Specifying Message Formats with Contiguity Types

Konrad Slind ✉
Trusted Systems Group, Collins Aerospace, Minneapolis, USA

Abstract

We introduce *Contiguity Types*, a formalism for network message formats, aimed especially at self-describing formats. Contiguity types provide an intermediate layer between programming language data structures and messages, offering a helpful setting from which to automatically generate decoders, filters, and message generators. The syntax and semantics of contiguity types are defined and used to prove the correctness of a matching algorithm which has the flavour of a parser generator. The matcher has been used to enforce semantic well-formedness conditions on complex message formats for an autonomous unmanned avionics system.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases Logic, verification, formal language theory, message format languages

Digital Object Identifier 10.4230/LIPIcs.ITP.2021.30

Supplementary Material HOL4 formalization may be found at:
Model: `https://github.com/loonwerks/splat/blob/monitors/contig/abscontigScript.sml`
archived at `swh:1:cnt:62ff99d4ed11f6d7f8452467e27b8013398d3577`

Acknowledgements This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions and/or findings expressed are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Discussions with Robby, Luke Ryon, Dave Greve, David Hardin, Magnus Myreen, and Michael Norrish have greatly improved this work.

1 Introduction

Serialized data, for example network messages, is an important component in many computer systems.[1] As a result, innumerable libraries and tools have been created that use high level specifications as a basis for automating the creation, validation, and decoding of such data. Usually, these high level specifications describe the format of a message in terms of how the elements (fields) of the message are packed side-by-side to make the full message. When the size of each field is known in advance, there are really no conceptual difficulties. However, messages can be more complicated than that.

The main source of difficulty is *self-describing* messages: those where information embedded in fields of the message determines the final structure of the message. Two of the main culprits are variable-length arrays and unions. A *variable-length array* is a field where the number of elements in the field depends on the value of some already-seen field (or, more generally, as the result of a computation involving previously-seen information in the message). The length is therefore a value determined at runtime. A *union* is deployed when some information held in a message is used to determine the structure of later portions of the message. For example, unions can be used to support versioning where version i has n fields, and version $i + 1$ has $n + 1$. In settings where both versions need to be supported in a single format, it can make sense to encode the version handling inside the message, and unions are how this can be specified.

[1] DISTRIBUTION STATEMENT A. Approved for public release.

$$
\begin{array}{rcl}
\mathit{base} & = & \mathsf{bool} \mid \mathsf{char} \mid \mathsf{u8} \mid \mathsf{u16} \mid \mathsf{u32} \mid \mathsf{u64} \mid \mathsf{i16} \mid \mathsf{i32} \mid \mathsf{i64} \mid \mathsf{float} \mid \mathsf{double} \\
\tau & = & \mathit{base} \\
& \mid & \mathsf{Recd}\ (f_1 : \tau_1) \ldots (f_n : \tau_n) \\
& \mid & \mathsf{Array}\ \tau\ \mathit{exp} \\
& \mid & \mathsf{Alt}\ \mathit{bexp}\ \tau\ \tau
\end{array}
$$

Figure 1 Contiguity types.

We believe that tools and techniques from formal language theory such as regular expressions, automata, grammars, parser generators, etc. can provide an effective way to tackle message formats, and have been using the acronym *SPLAT* (Semantic Properties for Language and Automata Theory) to refer to this approach. For example, we have used regular expressions as a specification language for message formats having simple interval constraints on the values allowed in fields. Generation of the corresponding DFA results in an efficient table-driven automaton implementing the specified constraints, with a solid proof certificate connecting the original constraints with the DFA behavior [5].

However, self-describing data formats fall outside the realm of common formal language techniques; e.g., variable-length fields are clearly not able to be described by regular or context-free languages. (These language classes encompass repetitions of a fixed or unbounded size, but not repetitions of a size determined by parts of the input string.) It seems that context-sensitive grammars can, in principle, specify such information, but there are few tools supporting context sensitive languages. Knuth introduced attribute grammars [8] for dealing with context-sensitive aspects of parsing, and those techniques address similar problems to ours. Another possibility would be to use *parser combinators* in order to quickly stitch together a parser; it seems likely that the combinators can be instrumented to gather and propagate contextual information. However, we are seeking a high level of formal specification and automation, while still being rooted in formal languages, with their emphasis on sets of strings as the basic notion.

2 Contiguity Types

The characteristic property of a message is *contiguity*: all the elements of the message are laid out side-by-side in a byte array (or string). Our assumption is that a message is the *result* of a map from structured data and we will rely on a basic collection of programming language types to capture that structure. Contiguity types (Figure 1) start with common base types (booleans, characters, signed and unsigned integers, etc.) and are closed under the construction of records, arrays, and unions. [2]

Notice that τ is defined in terms of a type of arithmetic expressions *exp* and also *bexp*, boolean expressions built from *exp*. Now consider

$$\mathsf{Array}\ \tau\ \mathit{exp}\ .$$

For this to specify a varying length array dependent on other fields of the message, its dimension *exp* should be able to refer to the *values* of those fields. The challenge is just how to express the concept of "other fields", i.e., we need a notation to describe the *location* in

[2] We will use the terms "*contiguity type*, contig, and τ interchangeably.

the message buffer where the value of a field can be accessed. Our core insight is that this is similar to a problem that programming language designers had in the 60s and 70s, resolved by the notions of *L-value* and *R-value*. The idea is originally due to Christopher Strachey in CPL [13] and developed subsequently, for example by Dennis Ritchie in C [12].

Before getting into formal details, we discuss a few examples. We will use familiar notation: records are lists of $filedname : \tau$ elements enclosed by braces; an array field Array c dim is written $c\,[dim]$; and Alt b τ_1 τ_2 is written '`if` b `then` τ_1 `else` τ_2'. "Cascaded" Alts may be written in Lisp "cond" style, i.e., as

$$\begin{array}{ll} \mathsf{Alt} & b_1 \longrightarrow \tau_1 \ldots \\ & b_n \longrightarrow \tau_n \\ & \text{otherwise} \longrightarrow \tau_{n+1} \end{array}$$

1. The following is a record with no self-describing aspects: each field is of a statically known size.

```
{A : u8
 B : {name : char [13]
      cell : i32}
 C : bool
}
```

   The `A` field is specified to be an unsigned int of width 8 bits, the `B` field is a record, the first element of which is a character array of size 13, and the second element of which is a 32 bit integer; the last field is specified to be a boolean.
2. Variable-sized strings are a classic self-describing aspect. In this example the contents of the `len` field determines the number of elements in the `elts` field.

```
{ len : u16
  elts : char [len]
}
```

3. The following example shows the Alt construct being used to support multiple versions in a single format. Messages with the value of field `versionID` being less than 14 have three fields in the message, while all others have two.

```
{versionID : u8
 versions : if versionID < 14 then
               { A : i32, B : u16})
            else { Vec : char [13]}
 }
```

4. The following is a contrived example showing the need for resolution of multiple similarly named fields; it also shows how the information needed to determine the message structure may be deeply buried in some fields.

```
{len : u16
 A : {len : u16
      elts : u16[len]
     }
 B : char [A.len - 1 * len]
 C : i32 [A.elts[0]]
}
```

$$
\begin{array}{rcl}
\mathit{lval} & = & \mathit{varname} \mid \mathit{lval}\,[\mathit{exp}] \mid \mathit{lval}.\mathit{fieldname} \\
\mathit{exp} & = & \mathsf{Loc}\ \mathit{lval} \mid \mathsf{nLit}\ \mathsf{nat} \mid \mathit{constname} \mid \mathit{exp} + \mathit{exp} \mid \mathit{exp} * \mathit{exp} \\
\mathit{bexp} & = & \mathsf{bLoc}\ \mathit{lval} \mid \mathsf{bLit}\ \mathsf{bool} \mid \neg \mathit{bexp} \mid \mathit{bexp} \wedge \mathit{bexp} \mid \mathit{exp} = \mathit{exp} \mid \mathit{exp} < \mathit{exp}
\end{array}
$$

Figure 2 L-values, expressions, and boolean expressions.

2.1 Expressions, L-values, and R-values

In programming languages, an *L-value* is an expression that can occur on the left-hand side of an assignment statement. Similarly, an *R-value* can occur on the right-hand side of assignments. Following are a few examples:

```
x := x + 1
A[x] := B.y + 42
A[x].lens.fst[7] := MAX_LEN * 1024 + B.y
```

Figure 2 presents the formal syntax for L-values, R-values, and the boolean expressions we will use. An L-value can be a variable, an array index, or a record field access. R-values are arithmetic expressions that can contain L-values (we will use *exp* interchangeably with R-value).

An L-value denotes an *offset* from the beginning of a data structure, plus a *width*. In an R-value, an occurrence of an L-value is mapped to the value of the patch of memory between *offset* and *offset* + *width*. For the purpose of specifying message formats, it may not be immediately obvious that a notation supporting assignment in imperative languages can help, but there is indeed a form of assignment lurking.

The above explanation of L-values centers on indices into a byte buffer; in the following we will give a mild variant of this: instead of indices into the buffer, we lift out the designated slices. Thus, given environments $\theta : \mathit{lval} \mapsto \mathsf{string}$ (binding L-values to strings), and functions $\mathsf{toN} : \mathsf{string} \to \mathbb{N}$ and $\mathsf{toB} : \mathsf{string} \to \mathsf{bool}$ (which interpret byte sequences to numbers and booleans, respectively), expression evaluation and boolean expression evaluation have conventional definitions:

$$
\mathsf{evalExp}\ \theta\ e = \mathsf{case}\ e \begin{cases}
\mathsf{Loc}\ \mathit{lval} & \Rightarrow & \mathsf{toN}(\theta(\mathit{lval})) \\
\mathsf{nLit}\ n & \Rightarrow & n \\
e_1 + e_2 & \Rightarrow & \mathsf{evalExp}\ \theta\ e_1 + \mathsf{evalExp}\ \theta\ e_2 \\
e_1 * e_2 & \Rightarrow & \mathsf{evalExp}\ \theta\ e_1 * \mathsf{evalExp}\ \theta\ e_2
\end{cases}
$$

$$
\mathsf{evalBexp}\ \theta\ b = \mathsf{case}\ b \begin{cases}
\mathsf{bLoc}\ \mathit{lval} & \Rightarrow & \mathsf{toB}(\theta(\mathit{lval})) \\
\mathsf{bLit}\ b & \Rightarrow & b \\
\neg b & \Rightarrow & \neg(\mathsf{evalBexp}\ \theta\ b) \\
b_1 \vee b_2 & \Rightarrow & \mathsf{evalBexp}\ \theta\ b_1 \vee \mathsf{evalBexp}\ \theta\ b_2 \\
b_1 \wedge b_2 & \Rightarrow & \mathsf{evalBexp}\ \theta\ b_1 \wedge \mathsf{evalBexp}\ \theta\ b_2 \\
e_1 = e_2 & \Rightarrow & \mathsf{evalExp}\ \theta\ e_1 = \mathsf{evalExp}\ \theta\ e_2 \\
e_1 < e_2 & \Rightarrow & \mathsf{evalExp}\ \theta\ e_1 < \mathsf{evalExp}\ \theta\ e_2
\end{cases}
$$

▶ Remark 1 (Partiality). Expression evaluation is partial because there is no guarantee that $\theta(\mathit{lval})$ is defined: an *lval* being looked-up may not be in the map θ. Failure in evaluation is modelled by the option type, and must be handled in the semantics and the matching algorithm. However error handling is omitted in the presentation since it hampers readability. See the HOL4 formalization for full details.

2.2 Semantics

We now confess to misleading the reader: in spite of the notational similarity, a contiguity type is *not* a type: it is a formal language. A type is usually understood to represent a set, or domain, of values, e.g., the type int32 represents a set of integers. In contrast, the contiguity type i32 represents the set of strings of width 32 bits. An element of a contiguity type can be turned into an element of a type by providing interpretations for all the strings at the leaves and interpreting the Recd and Array constructors into the corresponding type constructs. (A base contiguity type therefore serves mainly as a *tag* to be interpreted as a width and also as an intended target type.) Thus, contiguity types sit – conveniently – between the types in a programming language and the strings used to make messages.

The semantics definition depends on a few basic notions familiar from language theory: language concatenation, and iterated language concatenation.

$$\begin{aligned} L_1 \cdot L_2 &= \{w_1 w_2 \mid w_1 \in L_1 \wedge w_2 \in L_2\} \\ L^0 &= \varepsilon \\ L^{n+1} &= L \cdot L^n \end{aligned}$$

▶ **Definition 2** (Semantics of contiguity types). *In the following, we assume given an assignment θ for evaluating expressions. If an expression evaluation fails, the language being constructed will be $\emptyset$.*

$$\mathcal{L}_\theta(\tau) = \texttt{case}\ \tau \begin{cases} \mathit{base} \Rightarrow \{s \mid \mathsf{len}(s) = \mathsf{width}(\mathit{base})\} \\ \mathsf{Recd}\ (f_1 : \tau_1) \ldots (f_n : \tau_n) \Rightarrow \mathcal{L}_\theta(\tau_1) \cdot \ldots \cdot \mathcal{L}_\theta(\tau_n) \\ \mathsf{Array}\ \tau\ \mathit{exp} \Rightarrow \\ \quad \begin{cases} \mathcal{L}_\theta(\tau)^{\mathsf{evalExp}\ \theta\ \mathit{exp}} & \text{if } \mathsf{evalExp}\ \theta\ \mathit{exp} \text{ succeeds} \\ \emptyset & \textit{if evaluation fails} \end{cases} \\ \mathsf{Alt}\ \mathit{bexp}\ \tau_1\ \tau_2 \Rightarrow \\ \quad \begin{cases} \mathcal{L}_\theta(\tau_1) & \text{if } \mathsf{evalBexp}\ \theta\ \mathit{bexp} = \mathsf{true} \\ \mathcal{L}_\theta(\tau_2) & \text{if } \mathsf{evalBexp}\ \theta\ \mathit{bexp} = \mathsf{false} \\ \emptyset & \textit{if evaluation fails} \end{cases} \end{cases}$$

▶ **Example 3.** Consider the following schema for an *option* contiguity type. The empty record `{}` associated with boolean expression b has no fields.

`if` b `then { } else` c

In case b evaluates to true, no portion of the string is consumed; otherwise, c specifies the remainder of the processing. It may be instructive to consider how this type works with arrays. For example, a string meeting the following contig specification

```
(if b then {} else i32) [3]
```

is either zero or twelve bytes in length (assuming that `i32` is four bytes wide).

3 Algorithms

The following are classical topics in formal language theory and practice, and they are worth investigating in the context of contiguity types. At present we have been working on decoding, filtering, and test generation.

Decoding A decoder breaks a sequence of bytes up and puts the pieces into a useful data structure, typically a parse tree. We will discuss this in more detail in Section 3.1.

Filtering A filter computes an answer to the question: "does a sequence of bytes meet the specification of a given contiguity type". This is an instance of the language recognition problem. More powerful filters enforce that certain fields of a message, when interpreted, meet specific semantic properties. We will discuss this further in Section 4.1.

Serialization Given a contiguity type, synthesize a function that writes a compact binary version of a data structure to a message.

Test generation Given a contiguity type, generate byte sequences that do (or do not) meet its specification and feed the sequences to implementations in order to observe their behaviour.

Learning Given training sets of messages that are accepted/rejected by an implementation, attempt to discover a contiguity type for the entire set of messages.

3.1 Decoding

Above we mentioned that decoding can result in parse trees; however, self-describing messages allow a different conceptual framework to be brought to bear. There is an important distinction between parsing, which *creates* structure (parse trees), and matching, which is given structure and calculates assignments (substitutions).[3] Giving some evocative types helps make the difference clear:

$$\mathsf{parse} : \mathit{grammar} \to \mathsf{string} \to \mathit{parsetree}$$
$$\mathsf{match} : \mathit{pattern} \to \mathsf{string} \to \mathit{assignments}$$

For our purposes, namely decoding datastructures in binary format, the central decoding algorithm is a *matcher*: given a contiguity type τ and a string s, the matcher will either fail, or succeed with an assignment $\theta : \mathit{lval} \mapsto \mathsf{string}$ mapping each L-value in τ to its corresponding slice of s. The assignment θ can be post-processed to yield a standard parse tree, but its novelty and strength is that θ can be dynamically consulted to access the values needed to guide the processing of self-describing messages.

▶ **Definition 4** (Matching algorithm). *The matcher operates over a triple* $(\mathit{worklist}, \mathit{str}, \theta)$ *where worklist is a stack used to linearize the input contiguity type* τ*, str represents the remainder of the input string, and* θ *is the assignment being built up. Each element of the worklist is a* (τ, lval) *pair, where* τ *is a* contig*, and lval is the path growing down from the root to* τ*. The notation* $(\mathit{lval} \mapsto \mathit{slice}) \bullet \theta$ *denotes the addition of binding* $\mathit{lval} \mapsto \mathit{slice}$ *to* θ*. We examine the cases in turn:*

1. *The worklist is empty; the match succeeds.*

$$([], \mathit{str}, \theta) \Rightarrow \mathsf{SOME}(\mathit{str}, \theta)$$

2. *The first element of the worklist is a base type. The prescribed number of bytes are broken off the front of the string, giving* $\mathit{str} = (\mathit{slice}, \mathit{rst})$*; then the binding is added to* θ *before recursing. If the string is shorter than the requested number of bytes, fail.*

$$((\mathsf{Basic}\ a, \mathit{lval}) :: t, \mathit{str}, \theta) \Rightarrow (t, \mathit{rst}, (\mathit{lval} \mapsto \mathit{slice}) \bullet \theta)$$

[3] Thus the notion of matching discussed here is in the tradition of term rewriting [1], the main difference being that our substitutions are applied to *lval*s rather than variables.

3. *The first element of the worklist is* $recd = \mathsf{Recd}\ (f_1 : \tau_1) \ldots (f_n : \tau_n)$. *Before recursing, the fields are pushed onto the stack, extending the path to each field element:*

$$((recd, lval) :: t, str, \theta) \Rightarrow ([(\tau_1, lval.f_1), \cdots, (\tau_n, lval.f_n)]@t, str, \theta)$$

4. *The first element of the worklist is an array. The dimension expression is evaluated to get the width* d*, then* d *copies are pushed onto the stack, where each path is extended with the array index.*

$$((\mathsf{Array}\ \tau\ exp, lval) :: t, str, \theta) \Rightarrow ([(\tau, lval[0]), \cdots, (\tau, lval[d-1])]@t, str, \theta)$$

5. *The first element of the worklist is an* Alt. *If* b *evaluates to* true, τ_1 *is pushed on to the worklist; if it evaluates to* false, τ_2 *is pushed. Otherwise, fail.*

$$((\mathsf{Alt}\ b\ \tau_1\ \tau_2, lval) :: t, str, \theta) \Rightarrow ((\tau_i, lval) :: t, str, \theta)$$

The matcher function, match *begins with an initial state*

$$state_0 = ([(root, \tau)], str_0, \emptyset)$$

where the initial path is a default lval variable named root, the initial string is str_0*, and the initial assignment has no bindings.*

▶ **Theorem 5** (Matcher termination). *As mentioned, the state of the matcher is held in a* $(worklist, str, \theta)$ *tuple. The termination relation is a lexicographic combination, where either str gets shorter, or, str is unchanged and worklist gets smaller under the multiset order. (The multiset order is useful in this proof since the handling of the* Array *construct is a nice version of the Hercules-Hydra problem [3].)*

▶ **Definition 6** (Substitution application). *Correctness depends on an operation* $\theta\ lval\ \tau$ *applying substitution* θ *to contiguity type* τ*, starting at* $lval$*, in order to reconstruct the original string.*

$$\theta\ lval\ \tau = \mathsf{case}\ \tau \begin{cases} base \Rightarrow \theta(lval) \\ \mathsf{Recd}\ (f_1 : \tau_1) \ldots (f_n : \tau_n) \Rightarrow \theta\ (lval.f_1)\ \tau_1 \cdot \ldots \cdot \theta\ (lval.f_n)\ \tau_n \\ \mathsf{Array}\ \tau_1\ exp \Rightarrow \\ \quad \begin{cases} \theta\ (lval[0])\ \tau_1 \cdot \ldots \cdot \theta\ (lval[d-1])\ \tau_1, & \text{if } d = \mathsf{evalExp}\ \theta\ exp \\ \emptyset & \text{if evaluation fails} \end{cases} \\ \mathsf{Alt}\ bexp\ \tau_1\ \tau_2 \Rightarrow \\ \quad \begin{cases} \theta\ lval\ \tau_1 & \text{if } \mathsf{evalBexp}\ \theta\ bexp = \mathsf{true} \\ \theta\ lval\ \tau_2 & \text{if } \mathsf{evalBexp}\ \theta\ bexp = \mathsf{false} \\ \emptyset & \text{if evaluation fails} \end{cases} \end{cases}$$

▶ **Theorem 7** (Correctness of substitution). *The correctness statement for the matcher is similar to those found in the term rewriting literature, namely that the computed substitution applied to the contiguity type yields the original string:*

$$\mathsf{match}\ state_0 = \mathsf{SOME}(\theta, s) \Rightarrow \theta\ root\ \tau \cdot s = str_0$$

Proof. By induction on the definition of match. ◀

▶ **Theorem 8** (Matcher soundness). *The connection to* $\mathcal{L}_\theta(\tau)$ *is formalized as*

$$str_0 = s_1 s_2 \wedge \mathsf{match}\ state_0 = \mathsf{SOME}(\theta, s_2) \Rightarrow s_1 \in \mathcal{L}_\theta(\tau)$$

Proof. By induction on the definition of match. ◀

In other words, a successful match provides a θ that will successfully evaluate all encountered expressions, and the matched string is indeed in the language of τ. A completeness theorem going in the other direction has not yet been tackled.

▶ **Example 9.** Given the contig

```
{A : Bool
 B : Char
 len : u16
 elts : i32 [len]
}
```

and an input string (listed in hex)

```
[0wx1, 0wx67, 0wx0, 0wx5, 0wx0, 0wx0, 0wx0, 0wx19, 0wx0, 0wx0,
 0wx9, 0wx34, 0wx0, 0wx0, 0wx30, 0wx39, 0wx0, 0wx0, 0wxD4,
 0wx31, 0wxFF, 0wxFF, 0wxFE, 0wxB3]
```

created by encoding: the boolean `true`, the letter `g`, the number 5 (MSB 2 byte unsigned), and the five MSB 4 byte signed twos complement integers 25, 2356, 12345, 54321, and -333, the matcher creates the following assignment of *lval*s to substrings of the input:

```
[(root.A,       (Bool, [0wx1])),
 (root.B,       (Char, [0wx67])),
 (root.len,     (u16,  [0wx0, 0wx5])),
 (root.elts[0], (i32,  [0wx0, 0wx0, 0wx0, 0wx19])),
 (root.elts[1], (i32,  [0wx0, 0wx0, 0wx9, 0wx34])),
 (root.elts[2], (i32,  [0wx0, 0wx0, 0wx30, 0wx39])),
 (root.elts[3], (i32,  [0wx0, 0wx0, 0wxD4, 0wx31])),
 (root.elts[4], (i32,  [0wxFF, 0wxFF, 0wxFE, 0wxB3]))
]
```

Note that each element of the list is of the form $(lval, (tag, bytes))$ where each slice is labelled with its corresponding base type, to support further translation.

Thus the matcher will break up the input string in accordance with the specification; the execution, in effect, generates a sequence of assignments that, if applied, would populate a data structure with the specified data in the specified places. Therefore it is not really necessary to generate parse trees to in order to decode messages: one merely needs a target data structure to write data into. (In fact, when filtering, no target data structure is needed at all.) The correctness property will ensure that *all* fields are written with the specified data. The assignments can be incrementally evaluated as the decoder runs, or can be stored and applied when the decoder terminates.

4 Extended contiguity types

In the discussion so far, contiguity types can only express bounded data: each base type has a fixed size and all Array types are given an explicit bound. Removing these two restrictions would greatly increase expressiveness. Of course, we look to the theory of formal languages to guide extensions to the formalism. We have thus explored the addition of the empty language $\emptyset$, Kleene star, and a lexer. The augmented syntax can be seen in Figure 3. The addition of $\emptyset$ (via Void) and Kleene star (via List) has been accomplished, along with the

$$
\begin{array}{rcl}
\tau & = & \mathsf{Base}\ (\mathit{regexp} \times \mathit{valFn}) \\
& | & \mathsf{Void} \\
& | & \mathsf{List}\ \tau \\
& | & \mathsf{Recd}\ (f_1 : \tau_1) \ldots (f_n : \tau_n) \\
& | & \mathsf{Array}\ \tau\ \mathit{exp} \\
& | & \mathsf{Alt}\ \mathit{bexp}\ \tau_1\ \tau_2
\end{array}
$$

Figure 3 Extended contiguity types.

proofs verifying the upgraded matcher. We also discuss replacing the existing base types with a lexer for completeness, even though that discussion more appropriately belongs to future work.

▶ **Definition 10** (Semantics additions). *A base type element has the form* Base(*regexp*, *valFn*), *where valFn is a function that maps the string recognized by regexp to a data value. The semantics of a* Base *type is just the (formal language) semantics of its regular expression,* Void *denotes the empty set, and* List *is a "tagged" version of Kleene star (*NilTag *and* ConsTag *are described in Section 4.2).*

$$
\mathcal{L}_\theta(\tau) = \mathsf{case}\ \tau \begin{cases} \mathsf{Base}\ (\mathit{regexp}, \mathit{valFn}) \Rightarrow \mathcal{L}(\mathit{regexp}) \\ \mathsf{Void} \Rightarrow \emptyset \\ \mathsf{List}\ \tau \Rightarrow (\mathsf{ConsTag} \cdot \mathcal{L}_\theta(\tau))^* \cdot \mathsf{NilTag} \\ \ldots \textit{(previous clauses)} \end{cases}
$$

4.1 Void and in-message assertions

The Alt constructor combined with Void supports an *in-message assertion* feature. The contiguity type Assert *bexp* is defined as follows:

▶ **Definition 11** (Assert).

$$\mathsf{Assert}\ \mathit{bexp} = \mathsf{Alt}\ \mathit{bexp}\ (\mathsf{Recd}\ [\,])\ \mathsf{Void}$$

The meaning of Assert b is obtained by simplification:

$$\mathcal{L}_\theta(\mathsf{Assert}\ \mathit{bexp}) = \texttt{if}\ \mathsf{evalBexp}\ \theta\ \mathit{bexp}\ \texttt{then}\ \varepsilon\ \texttt{else}\ \emptyset$$

and match evaluates it by failing when evalBexp θ *bexp* is false, and otherwise continuing on without advancing in the input.

▶ **Example 12** ($A^n B^n C^n$). It is well known that $L = A^n B^n C^n$ is not a context-free language. We can use Assert to specify a language that is *nearly* L with the following contiguity type:

```
charA = {ch : char, isA : Assert (ch = 65)}   (* "A" = ASCII 65 *)
charB = {ch : char, isB : Assert (ch = 66)}
charC = {ch : char, isC : Assert (ch = 67)}

mesg = {len : u16
        A : charA [len]
        B : charB [len]
        C : charC [len]
       }
```

In fact $\mathcal{L}_\theta(\mathsf{mesg}) = u \cdot A^{\mathsf{toN}(u)} \cdot B^{\mathsf{toN}(u)} \cdot C^{\mathsf{toN}(u)}$.

Assert expressions have been used extensively when specifying wellformedness properties for messages in our application work. The applications include restrictions on array sizes and constraints on array elements, e.g., requiring that every element in an array of GPS coordinates is an acceptable GPS coordinate.

▶ **Example 13** (Array limits)**.** In UxAS messages (see Section 5) the length of every array element is held in a separate *length* field which is two bytes in size. Thus the following contig, in the absence of any further constraint, supports arrays of length up to 65536 elements. A receiver system may well not be prepared for messages having collections of such potentially large components.

```
{ len : u16
  elts : i32 [len] }
```

In the meta-data for such messages, one can sometimes find information about the maximum expected size, usually a fairly small number. This can be directly expressed inside the contig with an Assert:

```
{ len : u16
  len-range : Assert (len <= 8)
  elts : i32 [len] }
```

Note that the expected array length should be specified before the array itself, otherwise the allocation attempt might be made before the check.

4.2 Kleene Star

Although the use of bounded Array types provides much expressiveness for representing sequences of data, ultimately some kinds of message can not be handled, i.e., those where there is no way to predict the number of nestings of structure: s-expressions, logical formulae, and programming language syntax trees are some typical examples. We address this shortcoming by adding a new contiguity type constructor – List – of unbounded lists. A message matching a List τ type will be subject to an encoding similar to implementations of lists in functional languages. The matching algorithm for contiguity types is extended to handle List objects by iteratively unrolling the recursive equation

$$L^* = \varepsilon \cup L \cdot L^*$$

Indeed the type List τ is represented by the following contiguity type, a recursive record:

$$\mathsf{List}\ \tau = \begin{cases} \mathsf{tag} & : & \mathsf{u8} & & & \\ \mathsf{test} & : & \mathsf{Alt} & (\mathsf{tag} = \mathsf{NilTag}) & \longrightarrow & \varepsilon \\ & & & (\mathsf{tag} = \mathsf{ConsTag}) & \longrightarrow & \{\mathsf{hd} : \tau,\ \mathsf{tl} : \mathsf{List}\ \tau\} \\ & & & \text{otherwise} & \longrightarrow & \mathsf{Void} \end{cases}$$

In words, a List τ matches a sequence of records where a single-byte tag (NilTag or ConsTag) is read, then tested to see whether to stop parsing the list (NilTag) or to continue on to parse a τ into the hd field and recurse in order to process the remainder of the list. An incorrect value for the tag results in failure. Thus, the list of integers

$$\mathsf{Cons}(1, \mathsf{Cons}(2, \mathsf{Cons}(3, \mathsf{Nil})))$$

can be represented in a message as (assume Code is an encoder for integers)

$$\mathsf{ConsTag} \cdot \mathsf{Code}(1) \cdot \mathsf{ConsTag} \cdot \mathsf{Code}(2) \cdot \mathsf{ConsTag} \cdot \mathsf{Code}(3) \cdot \mathsf{NilTag}$$

and match, given type List int (where int is a contiguity type for some flavor of integer) succeeds, returning the context

$$\begin{aligned}
\text{root.tag} &\mapsto \text{ConsTag} \\
\text{root.hd} &\mapsto \text{Code}(1) \\
\text{root.tl.tag} &\mapsto \text{ConsTag} \\
\text{root.tl.hd} &\mapsto \text{Code}(2) \\
\text{root.tl.tl.tag} &\mapsto \text{ConsTag} \\
\text{root.tl.tl.hd} &\mapsto \text{Code}(3) \\
\text{root.tl.tl.tl.tag} &\mapsto \text{NilTag}
\end{aligned}$$

This solution is compositional, in the sense that List types can be the arguments of other contiguity types, can be applied to themselves, e.g., List(List τ), etc. Thus quite general branching structures of arbitrary finite depth and width can be specified and parsed with this extension. The approach captures a certain class of *context-free-like* languages. However, it differs distinctly from the standard Chomsky hierarchy, mainly because sums are determined by *looking behind* when computing which choice to follow in an Alt type; for example, the list parser branches *after* it has seen the tag. The similarity with 'no-lookahead' parsing, such as LL(0) and LR(0), deserves further investigation.

▶ **Example 14** (First order term challenge)**.** Although lists of contig types can be straightforwardly constructed with the above encoding, there remains a problem when lists are part of a recursive construction. A classic example is *first order terms*, as described by the following ML-style datatype:

```
term = Var of string
     | App of string * term list
```

The following contiguity types capture a binary encoding of `term`, using tags to distinguish the two kinds of term:

$$\text{string} = \{\text{len} : \text{u16}, \text{elts} : \text{char}\,[\text{len}]\}$$

$$\text{term} = \begin{cases} \text{tag} : \text{u8} & & & \\ \text{test} : \text{Alt} & (\text{tag} = \text{VarTag}) & \longrightarrow & \{\text{varName} : \text{string}\} \\ & (\text{tag} = \text{AppTag}) & \longrightarrow & \{\text{fnName} : \text{string},\ \text{Args} : \text{List term}\} \\ & \text{otherwise} & \longrightarrow & \text{Void} \end{cases}$$

However, such *nested recursive* specifications demand more elaborate constructions, for example treating term and List as being mutually recursively defined, as is already done for nested recursive datatypes in theorem provers [6].

4.3 Lexing

Currently, the set of base contiguity types comprises the usual base types expected in most programming languages. Semantically, a base type denotes a set of strings of the specified width, but it is also coupled with an *interpretation function* for example, the contiguity type u8 denotes the set of all one-byte strings, interpreted by the usual unsigned valuation function:

$$\text{u8} = (\{s \mid \text{length}(s) = 1\}, \text{toN})$$

This approach cannot, however, capture base types such as string literals of arbitrary size, or bignums, or the situation in packed bit-level encodings where fields are of *ad hoc* sizes aimed at saving space. A common generalization is to express base types via regular expressions paired with interpretation functions. In that setting u8 can be defined as

$$\mathsf{u8} = (.\,, \mathsf{toN})$$

(where '.' is the standard regular expression denoting any character). Similarly,

$$\mathsf{Cstring} = ([\backslash 001 - \backslash 255]^*\backslash 000, \lambda x.\, x)$$

denotes the set of zero-terminated strings, as found in the C language. Its displayed interpretation is just the identity function, but could just as well be a function that drops the terminating \000 character.

Scott Owens has already formalized a HOL4 theory of regexp based, maximal munch, lexer generation, and it is future work to adapt contiguity types to use those lexemes instead of the current restricted set of base types.

5 Application

In the DARPA CASE project, we have been applying contiguity types to help create provably secure message filters and runtime monitors. One example we have been working with is *OpenUxAS*, which has been developed by the Air Force Research Laboratory.[4] UxAS is a collection of modular services that interact via a common message-passing architecture, aimed at unmanned autonomous systems. Each service subscribes to messages in the system and responds to queries. The content of each message conforms to the Light-weight Message Control Protocol (LMCP) format. In UxAS, software classes providing LMCP message creation, access, and serialization/deserialization are automatically generated from XML descriptions, which detail the exact data fields, units, and default values for each message. All UxAS services communicate with LMCP formatted messages.

An example LMCP message type is AirVehicleState. The following is its contiguity type:

```
AirVehicleState =
   {EntityState   : EntityState
    Airspeed      : float
    VerticalSpeed : float
    WindSpeed     : float
    WindDirection : float
   }
```

where an EntityState is quite an elaborate type:

```
{ ID : i64
  u  : float
  v  : float
  w  : float
  udot : float
  vdot : float
  wdot : float
  Heading : float
```

[4] See https://github.com/afrl-rq/OpenUxAS.

```
  Pitch    : float
  Roll     : float
  p : float,  q : float, r : float
  Course : float
  Groundspeed : float
  Location : mesgOption "LOCATION3D" location3D
  EnergyAvailable : float
  ActualEnergyRate : float
  PayloadStateList
     : uxasBoundedArray (mesgOption "PAYLOADSTATE" payloadState) 8
  CurrentWaypoint : i64
  CurrentCommand  : i64
  Mode : uxasNavigationMode
  AssociatedTasks : uxasBoundedArray i64 8
  Time : i64
  Info : uxasBoundedArray(mesgOption "KEYVALUEPAIR" keyValuePair) 32
}
```

Most of the fields are simple base types, but the Location field and the PayloadStateList are complex. They are expressed with some derived syntax, which we will explain.

- The Location field, `mesgOption "LOCATION3D" location3D`, may or may not occur (signalled with a tag field), but if it does, it is a location3D, which is a GPS location, and its latitude, longitude, and altitude fields are checked with an Assert to make sure they lie within the expected numeric ranges, which are expressed as floating point numbers.

```
AltitudeType = AGL | MSL

location3D = {
 Latitude  : double,
 Longitude : double,
 Altitude  : float,
 AltitudeType : AltitudeType,
 Wellformed : Assert (
     -90.0 <= Latitude <= 90.0 and
    -180.0 <= Longitude <= 180.0 and
     0.0 <= Altitude <= 15000.0)
}
```

- The PayloadStateList is a variable-length array of optional records, with maximum length 8. Each record has, along with other fields, its own variable-length array of key-value pairs, and the key and value of each such pair is a variable-length string.

We have formalized most of the LMCP messages as contiguity types, and created filters and parsers by instantiating the match algorithm. In order to meet the demands of LMCP message modelling, the matcher algorithm has been upgraded to support a fuller expression and boolean expression language, but the core algorithm is the same as our verified core version. The filters and parsers have been added to an existing UxAS design and successfully tested with the UxAS simulator.

6 Extensions and future work

Various extensions have been easy to add to the contiguity type framework, and we also have more substantial ideas to pursue for future work.

Enumerations An *enumeration* declaration introduces a new base contiguity type, and also adds the specified elements to a map associating constant names to numbers. Suppose that enumerations are allowed to have up to 256 elements, allowing any enumerated element to fit in one byte. The following enumeration is taken from UxAS messages:

```
NavigationMode
  = Waypoint | Loiter | FlightDirector
  | TargetTrack | FollowLeader | LostComm
```

A field expecting a NavigationMode element will be one byte wide, and thus there are 250 byte patterns that should not be allowed in the field. Thus, the contig

```
{ A : NavigationMode }
```

should be replaced by

```
{ A : NavigationMode
  A-range : Assert (A <= 5)
}
```

Raw blocks A raw chunk of a string (byte array) of a size that can depend on the values of earlier fields is easy to specify:

Raw *exp*

For example, a large Array form can lead to a large number of L-values being stored in θ; if none are ever accessed later, e.g., if the array is some image data, it can be preferable to simply declare a Raw block. Thus a 2D array can be blocked out in the following manner:

```
{ rows : i32
  cols : i32
  block : Raw (rows * cols)
}
```

Guest scanners It seemed useful to provide a general ability to host scanning functions. This is accomplished via the following constructor:

Scanner ($scanfn$: string $\rightarrow$ (string $\times$ string)option)

When a custom scanner is encountered during the matching process, the scanner is invoked on the input and should either fail or provide an (s_1, s_2) pair representing a splitting of the input. Then s_1 is added to θ at the current *lval*, and matching continues on s_2.

Non-copying implementations In the discussion so far, we have assumed that the input string is being broken up into substrings that are placed into the *lval* map θ. However, very little is changed if, instead of a substring, an *lval* in θ maps to a pair of indices $(pos, width)$ designating the location of the substring. The result is a matcher that never copies byte buffer data. This is necessary to synthesize efficient filters.

In making this representation change, there is a slight change to the semantics. In the original, $\theta(lval)$ yields a string whereas in the non-copying version, $\theta(lval)$ yields a pair of indices, which means that the original string str_0 needs to be included in applying the assignment.

Compilation Our current implementation of contiguity types is in an *interpreted* style: the evaluation of numeric Array bounds and boolean guards on Alt conditions is done with respect to the current context, which is explicitly accumulated as the message is processed. However, notice that the host programming language will no doubt already provide compilation for numeric and boolean expressions. This leads to the idea of *compiling contiguity types*: the current matching algorithm for contiguity types can be replaced by the generation of equivalent host-language code which is then compiled by the host-language compiler and evaluated. Although the current contiguity type matcher has proved to be fast enough to keep up with the real-time demands of the UxAS system, we expect that a compiled version of the code would be much faster.

We would like to formalize the compilation algorithm and prove it correct. Since the CakeML[9] formalization provides an operational semantics for CakeML programs, and a convenient translation from HOL4 expressions to CakeML ASTs, one should be able to prove a correctness theorem relating the matcher function of the present paper with a compiler that takes a contiguity type and generates CakeML. However, this is only speculative; there are many details to work out.

Relationship with grammars Contiguity types and match provide a type-directed and context-oriented parser generator that has some similarities with LL(0) or LR(0) languages wherein the parser can proceed with no lookahead. This is useful for binary-encoded datastructures. It would be very interesting to attempt to bridge the gap with conventional parsing technology based on grammars. A good beginning would be to understand the issues involved in attempting to translate context-free grammars into contiguity types and *vice versa*.

7 Related work

As mentioned in the introduction, domain specific languages for message formats have been around for a long time. Semantic definitions and verification for them is a much more recent phenomenon. The PADS framework [4] aimed at supporting a wide variety of formats, including text-based. Its core message description formalism was given semantics by translation into dependent type theory. An interesting integration of context-sensitivity into a conventional grammar framework has been done by Jim and Mandelbaum [7]. Everparse [11] is an impressive approach, based on parser combinators and having an emphasis on proving the invertibility of encode/decode pairs with automated proof (other properties are also established). Chlipala and colleagues [2] similarly emphasize encode/decode proofs, basing their work in Coq and using the power of dependent types to good effect. Formats based on dependent types can (and do) use the built-in expressive power of type theory to enforce semantic properties on data. A recent language in this vein is Parsely [10] which leverages the dependent records and predicate subtyping of PVS to provide a combination of PEG parsing and attribute grammars aimed at parsing complex language formats.

Many of these efforts obtain the semantics of the data description formalism by translation into features provided by a powerful host logic. In contrast, contiguity types use only very basic – and easily implemented – concepts. This means that contiguity type matchers, parsers, and extensions can be directly implemented in any convenient programming language. Contiguity types also provide a kind of dependency, without leveraging the type system of the theorem prover. We suspect that working at the representation level, and using L-values, allows one to get some of the benefits of type dependency. Another distinguishing aspect of our work is that our emphasis on filters means that we are primarily interested in the enforcement of semantic properties on message contents rather than encode/decode properties. In future work we expect to be able to leverage this in high performance filter implementations.

8 Conclusion

We have designed, formalized, proved correct, implemented, and applied a specification language for message formats, based on formal languages and the venerable notion of L- and R-values from imperative programming. The notion of contiguity type seems to give a lot of expressive power, sufficient to tackle difficult idioms in self-describing formats. Contiguity types integrate common structuring mechanisms from programming languages, such as arrays, records, and lists while keeping the foundation in sets of strings, which seems appropriate for message specifications.

References

1 Franz Baader and Tobias Nipkow. *Term Rewriting and all that.* Cambridge University Press, 1998.

2 Benjamin Delaware, Sorawit Suriyakarn, Clément Pit-Claudel, Qianchuan Ye, and Adam Chlipala. Narcissus: correct-by-construction derivation of decoders and encoders from binary formats. *Proc. ACM Program. Lang.*, 3(ICFP):82:1–82:29, 2019. doi:10.1145/3341686.

3 Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *CACM*, 22(8):465–476, 1979.

4 Kathleen Fisher, Yitzhak Mandelbaum, and David Walker. The next 700 data description languages. *J. ACM*, 57(2), 2010. doi:10.1145/1667053.1667059.

5 David S. Hardin, Konrad Slind, Mark Bortz, James Potts, and Scott Owens. A high-assurance, high-performance hardware-based cross-domain system. In Amund Skavhaug, Jérémie Guiochet, and Friedemann Bitsch, editors, *Computer Safety, Reliability, and Security - 35th International Conference, SAFECOMP 2016, Trondheim, Norway, September 21-23, 2016, Proceedings*, volume 9922 of *Lecture Notes in Computer Science*, pages 102–113. Springer, 2016. doi:10.1007/978-3-319-45477-1_9.

6 John Harrison. Inductive definitions: automation and application. In E. Thomas Schubert, Phillip J. Windley, and James Alves-Foss, editors, *Proceedings of the 1995 International Workshop on Higher Order Logic theorem proving and its applications*, number 971 in LNCS, pages 200–213, Aspen Grove, Utah, 1995. Springer-Verlag.

7 Trevor Jim and Yitzhak Mandelbaum. A new method for dependent parsing. In Gilles Barthe, editor, *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, volume 6602 of *Lecture Notes in Computer Science*, pages 378–397. Springer, 2011. doi:10.1007/978-3-642-19718-5_20.

8 Donald E. Knuth. Semantics of context-free languages. In *In Mathematical Systems Theory*, pages 127–145, 1968.

9 Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In *POPL '14: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 179–191. ACM Press, 2014. doi:10.1145/2535838.2535841.

10 Prashanth Mundkur, Linda Briesemeister, Natarajan Shankar, Prashant Anantharaman, Sameed Ali, Zephyr Lucas, and Sean Smith. Research report: The parsley data format definition language. In *2020 IEEE Security and Privacy Workshops (SPW)*, pages 300–307, 2020. doi:10.1109/SPW50608.2020.00064.

11 Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. Everparse: Verified secure zero-copy parsers for authenticated message formats. In Nadia Heninger and Patrick Traynor, editors, *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*,

pages 1465–1482. USENIX Association, 2019. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/delignat-lavaud.

12 Dennis Ritchie. The development of the C language. https://www.bell-labs.com/usr/dmr/www/chist.html.

13 Christopher Strachey. Fundamental concepts in programming languages. *High. Order Symb. Comput.*, 13(1/2):11–49, 2000. doi:10.1023/A:1010000313106.