Horn Binary Serialization Analysis

Gabriele Paganelli
https://gapag.noblogs.org/
gapag@distruzione.org

A bit layout is a sequence of fields of certain bit lengths that specifies how to interpret a serial stream, e.g., the MP3 audio format. A layout with variable length fields needs to include meta-information to help the parser interpret unambiguously the rest of the stream; e.g. a field providing the length of a following variable length field. If no such information is available, then the layout is ambiguous. I present a linear-time algorithm to determine whether a layout is ambiguous or not by modelling the behaviour of a serial parser reading the stream as forward chaining reasoning on a collection of Horn clauses.

1 Introduction

Programs can read data from files or network interfaces in serial form. Data might not be available at once, or its consumption might be inherently sequential, such as in digital music. Programs decode the data stream interpreting it through a structure that defines the layout, or binary format, of the bits within the stream. Call this process *deserialization*, or *parsing*, or *unmarshalling* interchangeably. Among the reasons to use ad-hoc binary formats there are: *a)* Conciseness over verboseness of ASCII-based exchange formats like XML or JSON; *b)* interfacing to legacy or closed-source software that uses custom binary formats for which no parser is accessible; *c)* application specific constraints on the binary format. The most painful drawback of an ad-hoc binary format is its mainteinance. Any change in the layout means changing the marshalling/unmarshalling routines, which is error prone due to the fact that bitwise logical and shifting operations are involved and off-by-one errors might fester. It is therefore appealing to have such routines automatically derived from an high level layout specification. Consider the Portable Network Graphic (PNG) format [13]. PNG image files are composed of a fixed 8 bytes header, followed by an arbitrary number of chunks[1]. A chunk itself has a variable length; Tab. 1 shows its layout. The

Table 1: The PNG chunk layout.

Meaning	Length	Type	Data	CRC
Bytes	4	4	Variable	4

Length field's value tells the length of the Data field. Without knowing the value of the Length field it is impossible to unambiguously parse the rest of the chunk (and any following chunks). This means that not only the presence, but also the position in the stream of the Length field is crucial for deserialization. Placing Length *after* Data would prevent deserialization since it is not possible to know at what point of the stream Length would begin. The example shows how variable fields urge the presence of meta-information in the stream. In practice these are pointer fields or terminator sequences of bits (or *syncwords*). These two solutions are not equivalent. In the PNG case it is desirable to know in advance how much memory to allocate, since the image data is buffered to be consumed e.g. for displaying on a

[1] Terminology taken from [13].

Submitted to:
HCVS 2016: 3rd Workshop on Horn Clauses for Verification and Synthesis

screen. In contrast, syncwords are preferred when the length of the variable field is not known when the data is sent. Consider the playback of audio-streams like MP3 [20]. Header packets give the information about the the bitrate of the following data; this fixes the amount of buffering needed, since the data is discarded as soon as it is played back. The arrival of a new packet header is signalled by a syncword.
Contributions. In this paper I show *a method to formally check if a layout is successfully deserializable or not*, by defining a stream model and a *parser* model, that is, a first-order logic axiomatization that encodes in horn clauses the behaviour of a parser that reads and interprets a sequential stream with respect to a layout. I use known reasoning techniques to infer the layout properties.
Paper structure. In **Sect. 2** I summarize the needed background about reasoning on knowledge bases. In **Sect. 3** I elaborate a simplified model of the layout, and introduce the parser model and the deserialization check. In **Sect. 4** I make the model more expressive and analyze the consequences on deserialization. In **Sect. 5** I discuss related work. In **Sect. 6** I conclude the paper and illustrate future work.
Disambiguation. In the following, the intended meaning of the word model is "mathematical description of a process" and *not* "interpretation that makes true a theory in first-order logic".

2 Background: Knowledge Representation and First-Order Logic

Let KB be a finite conjunction of first-order formulae of the form $\alpha \Rightarrow \beta$, called *rules*. A rule of the form $true \Rightarrow \beta$ is called a *fact*. α is a conjunction of (possibly negated) predicates; β is a predicate[2]. KB is called *knowledge base* and represents the known causal relations and facts about a modelled domain. It is possible to infer new facts using a *forward chaining* algorithm [22], that is, repeatedly applying modus ponens to the rules and facts present in KB until no new facts are inferred. A rule with no negated premises is called Horn rule (or clause); a knowledge base made of Horn rules is a Horn knowledge base. Such class of knowledge bases is relevant because inference can be efficient [15]. Using forward chaining on a KB of a first-order language without functions is guaranteed to always terminate, because the number of facts that can be generated is finite; without functions no other references to domain elements than the ones explicitly mentioned in the knowledge base can be created. Forward chaining might not terminate if the language has functions, as it might endlessly generate new facts; e.g., applying forward chaining to the Peano axioms. When used in rule-based languages and systems [17, 19] such as CLIPS or JESS, forward chaining models the reasoning process of an agent, where the knowledge base represents what the agent knows at a particular point of the reasoning. In this case, forward chaining uses negation as failure [14] besides modus ponens, which practically means that the lack of a fact in KB implies its falsity. For instance, consider $KB' = \{\neg A(1) \Rightarrow B(1), B(x) \Rightarrow A(x)\}$. An expert system like CLIPS would infer $KB'' = KB' \cup \{B(1), A(1)\}$; modus ponens alone would not apply.

3 Deserialization of Binary Layouts

A layout is the sequence of fields, left to right, expected in reading a stream. I first give a formal model to describe layouts with sufficient detail. I then formulate a first-order formal system $\mathfrak{D} = \langle \mathfrak{R}, \vdash, \mathcal{A} \rangle$ where $\mathfrak{R}$ is a first-order language, $\vdash$ is the modus ponens inference rule, $\mathcal{A}$ a set of axioms describing the parser's knowledge. I will write $\mathcal{A} \vdash^* \alpha$, where α is a formula in $\mathfrak{R}$, to mean that a proof exists for α in $\mathfrak{D}$. Let the following be: $\mathcal{B} = \{(\!|o \rhd s|\!) \,|\, o, s \in \mathbb{N}\}$; $\mathcal{I} = \mathcal{B} \cup \{\mathbf{f}, \mathbf{v}\}$; $\Phi = \mathcal{I} \times \mathbb{N}$. I represent a pair

[2] Propositions are considered here as nullary predicates.

$\langle \imath, i \rangle \in \Phi$ as $\imath_i$ instead of the usual tuple format. Let A be any set. Let $\centerdot : A^n \times A^m \longmapsto A^{n+m}$ be a family of associative concatenation operations, which concatenate together two tuples: e.g. $\mathbf{f}_1\mathbf{f}_2 \centerdot \mathbf{v}_5\mathbf{f}_2 = \mathbf{f}_1\mathbf{f}_2\mathbf{v}_5\mathbf{f}_2$. Define the family of size functions $|x| : A^k \longmapsto \mathbb{N}$ as $|x| = k$, which tells the number of elements in the tuple: e.g., $|\mathbf{f}_1\mathbf{f}_2\mathbf{v}_5\mathbf{f}_2| = 4$. The set of tuples of A of any size is denoted by $A^* = \bigcup_{k\in\mathbb{N}} A^k$. Given an $\vec{a} \in A^*$ and $\alpha \in A$ I will write $\alpha \hookrightarrow \vec{a}$ meaning that a occurs in $\vec{a}$ at any position; $\alpha \hookrightarrow_i \vec{a}$ where $\vec{a} \in A^n$ and $i \in \mathbb{N}, 0 \leq i < n$, to mean that α occurs in $\vec{a}$ at position i.

Define the function $\lambda : \mathcal{I}^* \longmapsto \Phi^*$ as $\lambda(\varepsilon) = \varepsilon$, $\lambda(\imath) = \imath_0$, $\lambda(\upsilon \centerdot \imath) = \lambda(\upsilon) \centerdot \imath_k$ where $\imath \in \mathcal{I}, \upsilon \in \mathcal{I}^{k-1}$, and $\varepsilon \in \mathcal{I}^0$ is the identity element of the $\centerdot$ operator. Let $\mathcal{L}' = Image(\lambda)$. Given a $\vec{\ell} \in \mathcal{L}'$ I will write $\imath_\omega \hookrightarrow \vec{\ell}$ meaning that $\imath_\omega$ occurs in $\vec{\ell}$; $\imath \hookrightarrow \vec{\ell}$ meaning that there is an $\imath$ occurring in $\vec{\ell}$ with any label. The set of layouts $\mathcal{L} \subset \mathcal{L}'$ is defined as follows: $\mathcal{L} = \{\vec{\ell} \in \mathcal{L}' | \forall \imath . \left(\imath \hookrightarrow \vec{\ell}\right) \wedge \left(\imath = (\!|o \rhd s|\!)_k\right) \Rightarrow \left(o < |\vec{\ell}|\right) \wedge \left(s \leq |\vec{\ell}| - o\right)\}$. The above cryptic formal introduction is to set a framework for describing, later in the paper, extensions to the layout model: the function $\lambda(x)$ assigns unique labels that identify the items in a tuple x with their position in x. I will sometimes drop the labeling subscripts for readability.

Each layout field has a length, that is, the number of contiguous bits that will represent the content of the field in the stream. The concrete value of the length is not important in this work. The meaning of each $\imath \in \mathcal{I}$ is defined as follows: *1)* $\mathbf{f}$ is a fixed length field; *2)* $\mathbf{v}$ is a variable length field, or *varfield*; *3)* any $(\!|o \rhd s|\!) \in \mathcal{B}$ indicates a fixed length pointer field, where o is the offset label of the pointer, *offset* in short, and $\omega = o + s$ is the label of the item the pointer is pointing to — thus I call s the *span* (*and not length!*) of a pointer; I define a function $range : \mathcal{B} \longmapsto \mathbb{N}$ as $range((\!|o \rhd s|\!)) = \{j \in \mathbb{N} | o \leq j < o + s\}$ which tells the *range* of a pointer. Note that the definition of $\mathcal{L}$ rules out pointers pointing or spanning beyond $|\vec{\ell}|$. A layout $\vec{\ell} \in \mathcal{L}$ defines the structure of a set of concrete bitstrings, denoted by $\mathcal{S}(\vec{\ell})$.
Example 3.1: *The representation of the* PNG *chunk of Sect. 1 is* $(\!|2 \rhd 1|\!)_0\mathbf{f}\ \mathbf{v}_2\mathbf{f}$*; and* $range((\!|2 \rhd 1|\!)) = \{2\}$.

3.1 Parser Model: Axioms and Knowledge Representation

A parser reads a stream sequentially and interprets the fields according to their layout $\vec{\ell} \in \mathcal{L}$. I assume that it is not possible to know whether the stream is over or not, e.g. with an *end-of-stream* signal, event, or symbol. Consider a first-order language $\mathfrak{R}_n = \langle \mathcal{C}, \mathcal{V}, \mathcal{F}, \mathcal{P} \rangle$ with a set of constants $\mathcal{C} = \{c_0, \ldots, c_n\}$, an infinite supply of variables $\mathcal{V}$, a single binary function $+ \in \mathcal{F}$, the unary predicate set $\mathcal{P}_u = \{Beg(), Len(), Val()\}$, and the ternary predicate set $\mathcal{P}_t = \{Ptr()\}$. Let $\mathcal{P} = \mathcal{P}_u \cup \mathcal{P}_t$. I define the parser model as a theory $\mathcal{A}$ in $\mathfrak{R}_n$, in the following way[3]. Let the domain be $\mathbb{N}$. I impose that the interpretation $[\![c]\!] : \mathcal{C} \longmapsto \mathbb{N}$ of any $c \in \mathcal{C}$ is fixed: $[\![c_0]\!] = 0, \ldots, [\![c_n]\!] = n$. To force the interpretation of the $+$ function I add to $\mathcal{A}$ the axioms[4] defining the addition over $\mathbb{N}$. All $p \in \mathcal{P}$ have a corresponding predicate, $[\![p]\!]$, with integer arguments. My intention is to give the following meanings to the predicates. Let $\vec{\ell} \in \mathcal{L}, i \in \mathcal{V}, \imath \in \mathcal{I}$; let $\imath_{[\![i]\!]} \hookrightarrow \vec{\ell}$. Then *a)* $Beg(i)$ means "the parser knows where $\imath_{[\![i]\!]}$ begins in the stream"; *b)* $Len(i)$ means "the parser knows $\imath_{[\![i]\!]}$'s length"; *c)* $Val(i)$ means "the parser knows $\imath_{[\![i]\!]}$'s content"; *d)* $Ptr(o, s, i)$ tells that there is a pointer field with label $[\![i]\!]$ that contains a measure of how many bits there are between the beginning of the fields labeled with $[\![o]\!]$ and $[\![o + s]\!]$. Less verbosely, it means $\imath_{[\![i]\!]} = (\!|[\![o]\!] \rhd [\![s]\!]|\!)_{[\![i]\!]}$. I define the behaviour of a parser with the following axioms $\mathcal{A}_{\vec{\ell}}$ (implicitly universally quantified):

1. *The parser knows where* $\imath_{[\![0]\!]}$ *begins.*

$$true \Rightarrow Beg(0) \qquad \texttt{(begin)}$$

[3] It is understood that the theory is the conjunction of the formulae it contains.
[4] Not reported here.

2. *If a parser knows where $\iota_{[\![i]\!]}$ begins and its length, then it knows its value and where $\iota_{[\![i+1]\!]}$ begins.*

$$Beg(i) \wedge Len(i) \Rightarrow Val(i) \wedge Beg(i+1) \qquad (\texttt{forward}_i)$$

3. *If a parser knows where $\iota_{[\![i+1]\!]}$ begins and the length if its predecessor $\iota_{[\![i]\!]}$, then it knows where $\iota_{[\![i]\!]}$ begins and its value.*

$$Beg(i+1) \wedge Len(i) \Rightarrow Beg(i) \wedge Val(i) \qquad (\texttt{backward}_i)$$

4. *If a parser knows where $\iota_{[\![i]\!]}$ and its successor $\iota_{[\![i+1]\!]}$ begin, then it knows $\iota_{[\![i]\!]}$'s length.*

$$Beg(i) \wedge Beg(i+1) \Rightarrow Len(i) \qquad (\texttt{join}_i)$$

5. *If $\iota_{[\![i]\!]} = (\!|[\![o]\!] \rhd [\![b]\!]|\!)$ and the parser knows* **a)** *the value of $\iota_{[\![i]\!]}$* **b)** *where $\iota_{[\![o]\!]}$ begins, then it knows where $\iota_{[\![o+b]\!]}$ begins.*

$$Ptr(o,s,i) \wedge Val(i) \wedge Beg(o) \Rightarrow Beg(o+s) \qquad (\texttt{jumpRight}_{o,s,i})$$

6. *If $\iota_{[\![i]\!]} = (\!|[\![o]\!] \rhd [\![b]\!]|\!)$ and the parser knows* **a)** *the value of $\iota_{[\![i]\!]}$* **b)** *where $\iota_{[\![o+b]\!]}$ begins, then it knows where $\iota_{[\![o]\!]}$ begins.*

$$Ptr(o,s,i) \wedge Val(i) \wedge Beg(o+s) \Rightarrow Beg(o) \qquad (\texttt{jumpLeft}_{o,s,i})$$

The above axioms are common to all layouts; the following are axioms that are added according to the specific shape of the layout $\vec{\ell}$ under analysis[5]. For each $\iota_{[\![i]\!]} \hookrightarrow \vec{\ell}$: if $\iota = \mathbf{f}$ or $\iota = (\!|[\![o]\!] \rhd [\![s]\!]|\!)$, then the parser knows

$$true \Rightarrow Len(i) \qquad (\texttt{field}_i)$$

and additionally if $\iota = (\!|[\![o]\!] \rhd [\![s]\!]|\!)$ the parser knows

$$true \Rightarrow Ptr(o,s,i) \qquad (\texttt{ptr}_i)$$

I will drop the subscript to $\mathcal{A}_{\vec{\ell}}$ whenever the $\vec{\ell}$ it refers to is clear from the context. I will call $\mathcal{A}$ the *initial knowledge base*. In the following I will abuse the notation by using the same digit symbols to represent both *a)* the value represented *b)* the syntactic entity representing it, therefore not explicitly representing the interpretation function $[\![\cdot]\!]$ when such distinction is not necessary. Wrapping up, for each $\vec{\ell} \in \mathcal{L}$ there is a formal system $\mathfrak{D}_{\vec{\ell}} = \langle \mathfrak{R}_{|\vec{\ell}|}, \vdash, \mathcal{A} \rangle$ which is $\vec{\ell}$'s parser model; and in the following, whenever i write about a parser, I implicitly refer to such a structure.

3.2 Ambiguity

The presence of a varfield creates ambiguity. For instance the layout **fv** is ambiguous, because there is no way for a parser to know $\mathbf{v}_1$'s length; likewise in **fvf** there is no way, in a concrete stream, to tell $\mathbf{v}_1$ from $\mathbf{f}_2$. A layout $\vec{\ell}$ is unambiguous, or *deserializable*, if and only if a parser can infer the lengths of all $\mathbf{v}_i \hookrightarrow \vec{\ell}$. Pointers are *bounding* items, in that their presence can bound varfields and therefore disambiguate the layout.

[5]For completeness one can extend the layout model with a *constant field* **c** to signal the end of a variable field with a constant pattern. Thus, for each $\mathbf{c}_i \hookrightarrow \vec{\ell}$, add an axiom $Beg(i)$. This is sound under the assumption that the bits in the stream before $\mathbf{c}_i$ are such that the interpretation is not ambiguous, e.g. the pattern in $\mathbf{c}_i$ occurs in the bits of $\mathbf{v}_{i-1}$.

Example 3.2: *Consider layout* $(\!|1 \rhd 2|\!)\mathbf{v}_1(\!|1 \rhd 1|\!)(\!|2 \rhd 1|\!)_3$. *Item* $\mathbf{v}_1$ *is bounded by* $(\!|1 \rhd 2|\!)_0$ *and* $(\!|1 \rhd 1|\!)_2$*; it is not bounded by* $(\!|2 \rhd 1|\!)_3$.

Theorem 3.1. ***Necessary condition for deserializability.*** *If* $\vec{\ell} \in \mathcal{L}$ *is deserializable then for all* $\mathbf{v}_j \hookrightarrow \vec{\ell}$ *there exists a pointer* $x = (\!|b \rhd s|\!)_p \hookrightarrow \vec{\ell}$ *such that* $j \in range(x)$.

Proof. Let $\mathbf{v}_j \hookrightarrow \vec{\ell}$. Since $\vec{\ell}$ is deserializable, it is true that $\mathcal{A} \vdash^* Len(k), 0 \leq k \leq |\vec{\ell}|$. I show that any proof of $Len(j)$ contains the application of $\texttt{jumpRight}_{o,s,i}$ or $\texttt{jumpLeft}_{o,s,i}$, with $o \leq j < o+s$, by applying the inference steps backwards. Observe that *1)* $Len(j) \notin \mathcal{A}$, otherwise $\mathbf{v}_j$ would not be a varfield. *2)* The $\texttt{join}_j$ axiom is the only axiom that allows to infer $Len(j)$, so any proof necessarily applies $\texttt{join}_j$. *3)* $\mathbf{v}_0 \not\hookrightarrow \vec{\ell}$, otherwise $\vec{\ell}$ would not be deserializable. Assume that there are no proofs involving $\texttt{jumpRight}_{o,s,i}$ or $\texttt{jumpLeft}_{o,s,i}$. Further, observe the premises of $\texttt{join}_j$: $Beg(j)$ cannot be inferred through $\texttt{backward}_j$, since $Len(j)$ is in its premises leading to circularity; likewise $Beg(j+1)$ cannot be inferred through $\texttt{forward}_j$. Thus, *1)* $Beg(j)$ must then be inferred through $\texttt{forward}_j$, which means that a proof is a chain of $\texttt{forward}_k$, with $0 \leq k \leq j$; *2)* consequently $Beg(j+1)$ is inferred through $\texttt{backward}_{j+1}$. Since the layout is finite, at most $|\vec{\ell}| - j$ $\texttt{backward}_k$ inferences can be done. Note that any inference of a new $Beg(k)$ depends on some other $Beg(l)$; the only axiom of such shape is $Beg(0)$; so applying only $\texttt{backward}_k$ will not close the proof, which contradicts the hypothesis that $\mathcal{A} \vdash^* Len(k)$. Hence, the proof of at least one of $Beg(t)$ with $t > j$ must include an application of $\texttt{jumpRight}_{l,s,p}$ where $l + s = t$, because it allows to infer $Beg(t)$ from $Beg(l)$ where $l < j < t$; and as seen at point 1, leads backwards to the axiom $Beg(0)$. This requires that $(\!|l \rhd s|\!)_p \hookrightarrow \vec{\ell}$. □

Note that this condition is not sufficient: layout $(\!|0 \rhd 4|\!)\mathbf{v}_1(\!|3 \rhd 1|\!)\mathbf{v}_3$ satisfies the necessary condition, but it is not possible to know the length of $\mathbf{v}_1$ nor $\mathbf{v}_3$. The interesting fact about Theorem 4.1 is that there is no constraint on the value p in the layout. This means that pointer and varfield can be in any relative order.

Example 3.3: *Consider* $\vec{\ell} = \mathbf{f}\ (\!|2 \rhd 3|\!)\ \mathbf{f}\ \mathbf{v}_3\mathbf{v}_4(\!|3 \rhd 1|\!)$ *and Fig. 1. A parser can read until* $\mathbf{f}_2$ *by applying* $\texttt{forward}_0$*, storing the value of the pointer field at 1; from that point on it can buffer the stream (*$\texttt{jumpRight}_{2,3,1}$*) until* $(\!|3 \rhd 1|\!)_5$*, which once read with* $\texttt{forward}_5$ *allows to determine the lengths of* $\mathbf{v}_3$ *and* $\mathbf{v}_4$ *through* $\texttt{jumpLeft}_{3,1,5}$ *and* $\texttt{join}_i$*, where* $i \in \{2,3\}$ *followed by, respectively,* $\texttt{forward}_3$ *and* $\texttt{backward}_3$.

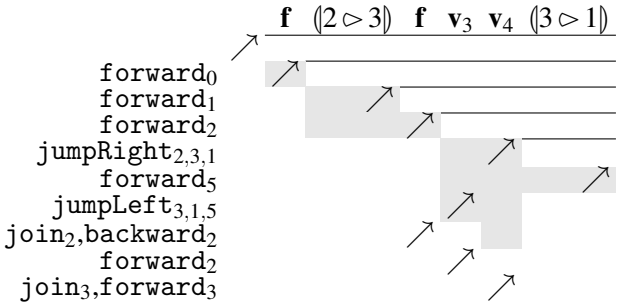

Figure 1: Parsing a stream, Ex. 3.3. Read top to bottom. Each row is a snapshot of the state of the parsing before the application of the axiom on the left. The arrow indicates the position of the parser in the stream; the greyed areas represent pictorially the amount of buffering. The thin line represents the amount of stream consumed. The deserialization is successful when all the stream is consumed, and no buffering is left.

3.3 Deserializability Check Algorithm

Alg. 1 shows in pseudocode how to check for deserializability. It is correct and complete because *forwardChainingInference* is [22][6].

Observation 3.1: Alg. 1 terminates. Let $\vec{\ell} \in \mathcal{L}$ be Alg. 1's input. The $\texttt{forward}_i$ and $\texttt{jumpRight}_{o,s,i}$ are the only axioms that can introduce more complex terms using the $+$ function. I show that such axioms are applied at most $|\vec{\ell}|$ times to produce new facts. $\texttt{jumpRight}_{o,s,i}$ cannot introduce more facts than the number of pointer fields $\pi < |\vec{\ell}|$ since no inference can introduce new $Ptr(o,s,i)$; $\texttt{forward}_i$ introduces a new $Beg(i+1)$ if there is a $Len(i)$ fact in the knowledge base; there are two cases. *1)* $Len(i)$ was already present in the initial knowledge base, so $i < |\vec{\ell}|$, thus $i+1 \leq |\vec{\ell}|$. *2)* $Len(i)$ could have been inferred through $\texttt{join}_i$, but this breaks the assumption that $Beg(i+1)$ is not in the knowledge base. The conditional checks the results of the forward chaining algorithm by comparing two finite structures. This proves termination.

The algorithm is $O(|\vec{\ell}|)$ because propositionalizing the axioms takes linear time due to the shape of the axioms and the absence of uninterpreted function symbols; then, inference is linear on propositional Horn knowledge bases [15, 22]. Note that the deserializability check is not sufficient to decide properties of $\mathcal{S}(\vec{\ell})$, e.g. whether a layout $\vec{\ell}$ has $\mathcal{S}(\vec{\ell}) = \emptyset$. Consider the following scenario: $\vec{\ell} = (\!|0 \rhd 3|\!)\mathbf{f}_1\mathbf{v}_2\mathbf{f}_3$, and suppose the lengths of the non-variable length fields are, respectively, 1 bit, 3 bits and 3 bits. If the meaning of the value of the pointer is to measure the number of bits of the items with labels in $range((\!|0 \rhd 3|\!))$, this value cannot encode that number with just one bit. This additional check is not needed to decide deserializability, and can be performed after the deserializability check by analysing, considering field lengths and encodings, the spans of all pointers.

4 The Repetition Field

A reasonable extension of the model is to have variable occurrences of portions of layout, like the Kleene star in regular expressions: $\vec{\ell}' = (\!|1 \rhd 1|\!)_0[(\!|1 \centerdot 0 \rhd 1|\!)_{1.0}\mathbf{v}_{1.1}]*_1$ to indicate the infinite set of layouts beginning with a pointer and a sequence of alternating pointer and variable fields. The layout is now a tree structure with a new basic item $[]*$ called *repetition*, and identifiers are tuples of integers in $\mathbb{N}^*$. Let $\mathcal{B}' = \{(\!|o \rhd s|\!) | o \in \mathbb{N}^*, s \in \mathbb{N}\}$. Let the set $\mathcal{R}$ be defined recursively as follows: $\varepsilon \in \mathcal{R}$, $\iota \in \mathcal{R}$ where $\iota \in \mathcal{I}'$, $[W]* \in \mathcal{R}$ where $W \in \mathcal{R}^*$. Nothing else is in $\mathcal{R}$. The empty layout ε is defined as the identity operator for $\centerdot$, and $\mathcal{I}' = \mathcal{I} \cup \mathcal{B}'$. To identify each item I define, following the same pattern of Sect. 3, the set $P = \mathcal{R} \times \mathbb{N}^*$

[6] *forwardChainingInference* can be replaced with any existing implementation of forward chaining.

Algorithm 1: Deserializability check

Data: $\vec{\ell} \in \mathcal{L}, V = \{i | \mathbf{v}_i \hookrightarrow \vec{\ell}\}$
Result: A modified knowledge base $\mathcal{A}'$ and the inference graph G
Build $\mathcal{A}$ according to subsection 3.1;
$\langle \mathcal{A}', G \rangle \leftarrow$ *forwardChainingInference*$(\mathcal{A})$;
if $\exists . i \in V | Len(i) \notin \mathcal{A}'$ **then**
  return $\langle NonDeserializable, \mathcal{A}', G \rangle$;
else
  return $\langle Deserializable, \mathcal{A}', G \rangle$

(cf. Φ) and the function $\boldsymbol{\mu} : \mathcal{R}^* \longmapsto P^*$ (cf. λ) defined as follows:

$$
\begin{aligned}
\boldsymbol{\mu}(\varepsilon) &= \varepsilon \\
\boldsymbol{\mu}(\iota) &= \boldsymbol{\mu}_0(\iota) \\
\boldsymbol{\mu}_l(\varepsilon) &= \varepsilon \\
\boldsymbol{\mu}_l(\iota) &= \iota_l \qquad \text{where } \iota \neq [\upsilon]* \\
\boldsymbol{\mu}_l([\iota]*) &= [\boldsymbol{\mu}_0^l(\iota)]*_l \\
\boldsymbol{\mu}_l(\upsilon \centerdot \iota) &= \boldsymbol{\mu}_l(\upsilon) \centerdot \boldsymbol{\mu}_{l+n}(\iota)
\end{aligned}
\qquad\qquad
\begin{aligned}
\boldsymbol{\mu}_l^k(\varepsilon) &= \varepsilon \\
\boldsymbol{\mu}_l^k(\iota) &= \iota_{k.l} \\
\boldsymbol{\mu}_l^k(\iota) &= [\boldsymbol{\mu}_0^{k.l}(\iota)]*_{k.l} \\
\boldsymbol{\mu}_l^k(\upsilon \centerdot \iota) &= \boldsymbol{\mu}_l^k(\upsilon) \centerdot \boldsymbol{\mu}_{l+n}^k(\iota)
\end{aligned}
$$

In words: the function $\boldsymbol{\mu}_l$ labels the items left to right, starting from $l \in \mathbb{N}$ and introducing a context l when it applied to a repetition; $\boldsymbol{\mu}_l^k$ labels the items starting from $l \in \mathbb{N}$, in the context $k \in \mathbb{N}^*$.

Let $\mathcal{M}' = \mathit{Image}(\boldsymbol{\mu})$. The set of layouts $\mathcal{M} \subset \mathcal{M}'$ is defined as follows:

$$
\begin{aligned}
\mathcal{M} = \{ \vec{\ell} \in \mathcal{M}' \mid \\
&\forall \iota . \left(\iota = (\!|a \rhd b|\!)_k \hookrightarrow \vec{\ell} \right) \Rightarrow \left(\left(\iota = (\!|s \centerdot c \rhd b|\!)_{s.d} \vee \iota = (\!|s \rhd b|\!)_{s.e} \right) \right), && (1) \\
&\forall y . \left([y]*_m \hookrightarrow \vec{\ell} \right) \Rightarrow \left(\left((\!|m \centerdot f \rhd g|\!) \hookrightarrow \vec{\ell} \right) \Rightarrow g \leq |l| - f \right), && (2) \\
&\forall \iota . \left(\iota = (\!|f' \rhd g'|\!)_{c'} \hookrightarrow \vec{\ell} \right) \Rightarrow \left(g' \leq |\vec{\ell}| - c' \right), && (3) \\
&l \in \mathcal{R}^*, y = \boldsymbol{\mu}_0^m(l), \{b,c,c',d,e,f,f',g,g'\} \subset \mathbb{N}, \{a,k,m\} \subset \mathbb{N}^* \setminus \mathbb{N}^0, s \in \mathbb{N}^* \\
\}.
\end{aligned}
$$

In words, in all $\vec{\ell} \in \mathcal{M}$: (1) tells that the offset of any pointer refers to a label of a parent scope, or to an element at the same level. This is needed to prevent ambiguous references. For instance in $\vec{\ell} = (\!|1 \centerdot 0 \rhd 1|\!)_0 [\mathbf{f}_{1.0}]*_1 \notin \mathcal{M}$ the pointer $(\!|1 \centerdot 0 \rhd 1|\!)_0$ points to $\mathbf{f}_{1.0}$ which in a concrete stream can appear an unbounded number of times and therefore the pointer would be ambiguous. (2) and (3) tell that all pointers have spans that do not exceed the number of fields of the context they are in. The list labels give the context needed to state this property. The mapping $\nabla : \mathcal{M} \longmapsto 2^{\mathcal{L}}$ maps, informally[7], to a set of $\vec{\ell} \in \mathcal{M}$ without repetitions corresponding to all the combinations of unwindings of the repetitions, $0, 1, 2 \ldots$ times. For the above example: $\nabla(\vec{\ell}) = \{(\!|0 \rhd 0|\!), (\!|0 \rhd 2|\!)(\!|1 \rhd 1|\!)\mathbf{v}, (\!|0 \rhd 2|\!)(\!|1 \rhd 1|\!)\mathbf{v}(\!|3 \rhd 1|\!)\mathbf{v} \ldots\}$.

4.1 Parser Model

As in Sect. 3, I will define a formal system $\mathfrak{D}_{\vec{\ell}} = \langle \mathfrak{S}_{|\vec{\ell}|}, \vdash, \mathcal{A} \rangle$, where $\vec{\ell} \in \mathcal{M}$, to analyze the deserializability of $\vec{\ell}$. The system's first-order language is $\mathfrak{S}_n = \langle \mathcal{C}, \mathcal{V}, \mathcal{F}', \mathcal{P}' \rangle$ where $\mathcal{F}' = \mathcal{F} \cup \{\centerdot\}$ and $\mathcal{P}' = \mathcal{P} \cup \{\mathit{Rep}(), \mathit{RepLen}()\}$ where $\mathit{Rep}()$ is a binary predicate and $\mathit{RepLen}()$ is a unary predicate. Let the domain be $\mathbb{N}^*$; predicate symbols in $\mathcal{P}'$ map to predicates of the same arities and names. The interpretation of constant symbols is fixed as explained in subsection 3.1, *mutatis mutandis*. The function symbol $+$ is interpreted as addition over integers; it is left undefined for arguments $a \notin \mathbb{N}$. The symbol $\centerdot$ corresponds to the tuple concatenation function introduced in Sect. 3[8]. The axioms of subsection 3.1 are

[7] A formal definition is omitted. The mapping must take care of *a)* flattening the label structure *b)* change the pointer elements' spans and offsets. I rely on the intuitive meaning of ∇ to avoid a complicated formal definition.

[8] Axioms defining the behaviour of integers, lists of integers and the relevant operations are not reported here.

lifted to the list domain:

$$\begin{aligned}
\mathit{true} &\Rightarrow \mathit{Beg}(0) && (\texttt{begin})\\
\mathit{Beg}(b\texttt{.}a) \wedge \mathit{Len}(b\texttt{.}a) &\Rightarrow \mathit{Val}(b\texttt{.}a) \wedge \mathit{Beg}(b\texttt{.}a+1) && (\texttt{forward}_{b\texttt{.}a})\\
\mathit{Beg}(b\texttt{.}a+1) \wedge \mathit{Len}(b\texttt{.}a) &\Rightarrow \mathit{Beg}(b\texttt{.}a) \wedge \mathit{Val}(b\texttt{.}a) && (\texttt{backward}_{b\texttt{.}a})\\
\mathit{Beg}(b\texttt{.}a) \wedge \mathit{Beg}(b\texttt{.}a+1) &\Rightarrow \mathit{Len}(b\texttt{.}a) && (\texttt{join}_{b\texttt{.}a})\\
\mathit{Ptr}(b\texttt{.}a,s,i) \wedge \mathit{Val}(i) \wedge \mathit{Beg}(b\texttt{.}a) &\Rightarrow \mathit{Beg}(b\texttt{.}a+s) && (\texttt{jumpRight}_{b\texttt{.}a,s,i})\\
\mathit{Ptr}(b\texttt{.}a,s,i) \wedge \mathit{Val}(i) \wedge \mathit{Beg}(b\texttt{.}a+s) &\Rightarrow \mathit{Beg}(b\texttt{.}a) && (\texttt{jumpLeft}_{b\texttt{.}a,s,i})
\end{aligned}$$

where $a,s \in \mathbb{N}, b,i \in \mathbb{N}^*$ and $+$ has higher precedence than $\texttt{.}$. The intended meaning of $\mathit{Rep}(a,l)$ is "there is a repetition at position a which contains l fields". Note that a repetition is a field, consistently with how repetitions are labeled. $\mathit{RepLen}(a)$ means "the parser knows the length of the repetition at position a".

$$\begin{aligned}
\mathit{Rep}(b\texttt{.}a,l) \wedge \mathit{Beg}(b\texttt{.}a) \wedge \mathit{Beg}(b\texttt{.}a+1) &\Rightarrow \mathit{RepLen}(b\texttt{.}a) && (\texttt{replen}_{b\texttt{.}a})\\
\mathit{Rep}(b\texttt{.}a,l) \wedge \mathit{Beg}(b\texttt{.}a) &\Rightarrow \mathit{Beg}(b\texttt{.}a\texttt{.}0) && (\texttt{rephead}_{b\texttt{.}a})\\
\mathit{Rep}(b\texttt{.}a,l) \wedge \mathit{Beg}(b\texttt{.}a+1) &\Rightarrow \mathit{Beg}(b\texttt{.}a\texttt{.}l) && (\texttt{reptail}_{b\texttt{.}a})
\end{aligned}$$

where $a,l \in \mathbb{N}$ and $b \in \mathbb{N}^*$. Axiom $\texttt{replen}_{b\texttt{.}a}$ tells how a parser gets to know the length of a repetition; $\texttt{rephead}_{b\texttt{.}a}$ and $\texttt{reptail}_{b\texttt{.}a}$ tell how the parser accesses the fields inside a repetition. For each $[\iota_{b\texttt{.}0} \ldots \iota_{b\texttt{.}l-1}]*_b \hookrightarrow \vec{\ell}$, $\mathcal{A}$ contains the facts

$$\mathit{true} \Rightarrow \mathit{Rep}(b,l) \qquad (\texttt{repeat}_b)$$

As no axioms in $\mathcal{A}$ allow to deduce any $\mathit{Rep}()$, this is the only way they can be included in the knowledge base. This prevents by construction to have $l \notin \mathbb{N}$, without the need of typing $\mathfrak{S}_n$ or defining $+$ for all $i,j \in \mathbb{N}^*$. Additional axioms $\texttt{field}_i$ and $\texttt{ptr}_i$ are lifted to the list domain and added to the knowledge base under the same circumstances described for their counterparts in subsection 3.1.

Caveat! Consider $\vec{\ell} = (\!|1 \rhd 1|\!)_0[\mathbf{v}_{1.0}]*_1 \in \mathcal{M}$. Applying Alg. 1 with the modified knowledge base tells that $\vec{\ell}$ is deserializable (Fig. 2(a)). This is unsound: knowing the length of the repetition field does not allow to discriminate the occurrences of $\mathbf{v}$ in a concrete stream. The problem that the example exposes is that the theory confuses in a single identifier $1\texttt{.}0$ *all* the occurrences of the varfield in the repetition. I illustrate how to fix this shortcoming after some preliminary definitions. Let $\vec{\ell}' = \mathit{reverse}(\vec{\ell})$ be the permutation of $\vec{\ell}$ defined as follows:

$$\begin{aligned}
(\iota \notin \mathcal{B}') \wedge (\iota_i \hookrightarrow \vec{\ell}) &\Leftrightarrow \iota_{|\vec{\ell}|-i} \hookrightarrow \vec{\ell}'\\
(\!|a\texttt{.}k \rhd b|\!)_i \hookrightarrow \vec{\ell} &\Leftrightarrow (\!||\vec{\ell}| - (a+b)\texttt{.}k \rhd b|\!)_{|\vec{\ell}|-i} \hookrightarrow \vec{\ell}'
\end{aligned}$$

where $a \in \mathbb{N}, k \in \mathbb{N}^*$. Parsing $\mathit{reverse}(\vec{\ell})$ is equivalent to parsing $\vec{\ell}$ backwards, that is, substituting the axiom $\mathit{Beg}(0)$ with $\mathit{Beg}(|\vec{\ell}|)$.

Example 4.1: *Let* $\vec{\ell} = (\!|1 \rhd 2|\!)_0\mathbf{f}_1[(\!|1\texttt{.}0 \rhd 2|\!)\mathbf{v}]*_2$. *Then* $\mathit{reverse}(\vec{\ell}) = [(\!|1\texttt{.}0 \rhd 2|\!)\mathbf{v}]*_0\mathbf{f}_1(\!|0 \rhd 2|\!)_2$.

Note that $\mathit{reverse}(\vec{\ell}) \in \mathcal{M}$. Furthermore, let $\vec{r} \in \mathcal{R}^*$. Then $\vec{r}^n$ is the structure such that

$$\begin{aligned}
(\iota \notin \mathcal{B}') \wedge (\iota \hookrightarrow_i \vec{r}) &\Leftrightarrow \iota \hookrightarrow_{|\vec{r}|-i} \vec{r}^n\\
(\!|a\texttt{.}k \rhd b|\!) \hookrightarrow_i \vec{r} &\Leftrightarrow (\!|a+n\texttt{.}k \rhd b|\!) \hookrightarrow_i \vec{r}^n
\end{aligned}$$

where $a,n \in \mathbb{N}, k \in \mathbb{N}^*$. The sequence $\vec{r}^n$ is the same as $\vec{r}$, where the head of all offset labels is increased by n.

Example 4.2: *Let $\vec{r} \in \mathcal{R}^*$. Then*

$$\vec{r} = (\!|0 \rhd 2|\!)[(\!|1\,.\,0 \rhd 2|\!)\mathbf{v}[(\!|1\,.\,2\,.\,0 \rhd 2|\!)\mathbf{f}]*]*$$
$$\vec{r}^3 = (\!|3 \rhd 2|\!)[(\!|4\,.\,0 \rhd 2|\!)\mathbf{v}[(\!|4\,.\,2\,.\,0 \rhd 2|\!)\mathbf{f}]*]*$$

This transformation takes care of properly translating the pointer offsets when concatenating tuples of fields, as will happen below.

As a last premise, Theorem 4.1 is lifted to include repetitions. Let $range' : \mathcal{B}' \longmapsto \mathbb{N}^*$ be defined as $range'((\!|b\,.\,a \rhd s|\!)) = \{b\,.\,k | a \leq k < a+s\}$ where $a \in \mathbb{N}$.

Theorem 4.1. ***Necessary condition for deserializability with repetitions.*** *If $\vec{\ell} \in \mathcal{M}$ is deserializable then for all $\iota_j \hookrightarrow \vec{\ell}$, where $\iota \in \{[\mathfrak{v}]*, \mathbf{v}\}$, then there exists a pointer $x = (\!|b \rhd s|\!)_p \hookrightarrow \vec{\ell}$ such that $j \in range'(x)$.*

The proof is similar to that of Theorem 4.1 and is therefore omitted. □

Observe that there are $r \in \mathcal{R}^- \subset \mathcal{R}^*$ such that $\vec{\ell} = \boldsymbol{\mu}(r)$ is not deserializable, but if prepended with a bounding pointer they are:

$$\vec{\ell}' = \boldsymbol{\mu}((\!|1 \rhd |r|\,|\!)\,.\,r^1) \tag{ONCE}$$

This is the case of Fig. 2(c). This means that there is an item $\iota_i \hookrightarrow \vec{\ell}'$ such that knowing $Beg(1)$ and $Beg(|r|+1)$ entails $Len(i)$. If one considers

$$\vec{\ell}'' = \boldsymbol{\mu}((\!|1 \rhd 2|r|\,|\!)\,.\,r^1\,.\,r^{|r|+1}) \tag{TWICE}$$

then the following can be proved true $\forall r \in \mathcal{R}^-$, thus when $\vec{\ell}'$ is deserializable and $\vec{\ell}$ is not:

Theorem 4.2. *$\vec{\ell}''$ is deserializable $\Leftrightarrow$ $\vec{\ell}_r = reverse(\boldsymbol{\mu}(r))$ is deserializable.*

Proof. ($\Rightarrow$, SKETCH.) Suppose $\vec{\ell}_r$ is not deserializable. This means that there exists an $\iota_i \hookrightarrow \vec{\ell}_r$ whose length is unknown, which corresponds in $\vec{\ell}''$ to the two items ι_{i+1} and $\iota_{i+|r|+1}$. Observe that no pointers can span from r^1 to $r^{|r|+1}$ by construction, which together with **??** means that there is no chance that the deserializability of $\vec{\ell}''$ comes from concatenating r^1 and $r^{|r|+1}$. Then the knowledge of $Len(i+1)$ depends on $Beg(1)$ and $Beg(|r|+1)$, and that of $Len(i+|r|+1)$ depends on $Beg(|r|+1)$ and $Beg(2|r|+1)$, because $\vec{\ell}'$ is deserializable. $Beg(1)$ and $Beg(2|r|+1)$ can be reached from $Beg(0)$, respectively applying $\texttt{forward}_0$ and $\texttt{jumpRight}_{0,2|r|,0}$. $Beg(|r|+1)$ can be inferred in two ways: *1)* from $Beg(1)$ through r^1, but this contradicts that $\vec{\ell}$ is not deserializable because if one could infer $Beg(|r|+1)$ from $Beg(1)$ then $\vec{\ell}$ would be deserializable. Contradiction. *2)* from $Beg(2|r|+1)$, backwards through $r^{|r|+1}$, which would then mean that one could infer $Beg(|r|+1)$ from $Beg(2|l|+1)$, which means that $\vec{\ell}_r$ is deserializable. Contradiction.
($\Leftarrow$, SKETCH.) The pointer $(\!|1 \rhd 2|r|\,|\!)_0$ allows to buffer the whole layout until the end of $r^{|r|+1}$. Since $\vec{\ell}_r$ is deserializable, the parser can parse backwards the whole span of $(\!|1 \rhd 2|r|\,|\!)_0$. □

Example 4.3: *Let $r = \mathbf{v}_0(\!|0 \rhd 1|\!)_1$. Observe that $\vec{\ell}_r = reverse(\boldsymbol{\mu}(r)) = (\!|1 \rhd 1|\!)_0\mathbf{v}_1$ is deserializable; the layout $\vec{\ell}'' = \boldsymbol{\mu}((\!|1 \rhd 4|\!)_0\,.\,r^1\,.\,r^3)$ becomes*

$$(\!|1 \rhd 4|\!)_0\mathbf{v}_1(\!|1 \rhd 1|\!)_2\mathbf{v}_3(\!|3 \rhd 1|\!)_4$$

and is parsed by following the first pointer and reading backwards all that was buffered, since it is possible to infer the length of the varfields.

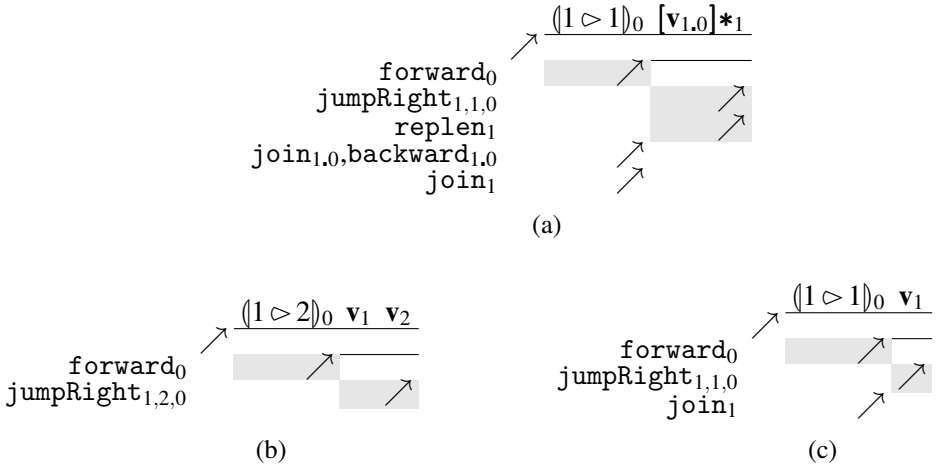

Figure 2: (a): Parsing $\vec{\ell} \in \mathcal{M}$, according to the model. (b): Parsing a concrete instance of $\vec{\ell}' \in \nabla(\vec{\ell})$: in $\vec{\ell}'$ the parser cannot distinguish $\mathbf{v}_1$ and $\mathbf{v}_2$, as $Beg(2)$ is never inferred to allow applying $\texttt{join}_2$ or $\texttt{backward}_2$. This is not sound as $\vec{\ell}$ is deserializable, and so should be all the $\vec{\ell}' \in \nabla(\vec{\ell})$ since $\mathcal{S}(\vec{\ell}) \supset \mathcal{S}(\vec{\ell}')$. (c): Successful parsing of a concrete instance of $\vec{\ell}'' \in \nabla(\vec{\ell})$.

Observation 4.1: $\vec{\ell}''$ is deserializable $\Rightarrow$ $\vec{\ell}'$ is deserializable. By Theorem 4.2 it means $\vec{\ell}_r = reverse(\boldsymbol{\mu}(r))$ is deserializable $\Rightarrow$ $\vec{\ell}'$ is deserializable, which is true because $\texttt{jumpRight}_{1,|r|,0}$ allows to infer $Beg(|r|+1)$ and then read backwards since $\vec{\ell}_r$ is deserializable;

Observation 4.2: If $\vec{\ell}''$ is deserializable, all $\vec{\ell}^n = \boldsymbol{\mu}(\langle\!\langle 1 \rhd n|r| \rangle\!\rangle \centerdot r^1 \centerdot \ldots \centerdot r^{(n-1)|r|+1})$ are, $n > 2$. Therefore it does not matter how many repetitions of r are there, since once $Beg(n|r|+1)$ is known, the stream is reconstructed backwards.

Observation 4.3: Otherwise, no $\vec{\ell}^n$ can be deserializable. This corresponds to those cases where not even the reversed layout is deserializable, such as $\mathbf{v}_0$ or $\langle\!\langle 2 \rhd 1 \rangle\!\rangle \mathbf{v}_1 \mathbf{v}_2$. Since no pointers of any r^i can span beyond r^i, a parser will not be able to proceed either forward to $Beg(|r|+1)$ or backwards from $Beg(n|r|+1)$ to $Beg((n-1)|r|+1)$.

One can therefore transform a layout under analysis $\vec{\ell}_0$ into $\vec{\ell}_1$ by substituting all $[\boldsymbol{\mu}_0^k(r)]*_k \hookrightarrow \vec{\ell}_0$ with $[\boldsymbol{\mu}_0^k(r) \centerdot \boldsymbol{\mu}_{|r|}^k(r^{|r|})]*_k \hookrightarrow \vec{\ell}_1$: that is, duplicating the content of each repetition. Observe that such device creates the same environment described in (TWICE): when the body of a repetition $[\upsilon]*_k$ is entered with $\texttt{replen}_j$, one knows the extremes of a repetition unwinded twice. This is, as sketched, sufficient to determine the deserializability of the repetition. This resumes the soundness of the model. The duplicating transformation is polynomial[9], terminates because of the finiteness of layouts, and preserves deserializability: i.e., if $\vec{\ell}_1$ is deserializable, so is $\vec{\ell}_0$[10]. The advantage of this solution is that it reuses the formal system defined above and does not require side proofs in the formal system $\mathfrak{D}$. Alg. 1 is upgraded to Alg. 2.

[9] A coarse estimate can be $O(nk)$, where n is the number of all items appearing in the layout, and k is the maximum level of nesting of repetitions; observe that $k \leq n$ since repetitions are items too.

[10] And the contrapositive: if $\vec{\ell}_0$ is not deserializable, $\vec{\ell}_1$ is not deserializable.

Algorithm 2: Deserializability check for enhanced parser model.

Data: $\vec{\ell}_0 \in \mathcal{L}, V = \{i|\mathbf{v}_i \hookrightarrow \vec{\ell}\}, R = \{i|[]*_i \hookrightarrow \vec{\ell}\ \}$
Result: A modified knowledge base $\mathcal{A}'$ and the inference graph G
Duplicate the content of each repetition in $\vec{\ell}_0$ into $\vec{\ell}_1$;
Build $\mathcal{A}$ according to subsection 4.1 from $\vec{\ell}_1$;
$\langle \mathcal{A}', G \rangle \leftarrow forwardChainingInference(\mathcal{A})$;
if $\exists.i \in V | Len(i) \notin \mathcal{A}' \vee \exists.i \in R | RepLen(i) \notin \mathcal{A}'$ **then**
  return $\langle NonDeserializable, \mathcal{A}', G \rangle$;
else
  return $\langle Deserializable, \mathcal{A}', G \rangle$

5 Related Work

None of the following works uses explicitly, to my knowledge, any Horn clause representation of the parsing task. The ERLANG language [10] has a pattern-matching construct whose patterns can be binary comprehensions [18], similar to list comprehensions in functional programming languages. Given a set of bit patterns, the matcher is synthesized by constructing a labeled automaton and expressing the matching as a series of elementary actions: test the size of a field, read bits, test match. The specification of a binary format is subject to the variable binding rules of ERLANG; this entails, in practice, that in the case of Ex. 3.3 one must code the layout manually, make an explicit analysis of the layout, and possibly spreading the definition through several functions or mixed with ERLANG statements, reducing the effectiveness of a layout specification as such. PACKET TYPES [21] addresses the processing of protocol packets, hence of bit-strings, through a protocol stack; DATASCRIPT [12] is even more concise, describing the language and its features. Both languages have a syntax that is influenced by the C language. They offer capabilites such as attaching constraints on fields and their content. The constraints can only refer to elements occurring earlier in the stream, thus ruling out instances such as Ex. 3.3. PADS [16] is a framework for analysing and defining bit-level formats; it can generate parsers and serialize data. PADS can even infer, given a set of binary data supposedly following the same layout, the actual layout and be tolerant with errors, by reporting them and continuing parsing. Moreover [16] introduces a general framework to express the semantics of data description languages, focussing on the *types* of fields, where a type represents details such as endianness and encoding of the concrete bitstrings of the field. The framework gives the building blocks to create a type system for the data description language. Type-correctness then entails parsability of a layout. This contrasts with my approach which does not make explicit mention of types of fields, which are not needed for deciding deserializabilty. Beyond the motivations described in Sect. 1, a huge effort in bit-level compilers targets space-efficient exchange formats. Popular ASCII-based data exchange formats have the advantage of being human-readable (JSON) and validable (XML); both do have a wealth of libraries for manipulation with standard interfaces; the disadvantage is that ASCII wastes bandwidth – e.g. encoding a single boolean value in several bytes, instead of a single bit. Programming languages have libraries that allow serialization of their data, like in HASKELL [1, 2] or in C [3], but the definition of the data format is done within the programming language. Data specification languages [4, 5, 6, 7, 8] allow the definition, processing and evolution of protocol messages and output parser/serializers for several target programming language. Such products hide the composition of the underlying stream to the user; unlike what presented here, the definition language does not allow to decide e.g. where to put a pointer item (see Sect. 3), because

the packing algorithms that optimize aspects such as alignment and evolvability rely on a predetermined physical layout.

6 Conclusion and Future Work

I presented a method to determine whether there is a parser that can parse a stream of bits given a description of the bit layout. I introduced a language for describing layouts and I described the behaviour of a parser as reasoning within an untyped first-order logic formal system having axioms in the form of Horn clauses. The typical use case of this method is the implementation of a bit-stream data-definition language, or of a serialization library. The benefit is that it enables to use existing Horn inference engines. At [9] there is a PYTHON [11] implementation of the method using the CLIPS [17] rule-based language to perform forward chaining. It defines a language to describe layouts and translate them to a CLIPS program encoding the axioms, input to the CLIPS interpreter; the PYTHON script interprets back the output. Using PROLOG gives no particular advantages over using other programming languages, since PROLOG interpreters do backward chaining reasoning, thus one can either implement forward chaining or delegate it to any existing library or external tool. The previous sections not discuss any preprocessing of layouts. I report some I observed during the development of this work, which are not closely related with this paper's contribution: *a)* Save bits by reducing the value contained in the pointer fields by substituting all $(\!|o \rhd s|\!) \hookrightarrow \vec{\ell}$ with $(\!|q \rhd t|\!)$ such that each pointer range is shrinked enough to begin and end with a variable length field. This can be done in linear time with a check on the span of every pointer and updating the labels or spans of the pointers left. Once a pointer p is shrinked, one might then redesign manually the layout by reducing the length of p. For instance, consider $\vec{\ell} = (\!|0 \rhd 5|\!)\mathbf{v}_1\mathbf{f}\ \mathbf{v}_3\mathbf{f}$. Applying the above optimization results in $\vec{\ell'} = (\!|1 \rhd 3|\!)\mathbf{v}_1\mathbf{f}\ \mathbf{v}_3\mathbf{f}$. *b)* Allow only forward pointers. Backward pointers are unusual in practice, because they can imply buffering that can be avoided. One could consider only those layouts such that $\forall \imath.\big(\imath \hookrightarrow \vec{\ell}\big) \wedge \big(\imath = (\!|b \centerdot a \rhd r|\!)_{b \centerdot x}\big) \Rightarrow \big(x \leq a\big)$, $a, x \in \mathbb{N}$. This only constraint does not anyway guarantee that all such layouts are deserializable. For instance $\vec{\ell'} = \mathbf{f}\ (\!|2 \rhd 4|\!)\ \mathbf{f}\ \mathbf{v}_3(\!|5 \rhd 1|\!)\mathbf{v}_5$ is not deserializable. *c)* If a pointer's purpose is exclusively to determine the lengths of variable fields, then remove pointers that span over no variable length fields or repetitions. This can be done in linear time with a check on the span of every pointer and updating the labels or spans of the pointers left. More complicated analyses and extensions, which are part of future work, are the following: *i)* Permute the fields so that minimal buffering is needed. Consider $\vec{\ell} = \mathbf{f}\ (\!|2 \rhd 3|\!)\ \mathbf{f}\ \mathbf{v}_3\mathbf{v}_4(\!|4 \rhd 1|\!)$. The layout $\vec{\ell'} = \mathbf{f}\,\mathbf{f}\ (\!|3 \rhd 1|\!)\ \mathbf{v}_3(\!|5 \rhd 1|\!)\mathbf{v}_5$ is a permutation of $\vec{\ell}$; but in $\vec{\ell}$ one must buffer both $\mathbf{v}_3$ and $\mathbf{v}_4$ before being able to distinguish them. *ii)* Have a side-effect free constraint language (like DATASCRIPT or PADS in Sect. 5 do) to express constraints between values and lengths of fields; the constraints contribute in building the axiom set. Consider layout $\vec{\ell} = \mathbf{f}_0\mathbf{f}_1\mathbf{v}_2$. If $\mathbf{v}_2$ is a sequence of samples, $\mathbf{f}_0$ tells the number of samples in $\mathbf{v}_2$ and $\mathbf{f}_1$ tells the number of bits each sample has, then this corresponds to the axiom $\mathit{Val}(0) \wedge \mathit{Val}(1) \Rightarrow \mathit{Len}(2)$.This constraint feature enables for instance to use variable fields as pointers. *iii)* The inference graph can be used to generate a parser for streams $\mathcal{S}(\vec{\ell})$. The axioms applied during the reasoning can be translated into actions, similarly to [18]: $\texttt{jumpRight}_{o,s,i}$ corresponds to buffering new data from the stream, and $\texttt{jumpLeft}_{o,s,i}$ or again $\texttt{jumpRight}_{o,s,i}$ to addressing within the buffer in case of already buffered data. $\texttt{join}_i$, $\texttt{forward}_i$ and $\texttt{backward}_i$ correspond to consuming data and associating it to a field. Note that it is an optimization problem: since the inference graph is a DAG, there are several topological orderings each of which would map to a parser with specific performances in e.g. memory consumption. Describing details of this optimization and related research is future work.

References

[1] CEREAL. https://github.com/GaloisInc/cereal.

[2] BINARY. https://github.com/kolmodin/binary.

[3] TPL. http://troydhanson.github.io/tpl/.

[4] APACHE AVRO™. http://avro.apache.org/docs/1.7.5/spec.html.

[5] PROTOCOL BUFFERS. https://developers.google.com/protocol-buffers/.

[6] BSON. http://bsonspec.org/.

[7] MESSAGE PACK. http://msgpack.org/.

[8] CAP'N'PROTO. https://capnproto.org/.

[9] *GitHub user gapag*. https://github.com/gapag/horn-binary-deserialization.

[10] ERLANG *Programming Language*. http://www.erlang.org/.

[11] *The* PYTHON *programming language*. https://www.python.org/.

[12] Godmar Back (2002): *DataScript-A specification and scripting language for binary data*. In: *Generative Programming and Component Engineering*, Springer, pp. 66–77.

[13] T. Boutell (1997): *PNG (Portable Network Graphics) Specification Version 1.0*. RFC Editor. Available at http://tools.ietf.org/html/rfc2083.

[14] KeithL. Clark (1978): *Negation as Failure*. In Herv Gallaire & Jack Minker, editors: *Logic and Data Bases*, Springer US, pp. 293–322, doi:10.1007/978-1-4684-3384-5_11. Available at http://dx.doi.org/10.1007/978-1-4684-3384-5_11.

[15] William F Dowling & Jean H Gallier (1984): *Linear-time algorithms for testing the satisfiability of propositional Horn formulae*. *The Journal of Logic Programming* 1(3), pp. 267–284.

[16] Kathleen Fisher & David Walker (2011): *The PADS project: an overview*. In: *Proceedings of the 14th International Conference on Database Theory*, ACM, pp. 11–17.

[17] Joseph C. Giarratano & Gary D. Riley (2005): *Expert Systems: Principles and Programming*. Brooks/Cole Publishing Co., Pacific Grove, CA, USA.

[18] Per Gustafsson & Konstantinos Sagonas (2005): *Bit-level binaries and generalized comprehensions in Erlang*. In: *Proceedings of the 2005 ACM SIGPLAN workshop on Erlang*, ACM, pp. 1–8.

[19] Ernest Friedman Hill (2003): *Jess in Action: Java Rule-Based Systems*. Manning Publications Co., Greenwich, CT, USA.

[20] ISO/IEC (1993): *ISO/IEC 11172-3:1993 - Information technology – Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s – Part 3: Audio*. Padrão.

[21] Peter J McCann & Satish Chandra (2000): *Packet types: abstract specification of network protocol messages*. *ACM SIGCOMM Computer Communication Review* 30(4), pp. 321–333.

[22] Stuart J. Russell & Peter Norvig (2003): *Artificial Intelligence: A Modern Approach*, 2 edition. Pearson Education.