# Semantic Properties for Language and Automata Theory (SPLAT)

Konrad Slind, Trusted Systems Group, Collins Aerospace

February 7, 2019

**Draft Status**. This document is a work in progress.

## Introduction

**SPLAT** (Semantic Properties for Language and Automata Theory) designates the application of language theory—which is based on *strings*—to help enforce, formulate and prove properties over a collection of other useful types, such as numbers, enumerations, records, and arrays. In particular, our focus is on applying tools and concepts from language theory to help automate the specification, creation, and verification of practical programs. As one example of this approach, we employ regular expressions to model and implement arithmetic constraints. This has been applied to the task of automating the generation and proof of correctness of encoders and decoders for network message formats.

### Language Theory and Encoding

The basic notions of theoretical computer science are often formulated in terms of strings of symbols drawn from an alphabet: automata, grammars, and Turing machines all use strings as their main data structure. On the practical side, the theory of grammars and regular expressions provides a solid basis for tools such as lexer and parser generators. In order to bridge the gap between strings and the rich universe of data structures used in computer science, *encoding* and *decoding* is employed. For example, showing the existence of a universal Turing machine requires an encoding/decoding scheme for the strings (Turing machines) being taken as arguments by the universal machine. In fact, the activity of packing data into a string format and then unpacking it is pervasive, not only in theoretical computer science, but also in practical contexts, *e.g.* data compression, data encryption, sending structured data over a network, *etc.*

## Regular expressions

(We are building on previously reported [safecomp2016] work in HOL4 providing a theory of regular expressions and a verified translator of regexps to table-driven DFAs (deterministic finite state automata). The formalization is available in the HOL4 distribution.)

Let $\mathbb{S}$ be the set of finite strings over an alphabet $\Sigma$. Throughout, we assume $\Sigma$ is the 256 ASCII characters. This supports the use of char list as a representation for $\mathbb{S}$, allowing the use of some list operations on elements of $\mathbb{S}$. A regular expression over the alphabet designates a set of strings $\mathbb{S} \rightarrow \textsf{bool}$. We use the following datatype to represent regular expressions:

$$\begin{array}{lcl} \textsf{regexp} & ::= & \textsf{Chset w64 w64 w64 w64} \\ & \mid & \textsf{Cat regexp regexp} \\ & \mid & \textsf{Or regexp list} \\ & \mid & \textsf{Star regexp} \\ & \mid & \textsf{Neg regexp} \end{array}$$

Chset represents a *charset*: a 256-wide bitset capable of representing any subset of $\Sigma$. We have chosen to represent bitsets with four-tuples of 64-bit numbers. Cat is the concatenation operator, Star is Kleene star, the Or operator is an $n$-ary disjunction, and Neg is a complement operator.

The semantics of regular expressions inteprets a regexp as a set of strings:

$$\begin{array}{rcl} \mathcal{L}(\textsf{Chset}\ cset) & = & \{[c] \mid c \in cset\} \\ \mathcal{L}(\textsf{Cat}\ r_1\ r_2) & = & \mathcal{L}(r_1) \cdot \mathcal{L}(r_2) \\ \mathcal{L}(\textsf{Or}\ [r_1, \ldots, r_n]) & = & \mathcal{L}(r_1) \cup \cdots \cup \mathcal{L}(r_n) \\ \mathcal{L}(\textsf{Star}\ r) & = & (\mathcal{L}(r))^* \\ \mathcal{L}(\textsf{Neg}\ r) & = & \textsf{COMPL}(\mathcal{L}(r)) \end{array}$$

where concatenation $(- \cdot -)$, Kleene star $(-)^*$, and set complement $\textsf{COMPL}(-)$ are pre-defined operators in a theory including sets and formal languages. It is also useful to have a function charset_of : $\mathbb{N}$ list $\rightarrow$ *cset* that creates a charset given a list of numbers less than 256.

Some common notation and derived operations in this representation:

$$\begin{array}{rcl} r_1 + \cdots + r_n & = & \mathsf{Or}[r_1, \ldots, r_n] \\ r_1 r_2 & = & \mathsf{Cat}\ r_1\ r_2 \\ [m-n] & = & \mathsf{Chset}(\mathsf{charset\_of}\ [m, m+1, \ldots, n-1, n]) \quad (\text{where } 0 \leq m \leq n \leq 255) \\ [m] & = & [m-m] \\ \bullet & = & [0-255] \\ \emptyset & = & \mathsf{Chset}(\mathsf{charset\_of}\ []) \\ \varepsilon & = & \mathsf{Star}(\emptyset) \\ \mathsf{And}\ r_1\ r_2 & = & \mathsf{Neg}(\mathsf{Or}[\mathsf{Neg}\ r_1, \mathsf{Neg}\ r_2] \\ \mathsf{Diff}\ r_1\ r_2 & = & \mathsf{And}\ r_1\ (\mathsf{Neg}\ r_2) \end{array}$$

The "dot" charset ($\bullet$) is the full charset that matches every one-element string; $\emptyset$ matches no string; and $\varepsilon$ is the empty string.

## Properties meet Encodings

Now we come to the main focus of this paper: statements of the form

$$x \in P \iff \mathsf{encode}\ x \in \mathcal{L}(\mathsf{regexp\_of}\ P)$$

where $P : \tau \to \mathsf{bool}$ represents a predicate on type $\tau$, $\mathsf{encode} : \tau \to \mathbb{S}$ maps items of type $\tau$ to strings, and $\mathsf{regexp\_of}\ P$ is an abstract operation that translates $P$ into an "equivalent" regular expression. Rephrasing the statement as:

$$P : \tau \to \mathsf{bool} = \mathcal{L}(\mathsf{regexp\_of}\ P) \circ \mathsf{encode}$$

more directly expresses the **SPLAT** setting, namely, "what useful properties can be expressed when we stick an encoder in front of some formal language machinery?" The main practical benefit for us comes from being able to compile the $\mathsf{regexp\_of}\ (P)$ expression to a DFA, thus giving an efficient, table-driven, implementation of $P$.

**Remark.** The $\mathsf{regexp\_of}$ operation depends on context. In some cases, *e.g.* intervals as below, there is a syntax-driven algorithm; in some cases it is a matter of human insight. Further, whether $\mathsf{regexp\_of}$ is *shallowly* or *deeply* embedded is an important consideration, one that we gloss over for now.

**Remark.** $\mathsf{regexp\_of}$ is currently limited to predicates expressible with regular languages. Moving up to grammars, such as context-free grammars, gives more expressive power, and would require a $\mathsf{grammar\_of}$ operator (along with a modified definition of $\mathcal{L}(-)$) in the statement, *i.e.*:

$$x \in P \iff \mathsf{encode}\ x \in \mathcal{L}(\mathsf{grammar\_of}\ P)$$

**Question.** To what extent can formula constructions (conjunction, disjunction, implication, *etc.*) over the predicates be paralleled by regular expression constructions on the right? As a start, we can prove the following theorems:

$$\begin{array}{rcl}
\neg P(x) & \iff & \mathsf{encode}(x) \notin \mathcal{L}(\mathsf{regexp\_of}\ P) \\
P(x) \wedge Q(x) & \iff & \mathsf{encode}(x) \in \mathcal{L}(\mathsf{And}\ (\mathsf{regexp\_of}\ P)\ (\mathsf{regexp\_of}\ Q)) \\
P(x) \vee Q(x) & \iff & \mathsf{encode}(x) \in \mathcal{L}(\mathsf{Or}\ [\mathsf{regexp\_of}\ P, \mathsf{regexp\_of}\ Q]) \\
P(x) \Rightarrow Q(x) & \iff & \mathsf{encode}(x) \in \mathcal{L}(\mathsf{Or}\ [\mathsf{Neg}(\mathsf{regexp\_of}\ P), \mathsf{regexp\_of}\ Q]) \\
P(x) \wedge Q(y) & \iff & \mathsf{encode}(x)\mathsf{encode}(y) \in \mathcal{L}(\mathsf{Cat}\ (\mathsf{regexp\_of}\ P)\ (\mathsf{regexp\_of}\ Q))
\end{array}$$

Further work is needed to see whether, *e.g.*, Presburger formulas could be encoded. (It is entirely plausible that this has already been reported in the vast literature on formal languages.)

**Example**

Consider the "even-number" predicate $\mathsf{even} : \mathbb{N} \to \mathsf{bool}$. Let $cset = \mathsf{charset\_of}\ [0, 2, 4, \ldots, 254]$ be the character set corresponding to the non-negative even numbers less than 256. Assuming

$$\mathsf{regexp\_of\ even} = \mathsf{Cat}\ (\mathsf{Chset}\ cset)\ (\mathsf{Star}\ \bullet),$$

(the encoding is in LSB format), we construct the formula

$$\mathsf{even}(n) \iff \mathsf{encode}(n) \in \mathcal{L}(\mathsf{Cat}\ (\mathsf{Chset}\ cset)\ (\mathsf{Star}\ \bullet))$$

This can be proven by reasoning with the regular language semantics, plus the encodings defined below. The regular expression compiles to a 3-state DFA.

## Encodings for common types

We now discuss a collection of common types and their encoding and decoding functions. There is of course a wide variety of encoding schemes. Our approach is to use binary encodings, reducing higher level types to a base encoder/decoder for natural numbers.

- **Natural numbers.** First, define the map $\mathsf{n2l} : \mathbb{N} \to \mathbb{N}\ \mathsf{list}$ from a number to its little-endian base-256 representation, and the inverse.

  $$\begin{array}{lcl}
  \mathsf{n2l}(n) & = & \mathsf{if}\ n = 0\ \mathsf{then}\ []\ \mathsf{else}\ (n\ \mathsf{mod}\ 256)\ ::\ \mathsf{n2l}\ (n\ \mathsf{div}\ 256) \\
  \mathsf{l2n}\ [] & = & 0 \\
  \mathsf{l2n}\ (h\ ::\ t) & = & h + 256 * \mathsf{l2n}(t)
  \end{array}$$

  Then we have $\vdash \forall n.\ \mathsf{l2n}(\mathsf{n2l}\ n) = n$. Big-endian versions are similarly defined. Another issue is *padding* to a specified width. Using a builtin operator, $\mathsf{pad\_right}$, we can define a basic number encoder $\mathsf{enc} : \mathbb{N} \to \mathbb{N} \to \mathbb{S}$ and the corresponding decoder $\mathsf{dec} : \mathbb{S} \to \mathbb{N}$:

  $$\begin{array}{rcl}
  \mathsf{layout}\ n\ width & = & \mathsf{pad\_right}\ 0\ width\ (\mathsf{n2l}\ n) \\
  \mathsf{enc}\ w\ n & = & \mathsf{map\ chr}\ (\mathsf{layout}\ n\ w) \\
  \mathsf{dec}\ s & = & \mathsf{l2n}\ (\mathsf{map\ ord}\ s)
  \end{array}$$

Then we have $\vdash \forall n\ w.\ \mathsf{dec}(\mathsf{enc}\ w\ n) = n$.

- **Integers.** Encoding of integers can be expressed by adding an encoding to map from integers to natural numbers. Twos complement representation is most commonly used. At the word level, the twos complement operation on negative integers is "flip each bit and add 1". But it is expressible at the level of integers and natural numbers since an $(n-1)$-bit integer added to its twos complement equals $2^n$. Thus

  $$\mathsf{i2n}_N : \{i \in \mathbb{Z} \mid -2^{N-1} \leq i < 2^{N-1}\} \to \{k \in \mathbb{N} \mid 0 \leq k < 2^N\}$$

  specifies a map from an integer to its corresponding natural number defined by

  $$\mathsf{i2n}_N(i) = \mathsf{if}\ 0 \leq i < 2^{N-1}\ \mathsf{then}\ \mathsf{Nat}(i)\ \mathsf{else}\ 2^N - \mathsf{Nat}(\mathsf{Abs}(i))$$

  ($\mathsf{Nat}$ maps from non-negative integers to $\mathbb{N}$; $\mathsf{Int}$ maps from $\mathbb{N}$ to non-negative integers; and $\mathsf{Abs}$ is an absolute value operator that maps integers to non-negative integers.) The inverse function is

  $$\mathsf{n2i}_N(n) = \mathsf{if}\ n < 2^{N-1}\ \mathsf{then}\ \mathsf{Int}(n)\ \mathsf{else}\ -\mathsf{Int}(2^N - n)$$

  An encoder and decoder for integers is then directly obtained:

  $$\begin{array}{l}\mathsf{enci}\ w\ i = \mathsf{enc}\ w\ (\mathsf{i2n}\ (8w)\ i)\\ \mathsf{deci}\ w\ s = \mathsf{n2i}\ (8w)\ (\mathsf{dec}\ s)\end{array}$$

  which enjoy the inversion property, provided integer $i$ is representable:

  $$\vdash 0 < w \wedge -\mathsf{Int}(2^{8w-1}) \leq i \wedge i < \mathsf{Int}(2^{8w-1}) \ \Rightarrow\ \mathsf{deci}\ w\ (\mathsf{enci}\ w\ i) = i$$

  **Note.** $\mathsf{enci}(i)$ renders $i$ into twos complement representation, thus supporting the standard bit format for signed integers.

- **Enumerations.** An enumerated type can be encoded by providing a map from elements of the type to $\mathbb{N}$. For example the type of booleans give rise to maps

  $$\begin{array}{ll}\mathsf{num\_of\_bool\ false} = 0 & \mathsf{num\_of\_bool\ true} = 1\\ \mathsf{bool\_of\_num}\ 0 = \mathsf{false} & \mathsf{bool\_of\_num}\ 1 = \mathsf{true}\end{array}$$

  which gives the basis for the encoder and decoder:

  $$\begin{array}{l}\mathsf{enc\_bool}\ b = \mathsf{enc}\ 1\ (\mathsf{num\_of\_bool}\ b)\\ \mathsf{dec\_bool}\ s = \mathsf{bool\_of\_num}\ (\mathsf{dec}\ s)\end{array}$$

  Trivially, we have $\vdash \forall b.\ \mathsf{dec\_bool}(\mathsf{enc\_bool}\ b) = b$.

- **Records.** Sets of records of a given type $\mathsf{recd}$ can be encoded by fixing an order on the fields and concatenating their encodings. Thus given record type

  $$\mathsf{recd} = \{f_1 : \tau_1; \ldots ; f_n : \tau_n\}$$

  and encoders for $\tau_1, \ldots, \tau_n$, we can define

  $$\mathsf{encode\_recd}(x) = \mathsf{encode}_{\tau_1}(x.f_1) \cdots \mathsf{encode}_{\tau_n}(x.f_n)$$

- **Arrays.** A $\tau$ array of size $K$ can be encoded by concatenating the result of applying the encoder for $\tau$ $K$ times. While this is conceptually simple, the resulting regular expressions for large or even medium size arrays will probably be too large for our tools.
- **Fixed- and variable-width encodings.** So far we have dealt solely with data of fixed size. Techniques for encoding variable-width data, such as lists or trees, and for handling polymorphic types, have been explored in a variety of settings. For theorem proving, focusing on polytypism, see "Applications of polytypism in theorem proving" (Slind and Hurd, 2003) and similar work in other proof systems.

## Interval Properties and Regexps

Now that we have covered an approach to data encoding, we discuss the encoding of data properties by regular expressions. The $\mathsf{even}(n)$ predicate from our initial example was created in an *ad hoc* manner, but we seek some properties that can be algorithmically translated to regexps. One of the most useful for us so far has been *intervals*.

An interval $[lo, hi]$ is defined to be $\{n \in \mathbb{N} \mid lo \leq n \leq hi\}$; another way of expressing it is as a property $\lambda n. lo \leq n \wedge n \leq hi$. An interval is mapped to a regular expression by an algorithm $\mathsf{regexp\_of}$ $[lo, hi]$ that proceeds in parallel on the base-256 representations given by $\mathsf{reverse} \circ \mathsf{n2l}$, *i.e*, following the MSB (most significant byte leftmost) format. However, the algorithm on the representations is quite detailed, and better insight is obtained by describing the transformations at the level of numbers. Let $\mathbf{B} = 256$ so that

$$\begin{array}{lcl} lo = p\mathbf{B}^a + r_1 & ; & r_1 < \mathbf{B}^a \\ hi = q\mathbf{B}^b + r_2 & ; & r_2 < \mathbf{B}^b \end{array}$$

The algorithm breaks large intervals down to unions of sub-intervals in such a way that regular expressions can be easily generated from the subintervals. Proceeding lexicographically on exponents, then factors, then remainders, there are essentially four ways for $lo \leq hi$ to hold:

1. $a < b$. The span between the exponents is split into one interval per intermediate exponent

$$[lo, \mathbf{B}^{a+1} - 1] \cup \underbrace{[\mathbf{B}^{a+1}, \mathbf{B}^{a+2} - 1] \cup \cdots \cup [\mathbf{B}^{b-1}, \mathbf{B}^b - 1]}_{\text{exponent slices}} \cup [\mathbf{B}^b, hi]$$

An *exponent slice* interval $[\mathbf{B}^k, \mathbf{B}^{k+1} - 1]$ is represented by the regexp $[1 - 255]\bullet^k$. This leaves the endpoint intervals $[lo, \mathbf{B}^{a+1} - 1]$ and $[\mathbf{B}^b, hi]$, each with a uniform exponent, which are dealt with by the next case.

2. $a = b, p < q$. The span between factors can also lead to splits:

$$[p\mathbf{B}^a + r_1, (p+1)\mathbf{B}^a - 1] \cup \underbrace{[(p+1)\mathbf{B}^a, q\mathbf{B}^a - 1]}_{\text{factor slice}} \cup [q\mathbf{B}^a, q\mathbf{B}^a + r_2]$$

   The *factor slice* interval $[(p+1)\mathbf{B}^k, q\mathbf{B}^k - 1]$ is only defined when $p+1 < q$ and is represented by the charset $[(p+1) - (q-1)]$ followed by $k$ dots:

$$[(p+1) - (q-1)] \bullet^k .$$

   This leaves the endpoint intervals $[p\mathbf{B}^a + r_1, (p+1)\mathbf{B}^a - 1]$ and $[q\mathbf{B}^a, q\mathbf{B}^a + r_2]$, each with a uniform factor, which takes us to the next case.

3. $a = b, p = q, r_1 \leq r_2$. In this case, the algorithm recurses on $[r_1, r_2]$, yielding regexps $s_1, \ldots, s_k$, and we return $[p]s_1 + \cdots + [p]s_k$.

4. The recursion bottoms out when *lo* and *hi* are less than 256, in which case the charset regexp $[lo - hi]$ is returned.

**Integer Intervals.** Translation of integer intervals is accomplished by reduction to natural number intervals, based on the following case split (number of bytes is $w$):

1. $0 \leq lo \leq hi$. No translation needed; the interval is regexp_of $[lo, hi]$.

2. $lo < 0 \leq hi$. A disjunction is made around 0.

$$\mathsf{regexp\_of}\ [lo, hi] = \mathsf{regexp\_of}\ [2^{8w} + lo, 2^{8w} - 1] + \mathsf{regexp\_of}\ [0, hi]$$

   There is a special case here, where $lo = -2^{8w-1}$ and $hi = 2^{8w-1} - 1$, in which case regexp_of $[lo, hi] = \bullet^w$.

3. $lo \leq hi < 0$. Both numbers are negative; we generate the interval from their twos complements.

$$\mathsf{regexp\_of}\ [lo, hi] = \mathsf{regexp\_of}\ [2^{8w} + lo, 2^{8w} + hi]$$

**Examples**

In the following the implementation prints regexps out in a format similar to the one used in this document. A charset may be expressed as a sequence of ranges $[(m_1 - n_1), ..., (m_k - n_k)]$, where $m$ and $n$ are written as decimal ASCII codes or as printable characters. Also, disjunction is expressed with $(- \mid -)$. The following are in MSB format.

The interval $[-90, 90]$ generates the regexp

```
[\166-\255] | [\000-Z]
```

This disjunction is on single charsets, so can be replaced by (and is, in the translation to DFA):

```
[\000-Z\166-\255]
```

The interval $[-180, 180]$ is recognized by the regexp

```
[\255][L-\255] | [\000][\000-\180]
```

The interval $[0, 14000]$ maps to

```
[\000]. | [\001-5]. | [6][\000-\176]
```

A 5-state DFA is generated. The interval $[-2^{63}, 2^{63}-1]$ maps to

```
........
```

A 10-state machine is generated. The interval $[123456789, 9876543210]$ generates the regexp:

```
[\000][\007][[][\205][\021-\255] |
[\000][\007][[][\206-\255]. |
[\000][\007][\-\255].. |
[\000][\008-\255]... |
[\001].... |
[\002][\000]... |
[\002][\001-K]... |
[\002][L][\000].. |
[\002][L][\001-\175].. |
[\002][L][\176][\000]. |
[\002][L][\176][\001-\021]. |
[\002][L][\176][\022][\000-\234]
```

which translates to a DFA with 15 states.

## Record properties

Given record type

$$\mathsf{recd} = \{f_1 : \tau_1; \ldots; f_n : \tau_n\}$$

and property $P : \mathsf{recd} \to \mathsf{bool}$ defined with field properties $P_1, \ldots, P_n$ by

$$P(r) \Leftrightarrow P_1(r.f_1) \wedge \ldots \wedge P_n(r.f_n),$$

we can build $\mathsf{regexp\_of}\ (P)$ as the concatenation

$$\mathsf{regexp\_of}\ (P_1) \cdots \mathsf{regexp\_of}\ (P_n)$$

In our examples so far, the field properties have been intervals and subsets of enumerations.

## A record message package

The modelling of record properties via regexps, as just outlined, gives us a useful level of abstraction for a package that takes high level specifications of records and generates encoder/decoder/filter implementations plus associated correctness proofs.

In particular, given a declaration of record type $\mathsf{recd}$ plus a predicate $P : \mathsf{recd} \to \mathsf{bool}$ over the fields of the record, we generate the following artifacts:

- a regexp $\mathcal{R} = \mathsf{regexp\_of}\ P$
- an encoding function $\mathsf{encode} : \mathsf{recd} \to \mathbb{S}$
- a decoding function $\mathsf{decode} : \mathbb{S} \to \mathsf{recd\ option}$
- an inversion theorem showing that decoding inverts encoding for well-formed records

  $$\vdash \forall r : \mathsf{recd}.\ P(r) \ \Rightarrow\ \mathsf{decode}(\mathsf{encode}\ r) = \mathsf{SOME}\ r$$

- a correctness theorem showing that encodings of well-formed records are exactly in the language of $\mathcal{R}$.

  $$\vdash \forall r : \mathsf{recd}.\ P(r) \Leftrightarrow \mathsf{encode}(r) \in \mathcal{L}(\mathcal{R})$$

- a deterministic finite state automaton (DFA) $M$ with the property that

  $$\vdash \forall s : \mathbb{S}.\ s \in \mathcal{L}(\mathcal{R}) \Leftrightarrow \mathsf{exec}(M, s) = \mathsf{true}$$

- a theorem stating that DFA $M$ enforces property $P$:

  $$\vdash \forall r : \mathsf{recd}.\ P(r) \Leftrightarrow \mathsf{exec}(M, \mathsf{encode}(r)) = \mathsf{true}$$

- CakeML programs and proofs (Johannes)

In summary, given the record type and a wellformedness predicate, the package produces implementations for an encoder and decoder, a DFA that checks wellformedness of encoded records, along with correctness proofs.

### Property discussion

A variety of theorems can be proved in this setting, and it is worth discussing how they can be applied. For example, invertibility of encoding

$$\vdash \forall r : \mathsf{recd}.\ P(r) \ \Rightarrow\ \mathsf{decode}(\mathsf{encode}\ r) = \mathsf{SOME}\ r$$

is dependent on the well-formedness of the record. What happens when the record is not well-formed? Proving the "iff" form of this might be better. Similarly, one might want the property that successfully decoding a string into a record implies that the record is well-formed. One version of this is

$$\mathsf{decode}(s) = \mathsf{SOME}\ r \ \Rightarrow\ P(r)$$

Another version:

$$\mathsf{decode}(s) = \mathsf{SOME}\ r \ \Rightarrow\ s \in \mathcal{L}(\mathsf{regexp\_of}\ P)$$

Both of these fail to exclude an incorrect encoder that (say) maps all records into one (well-formed) value. To forbid that, one could demand that the encoder is injective:

$$\forall \mathit{recd}_1\ \mathit{recd}_2.\ \mathsf{encode}(\mathit{recd}_1) = \mathsf{encode}(\mathit{recd}_2)\ \Rightarrow\ \mathit{recd}_1 = \mathit{recd}_2$$

Regarding applicability, another consideration is that, typically, encoders and decoders are written by different people at different times and places, trying to conform to a standard. Thus statements about a generated encoding/decoding pair are of limited utility, so separate properties about the encoder and decoder are valuable, whence the importance of the well-formedness specification.

## Proofs and Automation

## Property-enhanced lexing

Lexing provides an interesting application of **SPLAT** notions. A typical lexer specification has the form $(r_1, f_1), \ldots, (r_n, f_n)$ where an $(r, f)$ pair binds a regular expression $r$ with an *action* function $f : \mathbb{S} \to \tau$. From this specification a lexer of type $\mathbb{S} \to \tau\ \mathsf{list}$ can be automatically created. By attaching properties to regular expressions one can imagine a lexer specification augmented with post-conditions. In particular, an element $(\mathsf{regexp\_of}\ P, f, Q)$ of such a lexer spec comprises regular expression $\mathsf{regexp\_of}\ P$, action function $f$, and post-condition $Q$. The following theorem can be automatically proved for $\mathsf{regexp\_of}\ P$:

$$\vdash s \in \mathcal{L}(\mathsf{regexp\_of}\ P)\ \Rightarrow\ P(\mathsf{the}(\mathsf{decode}\ s))$$

(where $\mathsf{the}$ projects the wrapped element from an option datum.) The theorem can be used in the following way: when the lexer matches rule $(\mathsf{regexp\_of}\ P, f, Q)$ on substring $s$ of the input, a normal lexer would invoke $f$ on $s$ to produce a lexeme. Instead, we have the opportunity to invoke $f$ on the decoded substring, $(\mathsf{the}(\mathsf{decode}\ s))$, knowing that the result is in $P$. This gives the Hoare-style formula

$$\forall s.\ P\,(\mathsf{the}(\mathsf{decode}\ s))\ \Rightarrow\ Q\ (f\ (\mathsf{the}(\mathsf{decode}\ s)))$$

to be proved, which can be chained with the original theorem to yield

$$\vdash \forall s.\ s \in \mathcal{L}(\mathsf{regexp\_of}\ P)\ \Rightarrow\ Q\ (f\ (\mathsf{the}(\mathsf{decode}\ s)))$$

These theorems can be proved at lexer specification time, leaving only the last to be applied at runtime. Thus a successful run of an augmented lexer will yield a list of lexemes $[\ell_1, \ldots, \ell_n]$ and a list of theorems $[\vdash Q_1(\ell_1), \ldots, \vdash Q_n(\ell_n)]$.

In lexers for programming languages, this does not seem to offer much improvement in verification, but for lexing message formats, as sketched above, this may allow an improved verification and code synthesis experience.

## Further Issues

### Packed Record Formats

### Dependent Records

## Related Work

Buechi, Mona, Narcissus, D'Antoni, Slind and Hurd (encoding/decoding). Sail, PADS, message format specification languages. G. Rosu and generation of run-time monitors from regexps (any property proofs there?)

Note: unlike, say, Narcissus, we do not focus solely on encoding and decoding. We add verified compilation, and proof of well-formednes with respect to arithmetic specifications.

Much work in auto-synthesis of encoder/decoder pairs from high-level specs, e.g., ASN.1 and Google Protocol Buffers.

The details about encoding higher data in regexps are probably not novel, given the huge amount of work expended on transition system encodings for model-checking and in applying BDD and SAT algorithms. However, I haven't yet come across any literature focusing on intervals.

Check BDD encodings of numbers, etc.