Formal foundations for the Pocket+ algorithm: the RLE component

Cláudia Faria, Patrícia Bastos, and José N. Oliveira

University of Minho & INESC TEC, Braga, Portugal

Abstract. POCKET+ is a lossless compression algorithm for spacecraft housekeeping telemetry using low-level bitwise operations only. It is the CCSDS Recommended Standard For Robust Compression of Fixed-Length Housekeeping Data.
There already exist implementations of this non-trivial protocol but a formal proof of its correctness is yet to be made. Proving that this compression algorithm is correct means showing that after the compression of a series of inputs, the outputs upon decompression are the same. That is to say, no information is lost or incorrectly reconstructed in the case of packet losses.
This paper describes a high-level specification of POCKET+, written in Haskell, to be used as a reference for testing and validation, and gives a formal proof of correctness of one of its main components – the 'Run Length Encoder' (RLE) – using the relational calculus.
This work is funded by national funds through FCT – Fundação para a Ciência e a Tecnologia, I.P., under the support UID/50014/2025 (https://doi.org/10.54499/UID/50014/2025).

Keywords: Formal methods · Aerospace software · Safety-critical systems.

1 Introduction

POCKET+ is a compression algorithm for spacecraft housekeeping telemetry [7], designed to compress fixed-length packets on the fly using only bit-level operations. This makes it lightweight enough to run on flight hardware, and its built-in robustness to packet loss makes it suitable for the unreliable communication links typical of space missions. It is the CCSDS Recommended Standard For Robust Compression of Fixed-Length Housekeeping Data [2].

The main advantages of this algorithm are its fast reactivity, which reduces CPU usage and increases speed; increased efficiency compared to other compressions methods; insensitivity to multiple sequential packet losses in both compression and decompression operations; performance fast enough for use on real time data streams due to the almost exclusive use bitwise operations; and compression configuration agnostic decompression [3].

The highlight of the protocol is its robustness to packet loss, which is achieved through a resynchronization process during the decoding. When a packet is lost,

the Decoder can use the information from the subsequent packets to recover the original input vector.

Despite being a mission-critical protocol, there is currently no formal verification of its correctness. This project, suggested by Telespazio, aims to provide such a proof by implementing the protocol in Haskell, whose mathematical foundations make it easier to translate the definitions of the standard [2] into algebraic specifications. Relational algebra [1] will then be used to assist and provide the algorithm's correctness.

2 Background and Related Work

There already exist POCKET+ implementations: in C++ by Vision Space [13]; C, C++, Python, Go, Rust, and Java by Tanagra Space [12], the latter having even been sent on ESA's OPS-SAT mission [4]. Yet it is not formally verified. The effect of flags on the standard's compression efficiency has also been analyzed [6]. The main goal of the project reported in this paper is to provide a Haskell implementation that can be formally verified.

Haskell was chosen to implement the protocol due to its strong type system and mathematical foundations, which make it easier to translate the code into algebraic specifications and reason about its properties. Even if the reader is not fluent in Haskell, the code, fully available online [5] and equipped with diagrams, should be easy to understand and follow.

This implementation was done from scratch, following the standard specifications [2] and using the C++ implementation as a reference. The Haskell implementation was not derived from the C++ code's internal logic, only checked against its outputs for quick testing.

Validation of the Haskell implementation is ongoing, but preliminary results are consistent with the reference C++ and can be checked in [5]. Proving its correctness means showing that after the compression of a certain input, the output of its decompression does not differ from that input. This means that no information is lost or incorrectly reconstructed.

3 The Protocol

The POCKET+ protocol is a lossless compression algorithm designed for fixed-length housekeeping data. Its goal is to efficiently encode sequences of binary input vectors by identifying redundancy between successive packets. The protocol consists of four main components described in Table 1.

3.1 Parameters and Definitions

The protocol's parameters and definitions are essential for understanding how the compression process works (Fig. 1) and provide the necessary context for interpreting the operations performed by the Encoder and Decoder.

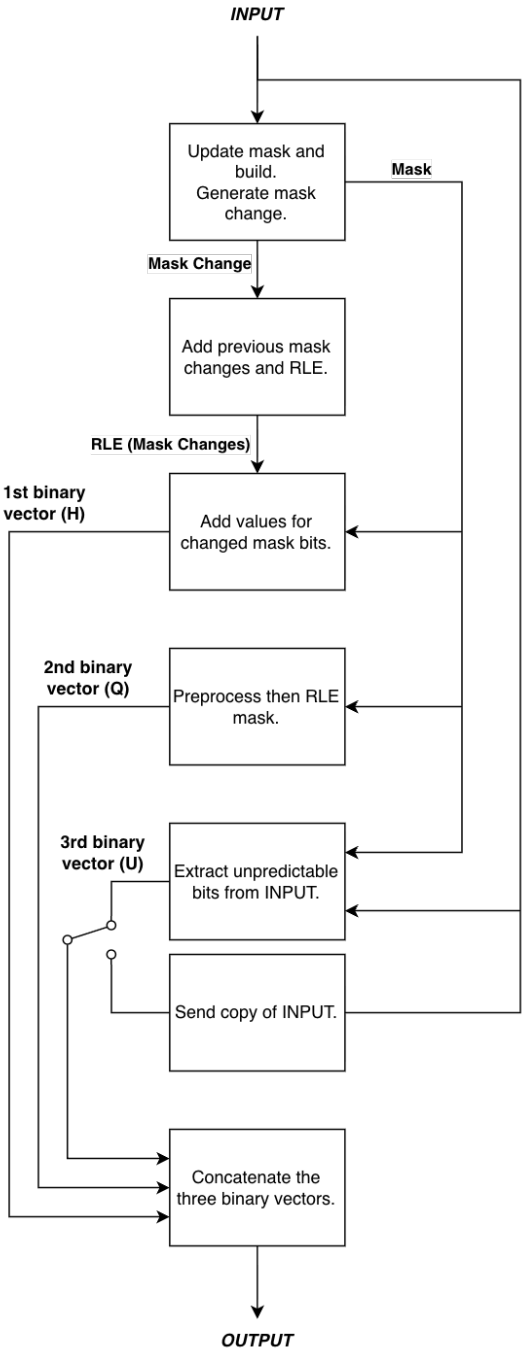

Fig. 1. Overview of Compression Process, showing the output vector components H, Q e U. Adapted from [2].

Table 1. The core of the protocol

Component	Purpose
Flags and Parameters	Mostly user-specified, determine the protocol's behaviour
Mask Update	Identifies predictable bits
RLE	Packet encoding function
Resynchronization	Restores lost information

The user can choose to specify certain parameters that influence the behavior of the compressor. These parameters allow for flexibility and can be adjusted based on the specific requirements of the data. Two important parameters are always user-defined and essential for the packet loss robustness properties of the protocol. One is the flag that controls the inclusion of the entire input vector in the output, which allows for a trade-off between compression efficiency and the ability to recover from data loss. On the other hand, by only including the unpredictable bits, the compressor can achieve better compression ratios, but it may be more susceptible to data loss.

The other important parameter is the minimum required effective robustness level, which allows the user to specify how many consecutive output vectors can be lost without affecting the ability to decompress the data and providing a guarantee of the compressor's resilience to packet loss.

3.2 Mask Update

The core idea of POCKET+ is that most housekeeping telemetry values do not change much between successive packets.

The Mask Update operation (Fig. 1) identifies redundancy (the so-called *predictable bits*) between the packets. It is based on Differential pulse-code modulation (DPCM), which changes between consecutive samples of a signal, rather than the signal's value directly [8]. DPCM is then decoded by integrating DPCM samples over time.

To better understand the concept behind the Mask Update, we can think of some kind of "discrete derivative" of the input with respect to change (Fig. 2). The Mask Update identifies which bits of the input vector are changing, and which are not, by comparing the current input vector with the previous one. The bits that are changing are marked as unpredictable by the Mask, while the bits that remain the same are marked as predictable. This information is then used in the encoding process to efficiently compress the data by only encoding the unpredictable bits.

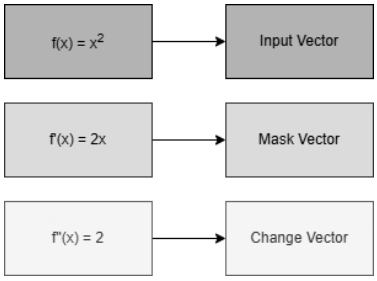

Fig. 2. Mask Update and the "discrete derivative" metaphor

The Change vector, which indicates which bits of the mask have changed between cycles, can be thought of as a "second derivative" of the input vector, as it captures the changes in the predictability of the bits over time.

3.3 Encoding

Run-length Encoding (RLE) [10] is a common and simple form of lossless data compression in which consecutive occurrences of the same data are stored as a single data value and count. Once the unpredictable bits have been isolated, RLE is applied to encode the changes of the mask itself (our "second derivative"), which indicates which bits of the mask have changed between cycles. The Change vector will have long runs of 0s (predictable bits) that RLE will handle efficiently.

In the Encoding, the output vector is a concatenation of three variable-length binary vectors H, Q and U — see Fig. 1 — which encode different information about the input vector and the mask. The exact content of these vectors depends on the user-specified parameters and the current state of the mask, but always include the information about the mask changes, the effective robustness level, and the unpredictable bits.

4 Formal Model

The first observation about this protocol is that the Encoder is a finite state machine, split into the Mask Update and the Encoding stages. The Mask Update calculates the Mask, Build, and Change vectors, which make up the cycle's state, while the Encoding stage calculates the output vector based on the input vector and the compressor parameters.

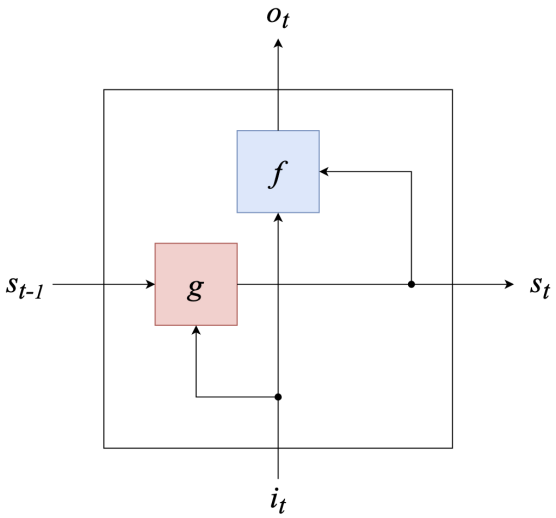

Fig. 3. Mealy Machine

Essentially, at each cycle, the protocol creates a new state based on the previous state and the current input, and then produces an output based on that new state and the current input. This behaviour is that of a Mealy Machine.

A Mealy Machine is a special type of finite automaton where the output is determined by both the current state and the current input. It therefore is made of two functions:

– **State transition function:** defines how the machine moves from one state to another depending on the input.

– **Output production function:** specifies the output produced from the current state and the current input.

This is shown in Fig. 3. Formally, our Mealy Machine is defined by

$$m = \langle \pi_1, f \rangle \cdot \langle g, \pi_2 \rangle \tag{1}$$

using functional sequencial composition, $(f \cdot g)\ x = f\ (g\ x)$, parallel functional composition $\langle f, g \rangle\ x = (f\ x, g\ x)$ and the two Cartesian projection functions: $\pi_1\ (s, x) = s$ and $\pi_2\ (s, x) = x$ [1,11].

Function g, which is responsible for the state transition ($s_{t-1} \rightarrow s_t$), acts before f produces the output o_t. Both are triggered by the current input i_t. In more detail, $\langle g, \pi_2 \rangle$ takes the current state and input and produce the new state, followed by $\langle \pi_1, f \rangle$, which takes the new state and input and produces the output.

Each cycle of the Encoder can be seen as application of m (1) for the current input vector and state, where the latter contains the Mask, Build and Change vectors along with the last Input vector, vector length, and parameters. The output is the concatenation of the H, Q and U vectors, see Fig. 1.[1]

As expected, the Decoder will also behave as a Mealy Machine with the same state, but its inputs are encoded vectors to be decoded, and its outputs will be the original input vectors recovered.

5 Correctness

The goal of the POCKET+ protocol is to ensure that the output of the Decoder is always the same as the original input vector that was fed to the Encoder. Formally, this can be stated as:

$$y = encoder\ x \Rightarrow decoder\ y = x \tag{2}$$

which is actually equivalent to

$$decoder \cdot encoder = id \tag{3}$$

where $id\ x = x$ is the identity function. Should (2,3) hold, then *encoder* will be an injective function (no loss of information when encoding) and *decoder* will be a surjective function (every input has an encoding) [11]. Then *decoder* is said to be a left inverse of *encoder*.

Since, when starting this correctness exercise, we do not know whether (2) holds or not, it is wiser to regard the functions in (2) as *relations* rather than pure functions. Such is the principle behind *relation algebra*, the formal calculus that will be used in the sequel and which has a long tradition in calculating functional programs [1,11].

[1] In *Part III — Haskell code* of [5], functions f_{Enc}/g_{Enc} make up the Encoder's Mealy machine and f_{Dec} / g_{Dec} do the same for the Decoder.

A main advantage of relations is the fact that they can always be reversed: given any relation $b\ R\ a$, its converse, denoted R°, relates b and a in the opposite direction, $a\ R^\circ\ b$.[2] This will be applicable to our case study as Decoder and Encoder work in converse directions.

However, knowing that the Encoder includes data compression, how can one be sure such a compression is lossless (injective)? A good way to address this question is to look at one of the core components of the POCKET+ algorithm: the Run-length Encoder (RLE).

6 Run-length Encoding

The RLE component of the Encoder is responsible for compressing the input vector by encoding runs of consecutive bits. It is defined in Haskell [5] as the following pipeline:[3]

$$\begin{aligned} &rle :: [Bit] \to [Bit] \\ &rle = counting \cdot init \cdot map\ (+1) \cdot map\ length \cdot splitOn\ [1] \end{aligned} \tag{4}$$

In words, it splits the input into blocks of consecutive zeros ($splitOn\ [1]$), counts their length and encodes such counts in binary ($counting$). Compression pays off wherever long blocks of consecutive zeros are frequent. Fig. 4 provides an illustration of such run-to-count encoding, taken from [2].

Reversing rle means finding some $unrle$ such that $unrle \cdot rle = id$. In a sense, it amounts to calculating the reverse pipeline based on the law $(f \cdot g)^\circ = g^\circ \cdot f^\circ$. One can find (left) inverses of some of the components of (4), for instance (Haskell):

$$(intercalate\ a) \cdot (splitOn\ a) = id \tag{5}$$

$$(subtract\ 1) \cdot (+1) = id \tag{6}$$

But others, e.g. $init$ (keep all but the last element) are not injective: how can one recover the last element of a list after throwing it away? Laws such as eg.

$$length \cdot (flip\ replicate\ a) = id$$

are useful but do not help here because they work in the opposite direction. The answer lies in finding strategies for reversing non injective functions.

Reversibility. Lossless compression implies no information is lost during the encoding and decoding process. This makes reversibility our primary concern, which can be ensured with so-called *minimal complements* [9]. In essence, a

[2] By contrast, only the converses of bijective (i.e. injective and surjective) functions are functions.

[3] The strategy is to look at the definition of the RLE function in the standard [2] and see how it can be decomposed into smaller functions that are easier to analyse and prove correct when compared to the C++ code.

0001 000001 000001 00001 0000001 1001 0000000000000000

$C_0 = 4, \quad C_1 = 6, \quad C_2 = 1, \quad C_3 = 6, \quad C_4 = 5, \quad C_5 = 7, \quad C_{H(\mathbf{a})-1} = 3$

Fig. 4. Example of the Run-to-Count Encoding, converting a binary vector α into a sequence of integers.

complement of a non-injective function f is another function g that contains enough of the information lost by f for the pairing $\langle f, g\rangle$ to be injective. Thais is, $\langle f, g\rangle$ is injective even if neither f nor g are so. A minimal complement contains *just* the information that was lost.

A good example is the non-injective operation of concatenating two lists,

$$conc\ (x, y) = x +\!\!+ y \tag{7}$$

Its minimal complement is $length \cdot \pi_1$, the length of its first argument.[4] That is, $\langle length \cdot \pi_1, conc\rangle$ is injective, its left-inverse being $\widehat{nspan}$, where $\widehat{f}\ (a, b) = f\ a\ b$ denotes the *uncurrying* of f and:

$$nspan\ n = \langle take\ n, drop\ n\rangle \tag{8}$$

In the case of *rle*, the last block of zeros thrown away by *init* (4) is recovered by knowing the length of the original vector. Each stage of unrle inverts the matching stage of rle in reverse order: map (subtract 1) undoes map (+1); map (flip replicate 0) undoes map length; intercalate [1] undoes splitOn [1]; and trail n recovers the final block init discarded, using the known total length n as the minimal complement described above.

```
unrle :: [Bit] → Int → [Bit]
unrle = flip pipeline where
  pipeline n = intercalate [1] · map (flip replicate 0) · map (subtract 1) ·
    trail n · uncounting
  trail n s = s ++ [n − (sum s + length s)]
```

Then, by attaching *length* as a minimal complement to the *rle* function,

```
rle′ :: ([Bit] → ([Bit], Int))
rle′ = ⟨rle, length⟩
```

we can obtain its left inverse as:

```
unrle′ :: ([Bit], Int) → [Bit]
unrle′ = ^unrle
```

However, the formal proof that $unrle' \cdot rle' = id$ holds is far from trivial. As the pipeline of *rle* is inverted step by step, one is left with the *counting* and *uncounting* functions to be proven inverses of each other. A strategy to do this is to invert *counting* to obtain *uncounting*. As *counting* is a recursive function (a list-fold) its inversion showcases how relational algebra reverses recursive programs.

[4] With this length, we know where the concatenation happened, and we can recover the original lists by splitting on that index — see (8).

7 Reversing counting

The last stage of the *rle* pipeline — $counting :: [Int] \to [Bit]$ in (4) — converts a list of positive integers into a bit vector via some smart encoding, to be explained shortly. The plan is to obtain its left inverse *uncounting*, such that

$$uncounting \cdot counting = id \tag{9}$$

holds. It can be shown that (9) is equivalent to

$$counting^{\circ} \subseteq uncounting \tag{10}$$

where, in general, $R \subseteq S$ denotes relation inclusion.[5] So, once we have a definition of *counting* and reverse it, we get a lowerbound for *uncounting*.

Let us start by seeing the way each count number is encoded bit-wise in the standard. It is based on marking four different cases using four disjoint prefixes:[6]

$$\begin{aligned}
&inj_0 = ([0]+\!\!+) \\
&inj_{10} = ([1,0]+\!\!+) \\
&inj_{110} = ([1,1,0]+\!\!+) \\
&inj_{111} = ([1,1,1]+\!\!+)
\end{aligned}$$

These are used in the function that performs the conversion:

$$\begin{aligned}
&count :: Int \to [Bit] \\
&count\ 1 = inj_0\ [\,] \\
&count\ a \\
&\quad \mid a \geqslant 2 \wedge a \leqslant 33 = inj_{110}\ (toNBits\ 5\ (a-2)) \\
&\quad \mid otherwise = inj_{111}\ (toNBits\ (e\ a)\ (a-2))
\end{aligned}$$

where $toNBits\ n\ x$ encodes number x using n bits. Function e calculates the number of bits needed to represent its argument in binary in a way that is suitable for decoding.[7] Then *counting* is a list-fold that uses *count*:

$$\begin{aligned}
&counting :: [Int] \to [Bit] \\
&counting = foldr\ f\ i\ \textbf{where} \\
&\quad i = inj_{10}\ [\,] \\
&\quad f\ n\ bs = count\ n +\!\!+ bs
\end{aligned}$$

What about *uncounting*? As a left-inverse of *counting*, it is expected to be an *unfoldr*, e.g.

$$\begin{aligned}
&uncounting :: [Bit] \to [Int] \\
&ucounting = unfoldr\ g
\end{aligned}$$

[5] $R \subseteq S$ means that $b\ R\ a$ logically implies $b\ S\ a$, for all a and b.

[6] These prefixing operations are all injective, thus the common prefix *inj*.

[7] For the details of this 'smart' encoding, interesting but not relevant for the moment, see function E in [2].

for some $g :: [Bit] \rightarrow Maybe\ (Int, [Bit])$.

Recalling the Haskell datatype *Maybe*, $g\ v$ will read bit-vector v and either issue *Nothing* to stop the process or issue *Just* (i, r), where i is the number just *parsed* from v and r is the *rest* of v to consider in the next iteration of *uncounting*. It can thus be regarded as a bit-level *parser* yielding a list of numbers.

However, how does one calculate g?

Going pointfree. The calculation of g presented in [5] is too long to be presented here. Therefore, we will emphasize the essence of the calculation strategy and omit the details.

An important feature of the reasoning is the change of style and the adoption of *pointfree* definitions, that is, functional definitions that do not require variables. These have the advantage of being more agile in calculations. In this vein, let us rephrase the *foldr* combinator as follows:

$$foldr\ f\ a = (\!|\, [\widehat{f}, \underline{a}] \,|\!) \tag{11}$$

Notation $(\!|\_|\!)$ is typical of so-called *catamorphisms* [1,11] and notation $\underline{a}$ is used for constant functions: $\underline{a}\ x = a$ for all x. It remains to show what the combinator $[f, g]$ means. In Haskell, it is written `either f g` and defined over the *Either* data type constructor.

We use the square bracket notation $[f, g]$ because it is more compact and we need to generalize it to relations:

$$[R, S] = R \cdot i_1^\circ \cup S \cdot i_2^\circ \tag{12}$$

Here $X \cup Y$ denotes relation *union* and injective functions i_1 and i_2 are convenient abbreviations of constructors *Left* and *Right* of the *Either* datatype. Using these pointfree combinators, we can redefine *counting* as follows,

$$\begin{aligned}
&counting :: [Int] \rightarrow [Bit] \\
&counting = (\!|\, [l, r] \,|\!)\ \textbf{where} \\
&\quad l = inj_{10} \cdot nil \\
&\quad r = conc \cdot (count \times id)
\end{aligned}$$

where $nil = [\,]$, *conc* has already been defined (7) and the parallel combinator $(\times)$ is defined by $(f \times g)\ (x, y) = (f\ x, g\ y)$.

Catamorphisms generalize to relations [1,11] and obey the following law,

$$(\!|\, R \,|\!)^\circ = [\!(R^\circ)\!] \tag{13}$$

where $[\!(R)\!] = unfoldr\ R$ generalizes *unfoldr* to relations. This generalization is called *anamorphism*.[8]

[8] These concepts are very general, as they are defined for *inductive types* which, in our case, are confined to finite lists.

All this said, we go back to (10) and instantiate it with our definitions, where the unkown is *uncounting*:

$$
\begin{array}{ll}
& \mathit{counting}^\circ \subseteq \mathit{uncounting} \\
\equiv & \quad \{ \text{ definition of } \mathit{counting} \ \} \\
& (\!|\, [l, r] \,|\!)^\circ \subseteq \mathit{uncounting} \\
\equiv & \quad \{ \ (13) \ \} \\
& [\!(\, [l, r]^\circ \,)\!] \subseteq \mathit{uncounting} \\
\equiv & \quad \{ \ (16); (12); (21); (17) \text{ twice (please see the appendix)} \ \} \\
& [\!(i_1 \cdot l^\circ \cup i_2 \cdot r^\circ)\!] \subseteq \mathit{uncounting} \\
\equiv & \quad \{ \text{ definitions of } l \text{ and } r \ \} \\
& [\!(i_1 \cdot \mathit{nil}^\circ \cdot \mathit{inj}_{10}^\circ \cup i_2 \cdot (\mathit{count}^\circ \times \mathit{id}) \cdot \mathit{conc}^\circ)\!] \subseteq \mathit{uncounting} \\
\equiv & \quad \{ \text{ make } \mathit{uncounting} = [\!(\mathit{parse})\!] \text{ for some } \mathit{parse} \text{ to be calculated } \} \\
& [\!(i_1 \cdot \mathit{nil}^\circ \cdot \mathit{inj}_{10}^\circ \cup i_2 \cdot (\mathit{count}^\circ \times \mathit{id}) \cdot \mathit{conc}^\circ)\!] \subseteq [\!(\mathit{parse})\!] \\
\Leftarrow & \quad \{ \text{ monotonicity } \} \\
& i_1 \cdot \mathit{nil}^\circ \cdot \mathit{inj}_{10}^\circ \cup i_2 \cdot (\mathit{count}^\circ \times \mathit{id}) \cdot \mathit{conc}^\circ) \subseteq \mathit{parse} \qquad (14)
\end{array}
$$

Note the change of unkown, which now is *parse* — if it exists. Report [5] shows that it does exist and is the function:

$$
\begin{array}{l}
\mathit{parse}\ [1, 0] = \mathit{Nothing} \\
\mathit{parse}\ (0 : r) = \mathit{Just}\ (1, r) \\
\mathit{parse}\ (1 : (1 : (0 : r))) = \mathit{Just}\ (\mathit{fromBinary}\ a + 2, r') \\
\quad \textbf{where}\ (a, r') = \mathit{nspan}\ 5\ r \\
\mathit{parse}\ (1 : (1 : (1 : r))) = \mathit{Just}\ (\mathit{fromBinary}\ a + 2, r') \\
\quad \textbf{where}\ (a, r') = \mathit{nspan}\ (6 + \mathit{length}\ x)\ r''; (x, r'') = \mathit{cspan}\ (\equiv 0)\ r \\
\mathit{parse}\ \_ = \mathit{Nothing}
\end{array}
$$

where $\mathit{cspan}\ p = \langle \mathit{takeWhile}\ p, \mathit{dropWhile}\ p \rangle$ and *nspan* has given by (8).

The *correct-by-construction* nature of the above exercise should be emphasized: instead of postulating *parse* and *proving* that $[\!(\mathit{parse})\!]$ meets its specification (9), it is *derived* directly from this using a program *calculus* [1].

Space constraints prevent us from presenting the calculation leading to the result above. One detail should nevertheless be given: to carry on the exercise above, we need a pointfree definition of *count* which, as seen above, is defined by cases in Haskell. How does one express conditional definitions such as

$$
\begin{array}{l}
h \mid p\ x = f\ x \\
\quad \mid \mathit{otherwise} = g\ x
\end{array}
$$

without writing x? Let predicate p be represented by the relation $\mathbf{\Phi}_p$ such that $a'\ \mathbf{\Phi}_p\ a \ \Leftrightarrow\ a' = a \wedge p\ a$. Then $h = f \cdot \mathbf{\Phi}_p \cup g \cdot \mathbf{\Phi}_{\neg\, p}$, where $\mathbf{\Phi}_p$ (resp. $\mathbf{\Phi}_{\neg\, p}$) is the

guard of the then-branch (resp. else-branch) of the conditional. It also extends to relations, in which case one has a non-deterministic conditional.

An equivalent definition of *count* that is more amenable to calculation is:

$$
\begin{aligned}
&count :: Int \to [\,Bit\,] \\
&count = [inj_0 \cdot nil, g_2] \cdot [\underline{0}, succ]^\circ \cdot pred \\
&\quad \textbf{where} \\
&\quad g_2\ a \\
&\qquad \mid a \geqslant 0 \wedge a \leqslant 31 = inj_{110}\ (toNBits\ 5\ a) \\
&\qquad \mid otherwise = inj_{111}\ (toNBits\ (e\ a)\ a) \\
&\quad e\ a = 2 * (minb\ a) - 6 \\
&\quad minb\ a = \lfloor log_2(fromIntegral\ a) + 1 \rfloor
\end{aligned}
$$

where $succ\ n = n + 1$ and $pred \cdot succ = id$. Then, assuming defined $tnb\ f = \widehat{toNBits} \cdot \langle f, id \rangle$, $pre = \lceil 2^5 \rceil$ and $\lceil n \rceil = (\in \{0\,..\,n-1\})$, one has

$$
count^\circ = \begin{cases} \quad i_1 \cdot nil^\circ \cdot inj_{10}^\circ \\ \cup\ i_2 \cdot (\underline{1} \times id) \cdot (inj_0 \cdot conc \cdot (nil \times id))^\circ \\ \cup\ i_2 \cdot ((2+) \cdot (tnb\ \underline{5} \cdot \Phi_{pre})^\circ \times id) \cdot (inj_{110} \cdot conc)^\circ \\ \cup\ i_2 \cdot ((2+) \cdot (tnb\ e \cdot \Phi_{\neg\ pre})^\circ \times id) \cdot (inj_{111} \cdot conc)^\circ \end{cases} \tag{15}
$$

Substituting (15) in (14) and performing a number of standard algebraic calculations, one reaches *parse* as given above.

It can be observed how the disjointness of the bit-level injections (inj_{10}, inj_{111}, and so on) concur for the determinisation of *parse*. Otherwise, such bit-level parsing would be non-deterministic. The missing details of the calculation of *parse* are provided on-line in [5].

8 Conclusion

This paper contributes with a formal specification of the POCKET+ protocol in the form of an implementation in Haskell to be used as a reference for testing and validation. The ultimate aim of this project is the formal correctness of the whole protocol. This paper and accompanying report [5] focus on the correctness of the RLE component only.

The encoder and decoder are not perfect mirrors of each other, as the protocol's design prioritizes compression performance and robustness over simplicity. The 124.0-B-1 Standard [2] itself is not a simple read and the presence of user-specified parameters and robustness adjustments makes the implementation and proof of correctness more difficult.[9]

Having struggled with the standard, the authors tried to create a formal (and also runnable) specification of the protocol, complete with descriptive diagrams to aid in comprehension. This description, along with the full code, can be found in [5], with hope that it can be used as a reference for future work on the protocol.

[9] As shown in this paper, RLE correctness is independent of such user-specified parameters.

The Mealy Machine structure of the protocol operation was translated into pairs of functions, one for the state transition and another for the output generation. This structure allowed us to reason about the protocol's behavior in a modular way, analyzing the state transitions and output generation separately.

9 Future Work

The next step in this implementation is cross-validation for compliance with the four corpora of housekeeping telemetry data defined by the CCSDS Data Compression working group. Future work regarding the proof of correctness includes completing the existing proofs and further exploring the role of the user-specified parameters in the protocol's behavior and correctness. Additionally, the resynchronization process should be implemented, as a crucial step to the conclusion of this implementation.

This work lays the foundation for a complete proof of correctness for the POCKET+ protocol, and perhaps encourage further research into the formal verification of similar algorithms with critical applications, where correctness is of utmost importance.

Acknowledgments. We sincerely thank Nuno Carvalho (Telespazio), Milenko Starcik (VisionSpace) and David Evans (ESA) for their support of this project.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Bird, R., de Moor, O.: Algebra of Programming. Prentice-Hall (1997), iSBN: 978-0-13-507245-5
2. Consultative Committee for Space Data Systems: Robust compression of fixed-length housekeeping data. Recommended Standard CCSDS 124.0-B-1, CCSDS (2023)
3. European Space Agency: Data compression and decompression for remote monitoring and control (pocket+), https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Data_Compression_and_Decompression_for_Remote_Monitoring_and_Control_POCKET
4. Evans, D., Labreche, G., Marszk, D., Bammens, S., Hernandez-Cabronero, M., Zelenevskiy, V., Shiradhonkar, V., Starcik, M., Henkel, M.: Implementing the new ccsds housekeeping data compression standard 124.0-b-1 (based on pocket+) on ops-sat-1. In: Proceedings of the Small Satellite Conference (2022), https://digitalcommons.usu.edu/smallsat/2022/all2022/133/, sSC22-XII-03
5. Faria, C., Bastos, P.: Pocket-plus (2026), technical report, Project in Formal Methods, 1st year MSc, U. Minho, https://github.com/claaaaaudia/Pocket-Plus
6. Hernández-Cabronero, M., Evans, D., Bartrina-Rapesta, J., Aulí-Llinàs, F., Blanes, I., Serra-Sagristà, J.: Resiliency and Efficiency of the CCSDS 124.0-B-1 Telemetry Compression Standard. IEEE Access **12**, 36702–36711 (2024). https://doi.org/10.1109/ACCESS.2024.3374227

7. Martinez-Heras, J., Evans, D., Timm, R.: Housekeeping telemetry compression: When, how and why bother? In: 2009 First International Conference on Advances in Satellite and Space Communications. pp. 35–40 (2009). https://doi.org/10.1109/SPACOMM.2009.30
8. MathWorks: Differential pulse code modulation (2026), https://www.mathworks.com/help/comm/ug/differential-pulse-code-modulation.html
9. Neri, A., Barbosa, R., Oliveira, J.: Compiling quantamorphisms for the ibm q experience. IEEE Transactions on Software Engineering **48**(11), 3–9 (2022)
10. Nguyen-Phi, K., Weinrichter, H.: A new binary source coder and its application in bi-level image compression. In: Proceedings of GLOBECOM'96. 1996 IEEE Global Telecommunications Conference. vol. 3, pp. 1483–1487 vol.3 (1996). https://doi.org/10.1109/GLOCOM.1996.591888
11. Oliveira, J.N.: Program Design by Calculation (2024), unpublished textbook draft, Sep. 2024 (313 p.). Informatics Dept., U.Minho (PDF).
12. Tanagra Space: (2026), multi-language implementations of CCSDS 124.0-B-1 in C, C++, Python, Go, Rust, and Java., https://github.com/tanagraspace/ccsds124
13. Vision Space: Pocketplus (2023), https://github.com/visionspacetec/PocketPlus

Appendix — Some laws of relation algebra

The following rules of relation algebra [11] are used in the correctness arguments of the main text.

Involution	$(R^\circ)^\circ = R$	(16)
Contravariance	$(R \cdot S)^\circ = S^\circ \cdot R^\circ$	(17)
Shunting Rules	$R \cdot f^\circ \subseteq S \equiv R \subseteq S \cdot f$	(18)
	$f \cdot R \subseteq S \equiv R \subseteq f^\circ \cdot S$	(19)
Relation Union Linearity	$R \cdot (S \cup Q) = R \cdot S \cup R \cdot Q$	(20)
Converse-$\cup$	$(R \cup S)^\circ = R^\circ \cup S^\circ$	(21)
Converse-coreflexive	$\Phi_p^\circ = \Phi_p$	(22)
Functor-$\times$-converse	$(R \times S)^\circ = R^\circ \times S^\circ$	(23)
$+$-fusion	$R \cdot [S, Q] = [R \cdot S, R \cdot Q]$	(24)
Divide and Conquer	$[R, S] \cdot [Q, U]^\circ = (R \cdot Q^\circ) \cup (S \cdot U^\circ)$	(25)