accessed: 2017-11-17

[15] Morse code, https://en.wikipedia.org/wiki/Morse_code, accessed: 2017-11-15

[16] Pairing heap, https://en.wikipedia.org/wiki/Pairing_heap, accessed: 2017-11-17

[17] The reference implementation of the linux fuse (filesystem in userspace) interface, https://github.com/libfuse/libfuse, accessed: 2017-11-17

[18] Three examples of problems with lazy i/o, http://newartisans.com/2013/05/three-examples-of-problems-with-lazy-io/, accessed: 2017-11-17

[19] Using the ring solver, http://wiki.portal.chalmers.se/agda/%5C?n=Libraries.UsingTheRingSolver, accessed: 2017-11-17

[20] Vellvm: Verifying the llvm, http://www.cis.upenn.edu/~stevez/vellvm/, accessed: 2017-11-17

[21] Welcome to native client, https://developer.chrome.com/native-client, accessed: 2017-11-17

[22] The Ynot project, http://ynot.cs.harvard.edu/, accessed: 2017-11-17

[23] Ackermann, W.: Zum hilbertschen Aufbau der reellen Zahlen. Mathematische Annalen 99(1), 118–133 (1928)

[24] Affeldt, R., Hagiwara, M., Sénizergues, J.: Formalization of shannon's theorems. Journal of Automated Reasoning 53(1), 63–103 (2014), http://dx.doi.org/10.1007/s10817-013-9298-1

[25] de Amorim, A.A.: Parse errors as type errors, http://poleiro.info/posts/2013-04-03-parse-errors-as-type-errors.html, accessed: 2017-11-17

[26] Appel, A.W.: Program Logics for Certified Compilers. Cambridge University Press (April 2014)

[27] Avigad, J., Feferman, S.: Gödel's functional ("dialectica") interpretation. Handbook of proof theory 137, 337–405 (1998)

[28] Bailey, D.: Raising lazarus - the 20 year old bug that went to mars, http://blog.securitymouse.com/2014/06/raising-lazarus-20-year-old-bug-that.html, accessed: 2017-11-17

[29] Barras, B.: A formalisation of the calculus of constructions, https://github.com/coq-contribs/coq-in-coq, accessed: 2017-11-17

[30] Berger, U., Jones, A., Seisenberger, M.: Program extraction applied to monadic parsing. Journal of Logic and Computation p. exv078 (2015)

[31] Berger, U., Schwichtenberg, H., Seisenberger, M.: The warshall algorithm and dickson's lemma: Two examples of realistic program extraction. Journal of Automated Reasoning 26(2), 205–221 (2001)

[32] Bhargavan, K., Fournet, C., Kohlweiss, M., Pironti, A., Strub, P.Y.: Implementing tls with verified cryptographic security (2013), http://www.mitls.org/downloads/miTLS-report.pdf, accessed: 2017-11-17

[33] Blanchette, J.C.: Proof pearl: Mechanizing the textbook proof of huffman's algorithm. J. Autom. Reason. 43(1), 1–18 (Jun 2009), http://dx.doi.org/10.1007/s10817-009-9116-y

[34] Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm (1994)

[35] Chen, H., Ziegler, D., Chajed, T., Chlipala, A., Kaashoek, M.F., Zeldovich, N.: Using crash hoare logic for certifying the fscq file system. In: Proceedings of the 25th Symposium on Operating Systems Principles. pp. 18–37. ACM (2015)

[36] Chlipala, A.: The bedrock tutorial, http://plv.csail.mit.edu/bedrock/tutorial.pdf, accessed: 2017-11-17

[37] Chlipala, A.: Certified programming with dependent types (2011)

[38] Chlipala, A.: Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant. MIT Press (2013)

[39] Claret, G.: Why and how to write code compatible with many coq versions, http://coq-blog.clarus.me/why-and-how-to-write-code-compatible-with-many-coq-versions.html, accessed: 2017-11-17

[40] Danielsson, N.A.: Total parser combinators. In: ACM Sigplan Notices. vol. 45, pp. 285–296. ACM (2010)

[41] Deutsch, P.: DEFLATE Compressed Data Format Specification version 1.3. RFC 1951 (Informational) (May 1996), http://www.ietf.org/rfc/rfc1951.txt

[42] Deutsch, P.: GZIP file format specification version 4.3. RFC 1952 (Informational) (May 1996), http://www.ietf.org/rfc/rfc1952.txt

[43] Deutsch, P., Gailly, J.L.: ZLIB Compressed Data Format Specification version 3.3. RFC 1950 (Informational) (May 1996), http://www.ietf.org/rfc/rfc1950.txt

[44] Dénès, M., Pédrot, P.M.: A proof of false, https://github.com/clarus/falso, accessed: 2017-11-17

[45] Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J.G.: A Fully Verified Executable LTL Model Checker, pp. 463–478. Springer Berlin Heidelberg, Berlin, Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-39799-8_31

[46] Fano, R.M.: The transmission of information. Massachusetts Institute of Technology, Research Laboratory of Electronics Cambridge, Mass, USA (1949)

[47] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard) (Jun 1999), http://www.ietf.org/rfc/rfc2616.txt, obsoleted by RFCs 7230, 7231, 7232, 7233, 7234, 7235, updated by RFCs 2817, 5785, 6266, 6585

[48] Floyd, R.: Assigning meanings to programs. In: Proceedings of the American Mathematical Society Symposia on Applied Mathematics. vol. 19, pp. 19–31 (1967)

[49] Fonseca, P., Zhang, K., Wang, X., Krishnamurthy, A.: An empirical study on the correctness of formally verified distributed systems. In: Proceedings of the Twelfth European Conference on Computer Systems. pp. 328–343. EuroSys '17, ACM, New York, NY, USA (2017), http://doi.acm.org/10.1145/3064176.3064183

[50] Gennari, J.: The zlib compression library is vulnerable to a denial-of-service condition, http://www.kb.cert.org/vuls/id/238678, accessed: 2017-11-17

[51] Gennari, J.: zlib inflate() routine vulnerable to buffer overflow, http://www.kb.cert.org/vuls/id/680620, accessed: 2017-11-17

[52] Godefroid, P.: dynamic software model checking (sep 2014), https://patricegodefroid.github.io/public_psfiles/talk-emc2014.pdf, accessed: 2017-11-17

[53] Gödel, K.: Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme i. Monatshefte für mathematik und physik 38(1), 173–198 (1931)

[54] Gödel, V.K.: Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. dialectica 12(3-4), 280–287 (1958)

[55] Gonthier, G.: Formal proof–the four-color theorem. Notices of the AMS 55(11), 1382–1393 (2008)

[56] Google Inc.: Zopfli compression algorithm, https://github.com/google/zopfli, accessed: 2017-11-17

[57] Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12(10), 576–580 (1969)

[58] Hoffmann, J.: Resource aware ml, http://www.raml.co/, accessed: 2017-11-17

[59] Hollenbeck, S.: Transport Layer Security Protocol Compression Methods. RFC 3749 (Proposed Standard) (May 2004), http://www.ietf.org/rfc/rfc3749.txt

[60] Howard, W.A.: The formulae-as-types notion of construction (1969)

[61] Huffman, D.: A method for the construction of minimum-redundancy codes. Proceedings of the IRE 40(9), 1098–1101 (Sept 1952)

[62] Hupel, L.: private communication (2016)

[63] Jang, D., Tatlock, Z., Lerner, S.: Establishing browser security guarantees through formal shim verification. In: Proceedings of the 21st USENIX conference on Security symposium. pp. 8–8. USENIX Association (2012)

[64] Kanav, S., Lammich, P., Popescu, A.: A conference management system with verified document confidentiality. In: International Conference on Computer Aided Verification. pp. 167–183. Springer (2014)

[65] Kaplan, H., Tarjan, R.E.: Purely functional representations of catenable sorted lists. In: Proceedings of the twenty-eighth annual ACM symposium on Theory of computing. pp. 202–211. ACM (1996)

[66] Kaplan, H., Tarjan, R.E.: Purely functional, real-time deques with catenation. J. ACM 46(5), 577–603 (Sep 1999), http://doi.acm.org/10.1145/324133.324139

[67] Kelsey, J.: Compression and information leakage of plaintext. In: Fast Software Encryption, 9th International Workshop, FSE 2002, Leuven, Belgium, February 4-6, 2002, Revised Papers. Lecture Notes in Computer Science, vol. 2365, pp. 263–276. Springer (2002), http://www.iacr.org/cryptodb/archive/2002/FSE/3091/3091.pdf

[68] Kiselyov, O.: Iteratees. Functional and Logic Programming pp. 166–181 (2012)

[69] Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R., Heiser, G.: Comprehensive formal verification of an OS

microkernel. ACM Transactions on Computer Systems 32(1), 2:1–2:70 (feb 2014)

[70] Lawrence, A., Berger, U., Seisenberger, M.: Extracting a dpll algorithm (2012)

[71] Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM 52(7), 107–115 (2009), http://gallium.inria.fr/~xleroy/publi/compcert-CACM.pdf

[72] Malecha, G., Morrisett, G., Shinnar, A., Wisnesky, R.: Toward a verified relational database management system. In: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 237–248. POPL '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1706299.1706329

[73] Martin-Löf, P.: Constructive mathematics and computer programming. In: L. Jonathan Cohen, Jerzy Łoś, H.P., Podewski, K.P. (eds.) Logic, Methodology and Philosophy of Science VIProceedings of the Sixth International Congress of Logic, Methodology and Philosophy of Science, Studies in Logic and the Foundations of Mathematics, vol. 104, pp. 153 – 175. Elsevier (1982), https://www.sciencedirect.com/science/article/pii/S0049237X09701892

[74] McCarthy, J.: A basis for a mathematical theory of computation. Studies in Logic and the Foundations of Mathematics 35, 33–70 (1963)

[75] McMillan, B.: Two inequalities implied by unique decipherability. Information Theory, IRE Transactions on 2(4), 115–116 (December 1956)

[76] Miyamoto, K.: The minlog system, http://minlog-system.de/, accessed: 2017-11-17

[77] Monin, J.F.: A toolkit to reason with programs raising exceptions, https://github.com/coq-contribs/continuations, accessed: 2017-11-17

[78] Moravec, H.: Mind children: The future of robot and human intelligence. Harvard University Press (1988)

[79] Morrisett, G., Tan, G., Tassarotti, J., Tristan, J.B., Gan, E.: Rocksalt: better, faster, stronger sfi for the x86. In: ACM SIGPLAN Notices. vol. 47, pp. 395–404. ACM (2012)

[80] Naumowicz, A., Korniłowicz, A.: A Brief Overview of Mizar, pp. 67–72. Springer Berlin Heidelberg, Berlin, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-03359-9_5

[81] Nogin, A.: Writing constructive proofs yielding efficient extracted programs. Electronic Notes in Theoretical Computer Science 37, 1–17 (2000)

[82] Postel, J.: DoD standard Transmission Control Protocol. RFC 761 (Jan 1980), http://www.ietf.org/rfc/rfc761.txt, obsoleted by RFC 793

[83] Ricketts, D., Robert, V., Jang, D., Tatlock, Z., Lerner, S.: Automating formal proofs for reactive systems. In: O'Boyle, M.F.P., Pingali, K. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014. p. 47. ACM (2014), http://doi.acm.org/10.1145/2594291.2594338

[84] Roelofs, G., Gailly, J., Adler, M.: A massively spiffy yet delicately unobtrusive compression library, http://www.zlib.net/, accessed: 2017-11-17

[85] Sacchini, J.L.: Exceptions in dependent type theory, https://www.irif.fr/~letouzey/types2014/abstract-18.pdf, accessed: 2017-11-17

[86] Schwichtenberg, H., Senjak, C.: Minimal from classical proofs. Annals of Pure and Applied Logic 164(6), 740–748 (2013)

[87] Scott, D.S.: A type-theoretical alternative to iswim, cuch, owhy. Theoretical Computer Science 121, 411–440 (1993), annotated version of the 1969 manuscript

[88] Senjak, C.S.: Minimal from classical proofs, https://uxul.de/akademisches/diplom.pdf, accessed: 2017-11-17

[89] Senjak, C.S., Hofmann, M.: An implementation of deflate in coq. In: FM 2016: Formal Methods: 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings 21. pp. 612–627. Springer (2016)

[90] Senjak, C., Hofmann, M.: An implementation of deflate in coq. CoRR abs/1609.01220 (2016), http://arxiv.org/abs/1609.01220

[91] Storer, J.A., Szymanski, T.G.: Data compression via textual substitution. J. ACM 29(4), 928–951 (Oct 1982), http://doi.acm.org/10.1145/322344.322346

[92] Swierstra, W.: The hoare state monad, http://www.cs.ru.nl/~wouters/Publications/HoareLogicStateMonad.pdf, accessed: 2017-11-17

[93] The Coq Development Team: The coq proof assistant reference manual, https://coq.inria.fr/distrib/current/refman/, accessed: 2017-11-17

[94] Thery, L.: Formalising huffman's algorithm. Tech. rep., Tech. report TRCS 034, Dept. of Informatics, Univ. of L'Aquila (2004)

[95] Turing, A.M.: On computable numbers, with an application to the entscheidungsproblem. Proceedings of the London mathematical society 2(1), 230–265 (1937)

[96] Turing, A.: Checking a large routine. EDSAC Inaugural Conference (1949)

[97] Univalent Foundations Program, T.: Homotopy Type Theory: Univalent Foundations of Mathematics. https://homotopytypetheory.org/book, Institute for Advanced Study (2013)

[98] Vafeiadis, V.: Adjustable references. In: Proceedings of the 4th International Conference on Interactive Theorem Proving. pp. 328–337. ITP'13, Springer-Verlag, Berlin, Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-39634-2_24

[99] Vafeiadis, V.: Adjustable references. In: International Conference on Interactive Theorem Proving. pp. 328–337. Springer (2013)

[100] Voevodsky, V.: Notes on homotopy lambda calculus (2006), https://github.com/vladimirias/2006_03_Homotopy_lambda_calculus, accessed: 2017-11-17

[101] der Wiskunde, T.H.E.O., Dijkstra, E.W.: Notes on structured programming (1969)

[102] Zakharov, I.S., Mandrykin, M.U., Mutilin, V.S., Novikov, E., Petrenko, A.K., Khoroshilov, A.V.: Configurable toolset for static verification of operating systems kernel modules. Programming and Computer Software 41(1), 49–64 (2015)

[103] Zhao, J., Nagarakatte, S., Martin, M.M., Zdancewic, S.: Formalizing the llvm intermediate representation for verified program transformations. In: ACM SIGPLAN Notices. vol. 47, pp. 427–440. ACM (2012)

[104] Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Transactions on information theory 23(3), 337–343 (1977)