Symbolic String Transformations with Regular Lookahead and Rollback

Margus Veanes(✉)

Microsoft Research, Redmond, USA
margus@microsoft.com

Abstract. Implementing string transformation routines, such as encoders, decoders, and sanitizers, correctly and efficiently is a difficult and error prone task. Such routines are often used in security critical settings, process large amounts of data, and must work efficiently and correctly. We introduce a new declarative language called Bex that builds on elements of regular expressions, symbolic automata and transducers, and enables a compilation scheme into C, C# or JavaScript that avoids many of the potential sources of errors that arise when such routines are implemented directly. The approach allows correctness analysis using symbolic automata theory that is not possible at the level of the generated code. Moreover, the case studies show that the generated code consistently outperforms hand-optimized code.

1 Introduction

Recent focus on string analysis is motivated by the fact that strings play a central role in all aspects of web programming. As soon as you visit a web page or read a file, several encoders, decoders and sanitizers launch for different purposes. Some coders are related to data integrity and format, such as UTF8 encoding and decoding that translates between standard text file representation (UTF8) and standard runtime memory representation (UTF16) of Unicode characters. Other encoders, called sanitizers, are used to prevent cross-site scripting (XSS) attacks; typical examples are Html encoder and Css encoder. While for such coders, basic functional correctness criteria is often vital for security, it may be notoriously difficult to implement them correctly or even reason about such correctness criteria [3,13]. One reason behind this difficulty is the subtle semantics resulting from a combinaton of *arithmetic* with *automata theory.* Individual characters are represented by integers and operations over characters often involve arithmetic operations such as bit-shifting and modulo arithmetic. Automata theory, on the other hand, is used overs strings (sequences of characters) to check for possible input or output patterns that may cause security vulnerabilities. Encoding related security vulnerabilities have been exploited for example through over-encoding [18,20], double-encoding [19], and XSS attacks. Some recent work has studied sanitizer correctness by utilizing automata techniques [6,16,17], including Bek [13] that our current work builds on.

Here we introduce a language called *Bex.* The main features of Bex that make it more expressive and succinct than Bek are: (1) *regex lookahead* for pattern

© Springer-Verlag Berlin Heidelberg 2015
A. Voronkov and I. Virbitskaite (Eds.): PSI 2014, LNCS 8974, pp. 335–350, 2015.
DOI: 10.1007/978-3-662-46823-4_27

matching that removes the burden of having to explicitly encode state machines; (2) *default rules* to specify what happens when a normal rule fails. In contrast, Bek supports only single-character guards and construction of default rules is then trivial by using the disjunction of all the negated guards from a given state as the guard of the default rule from that state.

Example 1. Consider the following Bex program `B`. `B` decodes two-digit html decimal encodings. The first rule, with pattern P_0 = `"&#00;"`, states that the null character must not be decoded. The second rule, with pattern P_1 = `"&#[0--9]{2};"`, is the normal decoding case. The third rule, with pattern P_2 = `"&#$"`, uses the end-anchor \$ so it applies only if the match occurs at the end of the input. The fourth rule is a default rule, it applies only when no other rule applies and it always reads a single character while a normal rules read k characters at a time with k being the length of the matched input.

```
program B { "&#00;"    ==> "&#00;";
            "&#\d\d;"  ==> [(10*(x2-48))+(x3-48)];
            "&#$"      ==> "&#";
            else       ==> [x0]; }
```

Consider the input u = `"&&#00;&#38;&#"`. No pattern matches initially, both P_0 and P_1 match from position 1, P_1 matches from position 6, and P_2 matches from position 11. For the overlapping case, P_i has priority over P_j for $i < j$. So

$$\mathtt{B}(u) = \texttt{"\&"} + \texttt{"\&\#00;"} + [(10{*}(\texttt{`3'}{-}48)){+}(\texttt{`8'}{-}48)] + \texttt{"\&\#"} = \texttt{"\&\&\#00;\&\&\#"}$$

where the ASCII character codes are `‘&’` $= 38$, `‘3’` $= 51$ and `‘8’` $= 56$. ⊠

Bek programs were originally compiled into *symbolic finite transducers* or SFTs [13]. Unlike sanitizers, a direct representation of *decoders* with SFTs is highly impractical due to state space explosion [24]. Even when registers are added to Bek and symbolic transducers with *registers* (STs) are being used, direct representation with Bek and STs is still very cumbersome and error prone, as illustrated by the representation of `HTMLdecode` (corresponding to `B`) in [24, Fig. 7]. The need to read several characters at once without storing them in registers and without introducing intermediate states, motivated the introduction of *extended* symbolic finite transducers (ESFTs) [8], that add support for *lookahead*. However, unlike in the classical case where lookahead can effectively be eliminated [26, Theorem 2.17], analysis of ESFTs does not reduce to analysis of SFTs and requires, for decidability, further restriction to the *Cartesian* case [7] where *guards* are conjunctions of unary predicates. Regexes such as P_1 in Example 1 naturally give rise to Cartesian guards, e.g., P_1 represents the guard $\lambda\bar{x}.(x_0 = \texttt{`\&'} \wedge x_1 = \texttt{`\#'} \wedge \texttt{`0'} \leq x_2 \leq \texttt{`9'} \wedge \texttt{`0'} \leq x_3 \leq \texttt{`9'} \wedge x_4 = \texttt{`;'})$. The guard is Cartesian because it has the form $\lambda\bar{x}.\bigwedge_{i=1}^{|\bar{x}|}\varphi_i(x_i)$.

Cartesian ESFTs are still a powerful extension of SFTs because *outputs* may depend on multiple variables and use nonunary functions. For example, the second rule of `B` in Example 1 has the output function $\lambda\bar{x}.[10 * (x_2 - 48) + x_3 - 48]$.

The main difficulty with Bex is how to efficiently deal with default rules. A naive implementation of the semantics of bex, e.g., by using a regex matching library, is far too inefficient. For example, the full version of HtmlDecode requires 280 rules. One approach would be to eliminate default rules by adding more normal rules in an attempt to transform Bex programs to ESFTs. For example, we could add the rule `"&[^#&]" ==> ['&',x1]` to cover the case when the matched subsequence starts with `&` but is not followed by `#` or `&`. Continuing this transformation quickly leads to an explosion of cases and requires intermediate states, obfuscating the semantics and defeating the purpose of the concise declarative style of Bex.

Instead, we provide here a novel compilation scheme from Bex programs to an intermediate form called *symbolic rollback transducers* SRTs that are subsequently compiled into STs. SRTs use *lookback* to avoid state space explosion. For example, an SRT may treat the pattern `"&#\d{6};"` of an html decoder using nine transitions rather than *100 k* transitions required by an SFT; once it successfully matches the pattern it refers back to the characters in the matched input, similar to k-SLTs [5]. SRTs incorporate the notion of *rollback* in form of *rollback*-transitions not present in STs [24], ESFTs [8] or k-SLTs [5], to accommodate default or exceptional behavior.

To summarize, this paper makes the following contributions:

- *Bex*: a new declarative language for specifying string coders;
- *SRTs*: a variant of ESFTs with the capability of rewinding the input tape;
- Algorithm for compiling bex programs into SRTs.

As a key component the algorithm makes use of the recent algorithm for minimizing SFAs [9].

2 Symbolic Automata

In this section we introduce the basic concepts of symbolic automata that we are using in this paper. A key role is played by symbolic representation of alphabets as effective Boolean algebras. An *effective Boolean algebra* $\mathcal{A}$ has components $(\mathfrak{D}, \varPsi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$. $\mathfrak{D}$ is a nonempty r.e. (recursively enumerable) set of *domain elements.* $\varPsi$ is an r.e. set of *predicates* closed under the Boolean connectives and $\bot, \top \in \varPsi$. The *denotation function* $[\![\_]\!] : \varPsi \to 2^{\mathfrak{D}}$ is r.e. and is such that, $[\![\bot]\!] = \emptyset$, $[\![\top]\!] = \mathfrak{D}$, for all $\varphi, \psi \in \varPsi$, $[\![\varphi \vee \psi]\!] = [\![\varphi]\!] \cup [\![\psi]\!]$, $[\![\varphi \wedge \psi]\!] = [\![\varphi]\!] \cap [\![\psi]\!]$, and $[\![\neg\varphi]\!] = \mathfrak{D} \setminus [\![\varphi]\!]$. For $\varphi \in \varPsi$, we write $\mathit{IsSat}(\varphi)$ when $[\![\varphi]\!] \neq \emptyset$ and say that φ is *satisfiable.* The intuition is that $\mathcal{A}$ is represented programmatically as an API with corresponding methods implementing the components. We use the following symbolic alphabets.

$\mathbf{2}^k$ is the powerset algebra with domain $\{n \mid 0 \leq n < 2^k\}$. Case $k = 0$ is trivial and is denoted $\mathbf{1}$. For $k > 0$, a predicate in $\varPsi_{\mathbf{2}^k}$ is a BDD of depth k.[1]

[1] The variable order of the BDD is the reverse bit order of the binary representation of a number, thus, the most significant bit has the lowest ordinal.

The Boolean operations are the BDD operations. The denotation $[\![\beta]\!]$ is the set of all n, $0 \le n < 2^k$, whose binary representation is a solution of β.

$\mathcal{U}$ We let $\mathcal{U} \stackrel{\text{def}}{=} \mathbf{2}^{16}$ denote the basic *Unicode* alphabet. We use standard regex character class notation to describe predicates in $\Psi_{\mathcal{U}}$. For example $[\![\texttt{A}]\!] = [\![\texttt{\textbackslash x41}]\!] = \{65\}$, $[\![\texttt{01}]\!] = \{48, 49\}$, and $[\![\texttt{[\textbackslash 0-\textbackslash xFF]}]\!] = \{n \mid 0 \le n \le 255\}$.

We use the following construct for alphabet extensions. Given a domain D we write D' for an injective renaming of all elements in D, $D' = \{a' \mid a \in D\}$. Similarly for D''. One concrete definition of D' is $D \times \{1\}$ and of D'' is $D \times \{2\}$. In particular, $D' \cap D'' = \emptyset$.

Definition 1. The *disjoint union* $\mathcal{A}+\mathcal{B}$ of two effective Boolean algebras $\mathcal{A}$ and $\mathcal{B}$, is the effective Boolean algebra $(\mathfrak{D}'_{\mathcal{A}} \cup \mathfrak{D}''_{\mathcal{B}}, \Psi_{\mathcal{A}} \times \Psi_{\mathcal{B}}, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$ where,

$$[\![\langle \alpha, \beta \rangle]\!] \stackrel{\text{def}}{=} [\![\alpha]\!]'_{\mathcal{A}} \cup [\![\beta]\!]''_{\mathcal{B}},\ \langle \alpha, \beta \rangle \diamond \langle \alpha_1, \beta_1 \rangle \stackrel{\text{def}}{=} \langle \alpha \diamond_{\mathcal{A}} \alpha_1, \beta \diamond_{\mathcal{B}} \beta_1 \rangle,\ (\diamond \in \{\vee, \wedge\})$$
$$\neg\langle \alpha, \beta \rangle \stackrel{\text{def}}{=} \langle \neg_{\mathcal{A}} \alpha, \neg_{\mathcal{B}} \beta \rangle, \quad \bot \stackrel{\text{def}}{=} \langle \bot_{\mathcal{A}}, \bot_{\mathcal{B}} \rangle,\ \top \stackrel{\text{def}}{=} \langle \top_{\mathcal{A}}, \top_{\mathcal{B}} \rangle.$$

It is straightforward to prove that $\mathcal{A}+\mathcal{B}$ is still an effective Boolean algebra. Observe that the implementation of $\mathcal{A}+\mathcal{B}$ is trivial given the implementations of $\mathcal{A}$ and $\mathcal{B}$, e.g., extension of $\mathcal{A}$ with a new element can be defined as $\mathcal{A}+\mathbf{1}$.

Given a word $u \in \mathfrak{D}^*_{\mathcal{A}}$ we write u' for the word $[u|'_0, u|'_1, \ldots, u|'_{|u|-1}]$ in $\mathfrak{D}^*_{\mathcal{A}+\mathcal{B}}$. Similarly for the second subdomain.

Definition 2. A *symbolic finite automaton* (*SFA*) M is a tuple $(\mathcal{A}, Q, q^0, F, \Delta)$ where $\mathcal{A}$ is an effective Boolean algebra, called the *alphabet*, Q is a finite set of *states*, $q^0 \in Q$ is the *initial state*, $F \subseteq Q$ is the set of *final states*, and $\Delta \subseteq Q \times \Psi_{\mathcal{A}} \times Q$ is a finite set of *moves* or *transitions*. Elements of $\mathfrak{D}_{\mathcal{A}}$ are called *characters* and finite sequences of characters are called *words*. ⊠

A word w of length $|w|$, is denoted by $[a_0, a_1, \ldots, a_{|w|-1}]$ where all the a_i are characters. Given a position $i < |w|$, $w|_i$ denotes the i'th character a_i of w. The empty word is $[]$. Given two words u and v, $u.v$ denotes their concatenation. In particular, $u.[] = [].u = u$.

A move $\rho = (p, \varphi, q)$ *from* p *to* q is also denoted by $p \xrightarrow{\varphi} q$ where p is the *source* state $Src(\rho)$, q is the *target* state $Tgt(\rho)$, and φ is the *guard* or *predicate* of the move $Grd(\rho)$. A move is *feasible* if its guard is satisfiable. In the following let $M = (\mathcal{A}, Q, q^0, F, \Delta)$ be a fixed SFA.

Definition 3. A word $w \in \mathfrak{D}^*_{\mathcal{A}}$, is *accepted at state* q *of* M, $w \in \mathscr{L}_q(M)$, if there exists a set of moves $\{q_i \xrightarrow{\varphi_i} q_{i+1}\}_{i<k} \subseteq \Delta$ where $k = |w|$, $q_0 = q$, $q_k \in F$, and $w|_i \in [\![\varphi_i]\!]$ for $i < k$. The *language of* M is $\mathscr{L}(M) \stackrel{\text{def}}{=} \mathscr{L}_{q^0}(M)$.

For $q \in Q$, we use the definitions

$$\Delta(q) \stackrel{\text{def}}{=} \{\rho \in \Delta \mid Src(\rho) = q\}, \quad \Delta^{-1}(q) \stackrel{\text{def}}{=} \{\rho \in \Delta \mid Tgt(\rho) = q\}.$$

A state q of M is a *deadend* when $\mathscr{L}_q(M) = \emptyset$. A *deadend-move* is a move whose target is a deadend. A state q of M is *partial* if $\neg \bigvee \{Grd(\rho) \mid \rho \in \Delta(q)\}$ is satisfiable. A move is *feasible* if the guard of the move is satisfiable. The following terminology is used to characterize various subclasses of SFAs.

- M is *deterministic*: for all $p \xrightarrow{\varphi} q, p \xrightarrow{\varphi'} q' \in \Delta$, if $\mathit{IsSat}(\varphi \wedge \varphi')$ then $q = q'$.
- M is *partial* (*incomplete*): there is a partial state.
- M is *clean*: all moves are feasible and all states are reachable from q^0,
- M is *trim*: M is clean and has no deadend-moves,
- M is *normalized*: forall $p, q \in Q$, there is at most one move from p to q.
- M is *minimal*: M is deterministic, trim, and normalized, and forall $p, q \in Q$, $p = q$ if and only if $\mathscr{L}_p(M) = \mathscr{L}_q(M)$.
- M is a *prefix acceptor* if M is minimal, M has a single final state q^{f}_M and either $\Delta_M(q^{\mathrm{f}}_M) = \emptyset$, or $\Delta_M(q^{\mathrm{f}}_M) = \{q^{\mathrm{f}}_M \xrightarrow{\top} q^{\mathrm{f}}_M\}$, and all paths from q^0_M to q^{f}_M without passing through q^{f}_M have a fixed length K, the *length of* M.

Regexes used here range over the Unicode alphabet $\mathcal{U}$ and support character classes, the syntax and the semantics is the same as in C# or JavaScript.[2] Given a regex P we write $\hat{\ }P$ for P prepended with the start anchor. We write $\mathscr{L}(P)$ for the regular language over $\mathfrak{D}_{\mathcal{U}}$ accepted by P. Given a regular language L, we write $\mathit{SFA}_{\min}(L)$ for a minimal SFA accepting L.

Anonymous functions. We write $\Lambda(D \to R)$ for some well-defined effective representation of functions, or λ-*terms*, with domain D and range R. A λ-term $f \in \Lambda(D \to R)$ denotes the mathematical function $\boldsymbol{f} : D \to R$.

Let the alphabet $\mathcal{A}$ be fixed and let $\mathfrak{D}$ stand for $\mathfrak{D}_{\mathcal{A}}$ and let Ψ stand for $\Psi_{\mathcal{A}}$. We let $\mathfrak{D}^k \overset{\text{def}}{=} \{w \in \mathfrak{D}^* \mid |w| = k\}$.[3] We write Λ for $\bigcup_{m>0,n\geq 0} \Lambda(\mathfrak{D}^m \to \mathfrak{D}^n)$, i.e., Λ is the set of λ-terms denoting functions from *nonempty* fixed length words to fixed length words (the range may be $\{[]\}$). Given $f \in \Lambda$, let $\natural(f)$ denote the *input rank* m of $f \in \Lambda(\mathfrak{D}^m \to \mathfrak{D}^n)$.

Example 2. Consider $\mathcal{A} = \mathcal{U}$. Let $h \in \Lambda(\mathfrak{D} \to \mathfrak{D})$ be $\lambda x.(x < 10\ ?\ x{+}48 : x{+}55)$. Then $\boldsymbol{h}$ encodes every nibble (value in $\{0, \ldots, 15\}$) as the corresponding hexadecimal (ASCII) digit,[4] e.g., $\boldsymbol{h}(11) =$ `'B'` and $\boldsymbol{h}(7) =$ `'7'`. Let $f \in \Lambda(\mathfrak{D}^1 \to \mathfrak{D}^2)$ be $\lambda x.[h(x|_0 \gg 4), h(x|_0 \,\&\, 15)]$ ($\gg$ is *shift-right* and & is *bitwise-and*). Then $\boldsymbol{f}$ encodes every single-byte-word as a word of two hexadecimal digits, e.g., $\boldsymbol{f}($`"K"`$) = \boldsymbol{f}([4\mathrm{B}_{16}]) = [\boldsymbol{h}(4\mathrm{B}_{16} \gg 4), \boldsymbol{h}(4\mathrm{B}_{16} \,\&\, 15)] = [$`'4'`, `'B'`$] =$ `"4B"`. ⊠

3 Symbolic Rollback Transducers

Symbolic transducers (STs) are a generalization of symbolic *finite* transducers or SFTs; STs were originally introduced in [24]. An ST may use *registers* in addition to a finite set Q of states. In general, registers can hold arbitrary values and the use of registers is unrestricted. Here we introduce another extension of

[2] Regular Expression Language - Quick Reference: http://msdn.microsoft.com/en-us/library/az24scfc.aspx.

[3] Observe that $\mathfrak{D}^0 = \{[]\}$ and $\mathfrak{D}^1 = \{[a] \mid a \in \mathfrak{D}\}$.

[4] No semantic distinction is made between characters and their numeric codes. Thus `'0'`, `'\x30'`, and `48` all denote number 48.

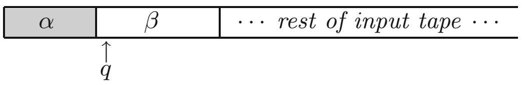


Fig. 1. Intuition behind a snapshot $\langle\alpha, q, \beta\rangle$ of an SRT.

SFTs called SRTs that do not allow explicit use of registers but allow *lookback* and *rollback* of input. SRTs have *three* kinds of transitions, defined below.

To formally define the semantics of transitions we introduce the notion of a *snapshot* $\mathfrak{s}$, that is a triple $\langle\alpha, q, \beta\rangle \in \mathfrak{S} = \mathfrak{D}^* \times Q \times \mathfrak{D}^*$ with *argument store* α, *state* q and *buffer* β. We say *current character* for the first character of the buffer if it is nonempty, else for the first character in the rest of the input. The unread portion of the input tape is not part of the snapshot. The idea behind the concept of a snapshot is illustrated in Fig. 1. The buffer is intended to be a *prepending* to the rest of the input; the semantics enforces that the buffer must be empty before any more characters are read from the rest of the input.

An *input-transition* $p \xrightarrow{\varphi} q \in Q \times \Psi \times Q$ has the following semantics. From source state p it reads and enqueues the current character a into the argument store, provided that $a \in [\![\varphi]\!]$, and enters the target state q, formally:

$$\begin{aligned}[\![p \xrightarrow{\varphi} q]\!] \stackrel{\text{def}}{=} & \{\langle\alpha, p, []\rangle \xrightarrow{[a]/[]} \langle\alpha.[a], q, []\rangle \mid a \in [\![\varphi]\!],\ \alpha \in \mathfrak{D}^*\} \cup \\ & \{\langle\alpha, p, [a].\beta\rangle \xrightarrow{[]/[]} \langle\alpha.[a], q, \beta\rangle \mid a \in [\![\varphi]\!],\ \alpha, \beta \in \mathfrak{D}^*\}\end{aligned}$$

An *output-transition* $p \overset{f}{\mapsto} q \in Q \times \Lambda \times Q$ has the following semantics. From state p it consumes the argument store α outputs the word $\boldsymbol{f}(\alpha)$ and enters state q. The transition is enabled when the length of α matches the arity of f.

$$[\![p \overset{f}{\mapsto} q]\!] \stackrel{\text{def}}{=} \{\langle\alpha, p, \beta\rangle \xrightarrow{[]/\boldsymbol{f}(\alpha)} \langle[], q, \beta\rangle \mid \beta \in \mathfrak{D}^*,\ \alpha \in \mathfrak{D}^{\natural(f)}\}$$

A *rollback-transition* $p \overset{\varphi}{\dashrightarrow} q$ has the following semantics. From state p, if the current character $a \in [\![\varphi]\!]$, it "rewinds the input tape" by pushing the current character and the argument store (back) into the buffer, and enters state q.

$$\begin{aligned}[\![p \overset{\varphi}{\dashrightarrow} q]\!] \stackrel{\text{def}}{=} & \{\langle\alpha, p, []\rangle \xrightarrow{[a]/[]} \langle[], q, \alpha.[a]\rangle \mid a \in [\![\varphi]\!],\ \alpha \in \mathfrak{D}^*\} \cup \\ & \{\langle\alpha, p, [a].\beta\rangle \xrightarrow{[]/[]} \langle[], q, \alpha.[a].\beta\rangle \mid a \in [\![\varphi]\!],\ \alpha, \beta \in \mathfrak{D}^*\}\end{aligned}$$

The idea is that a rollback-transition is taken when a normal input sequence cannot be completed, the target state q is then an "exception handling" state.

Definition 4. A *Symbolic Rollback Transducer* (*SRT*) is a tuple $(\mathcal{A}, Q, q^0, F, \Delta)$, where $\mathcal{A}$, Q, q_0, and F are as in Definition 2, and Δ is a finite set of *transitions* as defined above. ⊠

The semantics of an SRT B is defined using a transducer $(\mathfrak{s}_0, \mathfrak{S}, \mathfrak{T})$ that is the unwinding of B, where $\mathfrak{s}_0$ is the initial snapshot $\langle[], q^0, []\rangle$ of B, $\mathfrak{S}$ is the set $\mathfrak{D}^* \times Q \times \mathfrak{D}^*$ and $\mathfrak{T} \subseteq \mathfrak{S} \times \mathfrak{D}^* \times \mathfrak{D}^* \times \mathfrak{S}$ is the set $\bigcup_{\rho\in\Delta}[\![\rho]\!]$.

The relation $\mathfrak{s} \xrightarrow{u/v}\!\!\!\!\!\rightarrow \mathfrak{t}$ for $\mathfrak{s}, \mathfrak{t} \in \mathfrak{S}$ and $u, v \in \mathfrak{D}^*$ is defined as the least relation such that $\mathfrak{s}_0 \xrightarrow{[]/[]}\!\!\!\!\!\rightarrow \mathfrak{s}_0$ and if $\mathfrak{s} \xrightarrow{u/v}\!\!\!\!\!\rightarrow \mathfrak{s}_1$ and $\mathfrak{s}_1 \xrightarrow{u_1/v_1} \mathfrak{t} \in \mathfrak{T}$ then $\mathfrak{s} \xrightarrow{u.u_1/v.v_1}\!\!\!\!\!\rightarrow \mathfrak{t}$. The *transduction* of B is now defined as the following function from $\mathfrak{D}^*$ to $2^{\mathfrak{D}^*}$.

$$\mathscr{T}_B(u) \stackrel{\text{def}}{=} \{v \mid \exists q \in F\, \langle [], q^0, [] \rangle \xrightarrow{u/v}\!\!\!\!\!\rightarrow \langle [], q, [] \rangle\}$$

As a minimal requirement, we want the transition relation $\mathfrak{T}$ to be *well-founded* in the following sense: there is no infinite chain $\{\mathfrak{s}_i \xrightarrow{[]/v_i} \mathfrak{s}_{i+1}\}_{i<\omega}$ in $\mathfrak{T}$. For example, if there is a rollback-transition $p \overset{\top}{\dashrightarrow} p$ then $\mathfrak{T}$ is not well-founded, because $\langle [], p, [a] \rangle \xrightarrow{[]/[]} \langle [], p, [a] \rangle \in \mathfrak{T}$. A sufficient condition to ensure well-foundedness of $\mathfrak{T}$ is that the SRT is not *ill-defined*:

Definition 5. An SRT is *ill-defined* if there exists a path of states $(q_i)_{i\leq n}$ and states p_1 and p_2 such that, $p_1 \dashrightarrow q_0$, (for $0 \leq i < n$) $q_i \rightarrow q_{i+1}$, and $q_n \dashrightarrow p_2$. The SRT is *well-defined* otherwise. ⊠

In a well-defined SRT, any two rollback-transitions must be separated by at least one output-transition. For example, if $p \overset{\top}{\dashrightarrow} p$ then the SRT is ill-defined. An *output-state* is a state that has an outgoing output-transition.

Definition 6. An SRT B is *deterministic* if every output-state has exactly one outgoing transition and for every other state q, all transitions from q have mutually disjoint guards. B is *single-valued* if, for all u, $|\mathscr{T}_B(u)| \leq 1$. ⊠

Proposition 1. *Every deterministic SRT is single-valued.*

Proof. Determinism implies that for any snapshot and current character there can be at most one resulting snapshot. Thus, for any given $u \in \mathfrak{D}^*$, there can be at most one path $\{\mathfrak{s}_i \xrightarrow{u_i/v_i} \mathfrak{s}_{i+1}\}_{i<n}$ such that $u = u_0.u_1.\cdots.u_{n-1}$. Thus, either $\mathscr{T}_B(u) = \emptyset$ or $\mathscr{T}_B(u) = \{v_0.v_1.\cdots.v_{n-1}\}$. ⊠

We treat a deterministic SRT B as a partial function and we write $B(u) = v$ for $\mathscr{T}_B(u) = \{v\}$.

Example 3. Let f be defined as in Example 2. Let B be the SRT

$$(\mathcal{U}, \{q_0, q_1\}, q_0, \{q_0\}, \{q_0 \xrightarrow{\texttt{[\textbackslash 0-\textbackslash xFF]}} q_1, q_1 \overset{f}{\mapsto} q_0)\}$$

Since there are no rollback-transitions the buffer is never used. We have

$$\langle [], q_0, [] \rangle \xrightarrow{\texttt{"o"}/[]} \langle \texttt{"o"}, q_1, [] \rangle \xrightarrow{[]/\texttt{"6F"}} \langle [], q_0, [] \rangle \xrightarrow{\texttt{"k"}/[]} \langle \texttt{"k"}, q_1, [] \rangle \xrightarrow{[]/\texttt{"6B"}} \langle [], q_0, [] \rangle$$

Thus $\langle [], q_0, [] \rangle \xrightarrow{\texttt{"ok"}/\texttt{"6F6B"}}\!\!\!\!\!\rightarrow \langle [], q_0, [] \rangle$, so $B(\texttt{"ok"}) = \texttt{"6F6B"}$. ⊠

End anchors. Given an alphabet $\mathcal{A}$, in order to detect the end of the input string over $\mathfrak{D}_{\mathcal{A}}$, we can lift $\mathcal{A}$ to $\mathcal{A}+\mathbf{1}$ and lift all $u \in \mathfrak{D}^*_{\mathcal{A}}$ to $u'.[0''] \in \mathfrak{D}^*_{\mathcal{A}+\mathbf{1}}$ where the character $0'' \in \mathfrak{D}_{\mathcal{A}+\mathbf{1}}$ is used only as the last input character. Such end-of-input character can then be used to trigger a final output-transition that empties the store (when the store is nonempty).

4 Bex

The alphabet is fixed to $\mathcal{U}$ here, $\mathfrak{D}$ stands for $\mathfrak{D}_{\mathcal{U}}$. A bex program consist of a nonempty sequence of *pattern rules* $(P_\imath \Longrightarrow f_\imath)_{0\leq\imath<k}$ and a *default output* f_{d}, where all $P_\imath$ are regexes, called *patterns*, and all $f_\imath$ and f_{d} are *output expressions* such that the following well-formdness criteria hold.

– $\mathit{SFA}_{\min}(\mathscr{L}(\hat{}P_\imath))$ is a prefix acceptor of some length $K_\imath > 0$.
– $f_\imath \in \Lambda$ and $\natural(f_\imath) = K_\imath$.
– f_{d} is undefined or $f_{\mathrm{d}} \in \Lambda$ and $\natural(f_{\mathrm{d}}) = 1$.

The first well-formdness condition ensures that all patterns have fixed lengths. The second condition ensures that the output functions are in scope: depend only on the characters matched by the pattern. The third condition ensures that the default output function only depends on one character (the current one).

The formal semantics of bex programs is as follows. The intent is to support straightforward specification of how typical encoders and decoders work in practice. Given a word u and indices i and j, $0 \leq i \leq j < |u|$, we write $u[i..]$ for the suffix $[u|_i, \ldots, u|_{|u|-1}]$ and $u[i..j]$ for the subsequence $[u|_i, \ldots, u|_j]$ of u.

Definition 7. Given a bex program $B = ((P_\imath \Longrightarrow f_\imath)_{0\leq\imath<k}, f_{\mathrm{d}})$. The *denotation of* B, $\boldsymbol{B}$, is a (partial) function from $\mathfrak{D}_{\mathcal{U}}^*$ to $\mathfrak{D}_{\mathcal{U}}^*$. Let $u \in \mathfrak{D}_{\mathcal{U}}^*$ be the input sequence. Let $n := 0$ and $v := []$. Let $M_\imath = \mathit{SFA}_{\min}(\mathscr{L}(\hat{}P_\imath))$ and let $K_\imath$ be the length of $M_\imath$. Repeat the following until $n = |u|$:

1. Let $I = \{\imath \mid u[n..] \in \mathscr{L}(M_\imath)\}$.
2. If $I \neq \emptyset$ let $\imath = \min\{i \in I \mid K_i = \min\{K_j \mid j \in I\}\}$ and $(m, f) = (K_\imath, f_\imath)$
3. If $I = \emptyset$ let $(m, f) = (1, f_{\mathrm{d}})$.
4. Let $v := v + f(u[n..n + m - 1])$ and $n := n + m$.

Then $\boldsymbol{B}(u) = v$. ($\boldsymbol{B}(u)$ is undefined if f_{d} is used but is undefined). ⊠

Example 1 is a simplified version of an Html decoder. Its purpose is to illustrate the use and the semantics of typical pattern rules and the default rule. It is used as a running example in the rest of the paper. In the next section we describe an algorithm that converts a bex program into an equivalent SRT.

5 Bex to SRT Compiler

The purpose of the bex to SRT compiler is, given a well-formed bex program $B = ((P_\imath \Longrightarrow f_\imath)_{0\leq i<k}, f_{\mathrm{d}})$ as input, to generate a well-defined deterministic SRT that is equivalent to B. We assume that the default output f_{d} is defined. The case when f_{d} is undefined amounts to a trivial modification of the compiler.

The compiler works in two main phases. First, all the patterns of the rules are combined into a single pattern automaton N that is then minimized. The alphabet of N is $\mathcal{U}2 \stackrel{\text{def}}{=} \mathcal{U}+\mathcal{U}$. The first subuniverse $\mathcal{U}'$ serves the purpose of the Unicode alphabet, while the second subuniverse $\mathcal{U}''$ serves the purpose of bex rule identifiers.

Second, the pattern automaton N is (essentially) extended with output-transitions and rollback-transitions to form the final SRT. It follows from minimality of N and the construction of the additional transitions that the resulting SRT is well-defined and deterministic and preserves the semantics of the original bex program.

The alphabet of the generated SRT is going to be $\mathcal{U} + \mathbf{1}$. The new element $0'' \in \mathfrak{D}_{\mathcal{U}+\mathbf{1}}$ is used as the end-of-input symbol of words. Observe that $\mathfrak{D}_{\mathcal{U}+\mathbf{1}} = \mathfrak{D}'_{\mathcal{U}} \cup \{0''\}$. The main correctness theorem is the following.

Theorem 1. *Given a bex program B, $SRT(B)$ is a well-defined deterministic SRT such that, for all $u, v \in \mathfrak{D}^*_{\mathcal{U}}$, $B(u) = v$ iff $\mathscr{T}_{SRT(B)}(u'.[0'']) = \{v'.[0'']\}$.*

Proof. Formal proof is by induction over the length of computations, relating the points in Definition 7 to the constructs below and by using basic properties of

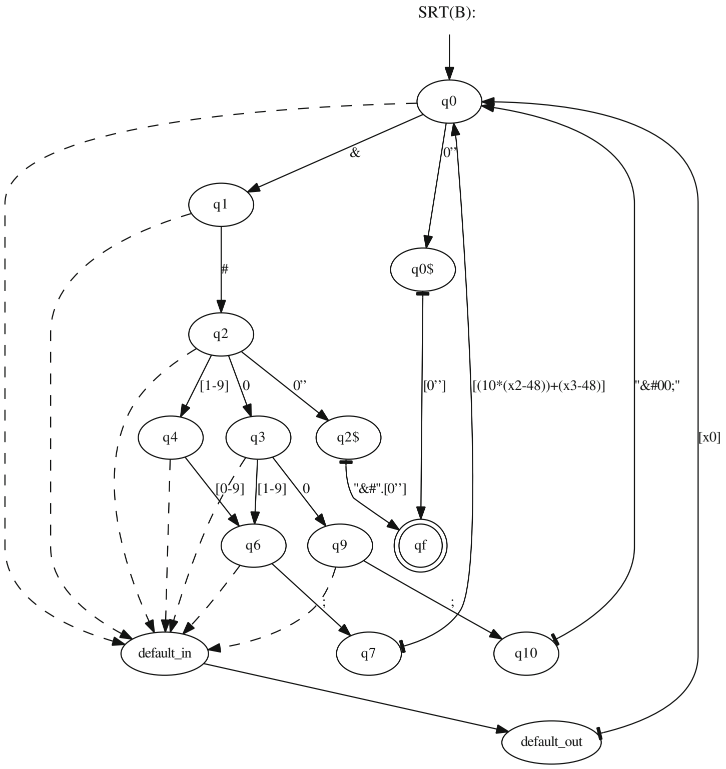


Fig. 2. Sample SRT with rollback-transitions.

N and SFAs operations. The construction of the SFA N itself uses an algorithm for minimizing SFAs [9]. ⊠

Detailed descriptions of the compilation phases are given below. The following example illustrates a small but realistic example.

Example 4. Consider the bex program B in Example 1. Figure 2 shows the generated $SRT(B)$. The rollback-transitions have a guard (not shown) that is the complement of the disjunction of all the guards from all other transitions from the source state. E.g., $q_2 \overset{\langle\texttt{[\^{}0-9]},\bot\rangle}{\dashrightarrow} \mathrm{default}_{\mathrm{in}}$ and $q_1 \overset{\langle\texttt{[\^{}\#]},\top\rangle}{\dashrightarrow} \mathrm{default}_{\mathrm{in}}$ ⊠

5.1 Pattern Automaton Construction

1. Let $\mathcal{E} := \emptyset$; $\mathcal{E}$ is computed as the set of all pattern ids having *end anchors*.
2. For $\imath = 0, \ldots, k-1$:
   (a) Let $M_\imath = SFA_{\min}(\mathscr{L}(\hat{}P_\imath))$. (Recall that $M_\imath$ is a prefix acceptor.)
   Let $K_\imath$ be the length of $M_\imath$.
   If $\Delta_{M_\imath}(q^{\mathrm{f}}_{M_\imath}) = \emptyset$ then $\mathcal{E} := \mathcal{E} \cup \{\imath\}$.
   (b) Let $q^\imath$ be a new state not it $Q_{M_\imath}$. Lift $M_\imath$ into $N_\imath$:

$$N_\imath = (\mathcal{U}2, Q_{M_\imath} \cup \{q^\imath\}, q^0_{M_\imath}, \{q^\imath\}, \Delta),$$
$$\text{where } \Delta = \{p \xrightarrow{\langle\varphi,\bot\rangle} q \mid p \xrightarrow{\varphi} q \in \Delta_{M_\imath}, p \neq q^{\mathrm{f}}_{M_\imath}\} \cup \{q^{\mathrm{f}}_{M_\imath} \xrightarrow{\langle\bot,\hat{\imath}\rangle} q^\imath\}$$

3. Let
$$N := SFA_{\min}(\bigcup_\imath \mathscr{L}(N_\imath)).$$
   N has a single final state, say $F_N = \{q^{\mathrm{f}}_N\}$, and $\Delta_N(q^{\mathrm{f}}_N) = \emptyset$. A move $p \xrightarrow{\langle\bot,\beta\rangle} q^{\mathrm{f}}_N$ is a *final move*; let $\imath = \min[\![\beta]\!]$, the state p is $\imath$*-final.*
4. *Cleanup*:
   (a) If a state p is $\imath$-final but $\imath \notin \mathcal{E}$ then delete all non-final moves from p.
   (b) Remove unreachable states from N.

Cleanup removes unreachable cases: shorter patterns override longer ones (for the overlapping cases) and for patterns of the same length the ones with smaller id have priority (see Definition 7.2). The following are key properties of N.

Proposition 2. *For all $w \in \mathfrak{D}^*_{\mathcal{U}2}$ the following statements are equivalent:*

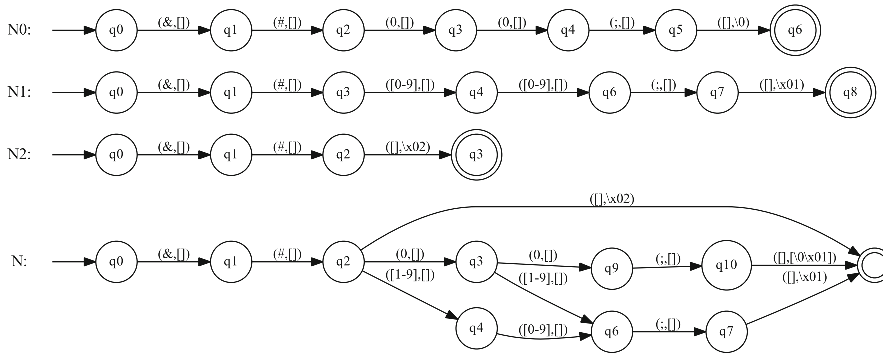
- *$w \in \mathscr{L}(N)$*
- *for some $u \in \mathfrak{D}^*_{\mathcal{U}}$ and $\imath \in \mathfrak{D}_{\mathcal{U}}$, $w = u'.[\imath'']$ and $w \in \mathscr{L}(N_\imath)$*
- *for some $u \in \mathfrak{D}^*_{\mathcal{U}}$ and $\imath \in \mathfrak{D}_{\mathcal{U}}$, $w = u'.[\imath'']$ and $u \in \mathscr{L}(M_\imath)$ and $|u| = K_\imath$*

Proposition 3. *If $q^0_N \overset{v}{\twoheadrightarrow} q$ and $q \xrightarrow{\langle\bot,\psi\rangle} q^f_N \in \Delta_N$ then for all $\imath \in [\![\psi]\!]$, $|v| = K_\imath$.*

Proof. Fix $\imath, \jmath \in [\![\psi]\!]$. Then $v{+}\imath^{(2)}, v{+}\jmath^{(2)} \in \mathscr{L}(N)$. So, by Proposition 2, $v = u^{(1)}$ for some $u \in \mathfrak{D}^*_{\mathcal{U}}$ such that $u \in \mathscr{L}(M_\imath) \cap \mathscr{L}(M_\jmath)$ and $|u| = K_\imath = K_\jmath$. ⊠

The purpose of N is going to be that N is used to construct a control flow graph of the SRT. N takes care of selecting the correct rule for a given input.

Example 5. Consider the bex program `B` in Example 1. The SFAs N_0, N_1, N_2 and N are as follows:[5]



In N the overlapping patterns are reflected in the final move $q_{10} \xrightarrow{\langle \texttt{[]}, \texttt{[\textbackslash 0\textbackslash x01]} \rangle} q_N^{\mathrm{f}}$ where $[\![\texttt{[\textbackslash 0\textbackslash x01]}]\!] = \{0, 1\}$. $\mathcal{E} = \{2\}$. If the end anchor was removed from P_2 then $\mathcal{E}$ would be empty and the cleanup step would delete the moves from q_2 to q_3 and q_4. Then the states $\{q_3, q_4, q_6, q_7, q_9, q_{10}\}$ would become unreachable. ⊠

5.2 Compute Normal Transitions

We will now use N as a starting point for constructing an SRT $SRT(B)$ from B.

We lift functions f over the universe $\mathfrak{D}_{\mathcal{U}}$ implicitly to functions over the universe $\mathfrak{D}_{\mathcal{U}+\mathbf{1}}$ by lifting elements in $\mathfrak{D}_{\mathcal{U}}$ to elements in the first subuniverse $\mathfrak{D}'_{\mathcal{U}}$ of $\mathfrak{D}_{\mathcal{U}+\mathbf{1}}$. Let Δ^{in} be the following set of input-transitions.

$$\Delta^{\mathrm{in}} = \{p \xrightarrow{\langle \varphi, \bot \rangle} q \mid p \xrightarrow{\langle \varphi, \bot \rangle} q \in \Delta_N\}$$

In other words, all nonfinal moves of N become input-transitions. Let Δ^{out} be the following set of output-transitions, where $q^0 = q_N^0$,

$$\Delta^{\mathrm{out}} = \{p \overset{f_\imath}{\mapsto} q^0 \mid p \xrightarrow{\langle \bot, \beta \rangle} q_N^{\mathrm{f}} \in \Delta_N,\ \imath = \min[\![\beta]\!],\ \imath \notin \mathcal{E}\}$$

In other words, if a state p is $\imath$-final and the pattern $P_\imath$ is not a suffix pattern of the input ($\imath \notin \mathcal{E}$) then, upon reaching the state p, the input store contains a word s of length $K_\imath$ matching the pattern $P_\imath$. The output function $\boldsymbol{f}_\imath$ is applied to the matched word s committing to the output word $\boldsymbol{f}_\imath(s)$. The process is repeated from the initial state q^0.

[5] Predicates in $\Psi_{\mathcal{U}}$ are denoted by regex character classes, or individual characters. The predicate $\bot$ is denoted by the empty character class `[]`.

5.3 Compute Ending Transitions

When a regex pattern $P_\imath$ ends with an end anchor (`$` or `\z`) then this is reflected in N by the fact that there is a state p that is $\imath$-final and $\imath \in \mathcal{E}$. This means that the match must end with the end-of-input character $0''$ because all valid input words have the form $u'.[0'']$ for $u \in \mathfrak{D}_{\mathcal{U}}^*$. There are new output states $p_\$^\imath$ for all $\imath \in \mathcal{E}$, with the following input-transitions leading to them.

$$\Delta^{\mathrm{in}\$} = \{p \xrightarrow{\langle\bot,\top\rangle} p_\$^\imath \mid p \xrightarrow{\langle\bot,\beta\rangle} q_N^{\mathrm{f}} \in \Delta_N,\ \imath = \min[\![\beta]\!],\ \imath \in \mathcal{E}\}$$

There are output-transitions from each $p_\$^\imath$ that apply the corresponding final output function to the final stored input word in each case, append the ending character $0''$, so that all output words are also $0''$-terminated, and transition to the final state $q^{\mathrm{f}} = q_N^{\mathrm{f}}$ of the SRT.

$$\Delta^{\mathrm{out}\$} = \{p_\$^\imath \xmapsto{\lambda x.(f_\imath(x).[0''])} q^{\mathrm{f}} \mid p \xrightarrow{\langle\bot,\beta\rangle} q_N^{\mathrm{f}} \in \Delta_N,\ \imath = \min[\![\beta]\!],\ \imath \in \mathcal{E}\}$$

There are also transitions from the initial state $q^0 = q_N^0$ leading to the final state (upon end of input), where $q_\$^0$ is a new output state:

$$\Delta^0 = \{q^0 \xrightarrow{\langle\bot,\top\rangle} q_\$^0 \xmapsto{\lambda x.[0'']} q^{\mathrm{f}}\}$$

There are no transitions outgoing from the final state q^{f}.

5.4 Compute Default Transitions

The default behavior kicks in from a state p when the current character does not match any of the possible guards of the outgoing input-transitions from p. Formally, let $G(p)$ be the disjunction of all the guards from transitions exiting from p. Here p is an *input state* that is a non-output state and not q^{f}.

$$G(p) \stackrel{\mathrm{def}}{=} \bigvee\{\varphi \mid \exists q(p \xrightarrow{\varphi} q \in \Delta^{\mathrm{in}} \cup \Delta^{\mathrm{in}\$} \cup \Delta^0)\}, \quad \gamma_p \stackrel{\mathrm{def}}{=} \neg G(p).$$

Predicate γ_p describes all characters that break all possible patterns at state p. If γ_p is satisfiable then, for all current characters in $[\![\gamma_p]\!]$, roll back the input tape back to the position before the match was started, then apply the default function to the first character in the input tape (it cannot be $0''$ because the input store is nonempty when $p \neq q^0$ and $0'' \notin [\![\gamma_{q^0}]\!]$), and finally continue the process from state q^0 and the next input position. This corresponds to Definition 7.3. Formally, the following transitions are added to capture this default behavior.

$$\begin{aligned} \Delta^{\mathrm{default}} = \{&p \overset{\gamma_p}{\dashrightarrow} \mathrm{default}_{\mathrm{in}} \mid p \text{ is an input state}, [\![\gamma_p]\!] \neq \emptyset\} \\ &\cup\{\mathrm{default}_{\mathrm{in}} \xrightarrow{\top} \mathrm{default}_{\mathrm{out}} \xmapsto{f_{\mathrm{d}}} q^0\}\end{aligned}$$

where $\mathrm{default}_{\mathrm{in}}$ and $\mathrm{default}_{\mathrm{out}}$ are fixed new states. Observe that the well-definedness criterion (see Definition 5) is trivially satisfied. Let Q be the set of all states that occur in the transitions. The final result of the compilation is the SRT:

$$SRT(B) \stackrel{\text{def}}{=} (\mathcal{U}+\mathbf{1}, Q, q^0, \{q^{\mathrm{f}}\}, \varDelta^{\mathrm{in}} \cup \varDelta^{\mathrm{out}} \cup \varDelta^{\mathrm{in\$}} \cup \varDelta^{\mathrm{out\$}} \cup \varDelta^{0} \cup \varDelta^{\mathrm{default}})$$

Moreover, a well-defined SRT can be further translated into an equivalent ST without rollback-transitions by performing a symbolic partial evaluation of the default cases. Generation of C# or JavaScript code is straightforward from either well-defined deterministic SRTs or deterministic STs.

6 Implementation and Experiments

The bex language and the algorithm for generating symbolic transducers from bex programs has been implemented and is available in an online toolkit and tutorial [4]. The tutorial includes several samples, such as base64 encoding and decoding, allows online editing, and enables JavaScript generation from the bex programs. The generated JavaScript can also be directly executed online.

We have built a prototype implementation of the compiler. In a final phase the compiler converts the generated SRT into an ST without rollback-transitions. It does so by symbolically forward executing the rollback-cases and by optimizing the generated code through a combination of SFA techniques and SMTlib representation of terms using Z3 [10,27]. Z3 terms are used to simplify arithmetic expressions and to prune unsatisfiable predicates. The STs are then converted into either C# or JavaScript implementations.

We have applied this technique to a variety of different encoders and decoders such as: Utf16Encoder and Decoder, Base32Encoder and Decoder, Base64Encoder and Decoder, CssEncoder, JavaScriptEncoder, JsonEncoder, and HtmlEncoder and Decoder. They all fall into a category of string transformation routines that can be very naturally expressed and analyzed in bex.

So far our largest case study is a bex program for the complete version of *HtmlDecode* that uses over 280 rules. The full bex program is less than 300 lines of code including comments. The large number of rules is due to many special cases of patterns such as `"&lt;"` $\Longrightarrow$ `"<"` and `"&le;"` $\Longrightarrow$ `"\u2264"` in addition to rules that decode numeric (decimal or hexadecimal) encodings of characters. The resulting *minimal* pattern automaton N has in this case 920 states and the generated C# code is just shy of 20 k lines of code (with sparsely generated code). The end-to-end compilation time was around 8 s that includes preprocessing as well as some analysis of the generated code. A key factor here was an efficient minimization algorithm of SFAs [9]. The minimization algorithm is used repeatedly in the loop where the SFAs N_i are being constructed during the pattern automaton construction phase of the bex compiler. For the alphabet algebra we use $\mathcal{U}2$ for most parts, but for dealing with λ-terms and satisfiability checking of linear arithmetic formulas in the final phases of the compiler we use SMT2lib representation of terms and Z3 [27].

We compared the running time of the bex generated HtmlDecoder in C# against the *hand-optimized* HtmlDecoder in the. NET System.Net.WebUtility library. As input to both decoders we used maximally encoded input (with hexadecimal encoding of all non-ASCII) texts over various parts of the Unicode alphabet. In this experiment, the bex coder *outperformed* the System coder by 2 times on average.

7 Related Work

Symbolic finite transducers (SFTs) and the Bek language were originally introduced in [13] and formally studied in [24]. SFTs were also extended to STs in [24] to allow the use of registers for increased expressive power. A common usage pattern that often occurs in the context of string decoders is that of a finite window of characters that are grouped and processed together. For such a class of problems, SFTs are too weak, while STs sacrifice analyzability. Two related formalisms have been proposed to address this issue, ESFTs [8] and k-SLTs [5]. The former uses bounded *lookahead* and reads several characters at once, while the latter uses bounded *lookback* and reads one character at a time. Further properties of ESFTs are studied in [7].

The formalism of SRTs is in spirit related to k-SLTs, because output-transitions refer to earlier characters as a form of lookback. However, once an output happens (is "committed"), there is no way to refer back to those input characters that were used, in later transitions; this is similar to the sematic of ESFTs. The aspect that is new in SRTs, is the notion of a *rollback*-transition that allows the input tape to be rewound or rolled back conditionally. As we demonstrated with bex, this aspect greatly simplifies the task of programming typical encoders and decoders, HtmlDecoder being a perfect example, where default rules are used extensively when pattern matching fails.

Automata over infinite alphabets have received a lot of interest [21], starting with the work on *register automata* [14]. A different line of work on automata with infinite alphabets called *lattice automata*, originates from verification of symbolic communicating machines [11]. *Streaming transducers* [1] provide a recent symbolic extension of finite transducers. Extended Finite Automata, or XFAs, is a succinct representation of DFAs that use registers, are introduced in [22] for network packet inspection. XFAs support only finite alphabets. History-based finite automata [15] are another extension of DFAs that have been introduced for encoding regular expressions in the context of network intrusion detection systems. Finite state transducers have been used for dynamic and static analysis to validate sanitization functions in web applications [17,25].

Symbolic transductions can also be considered over infinite strings. For finite alphabets, a study of transformations of infinite strings is proposed in [2]. Yet a different extension is symbolic transductions over trees [23].

We use the SMT solver Z3 [10] for incrementally solving and simplifying constraints in the process of composing predicates that arise during bex compilation. Similar applications of SMT techniques have been introduced in the context of symbolic execution of programs by using path conditions [12].

References

1. Alur, R., Cerný, P.: Streaming transducers for algorithmic verification of single-pass list-processing programs. In: POPL 2011, pp. 599–610. ACM (2011)
2. Alur, R., Filiot, E., Trivedi, A.: Regular transformations of infinite strings. In: LICS, pp. 65–74. IEEE (2012)
3. Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kirda, E., Kruegel, C., Vigna, G.: Saner: composing static and dynamic analysis to validate sanitization in web applications. In: SP 2008, pp. 387–401. IEEE (2008)
4. Bex (2013). http://www.rise4fun.com/Bex/tutorial
5. Botincan, M., Babic, D.: Sigma*: symbolic learning of input-output specifications. In: POPL 2013, pp. 443–456. ACM (2013)
6. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise analysis of string. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 1–18. Springer, Heidelberg (2003)
7. D'Antoni, L., Veanes, M.: Equivalence of extended symbolic finite transducers. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 624–639. Springer, Heidelberg (2013)
8. D'Antoni, L., Veanes, M.: Static analysis of string encoders and decoders. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 209–228. Springer, Heidelberg (2013)
9. Dantoni, L., Veanes, M.: Minimization of symbolic automata. In: POPL 2014. ACM (2014)
10. de Moura, L., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
11. Le Gall, T., Jeannet, B.: Lattice automata: a representation for languages on infinite alphabets, and some applications to verification. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 52–68. Springer, Heidelberg (2007)
12. Godefroid, P.: Compositional dynamic test generation. In: POPL 2007, pp. 47–54(2007)
13. Hooimeijer, P., Livshits, B., Molnar, D., Saxena, P., Veanes, M.: Fast and precise sanitizer analysis with Bek. In: USENIX Security, August 2011
14. Kaminski, M., Francez, N.: Finite-memory automata. TCS **134**(2), 329–363 (1994)
15. Kumar, S., Chandrasekaran, B., Turner, J., Varghese, G.: Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In: ANCS 2007, pp. 155–164. ACM/IEEE (2007)
16. Livshits, B., Nori, A.V., Rajamani, S.K., Banerjee, A.: Merlin: specification inference for explicit information flow problems. In: PLDI 2009, pp. 75–86 (2009)
17. Minamide, Y.: Static approximation of dynamically generated web pages. In: WWW 2005, pp. 432–441 (2005)
18. NVD. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2008-2938
19. OWASP. https://www.owasp.org/index.php/Double_Encoding
20. SANS. http://www.sans.org/security-resources/malwarefaq/wnt-unicode.php
21. Segoufin, L.: Automata and logics for words and trees over an infinite alphabet. In: Ésik, Z. (ed.) CSL 2006. LNCS, vol. 4207, pp. 41–57. Springer, Heidelberg (2006)
22. Smith, R., Estan, C., Jha, S., Kong, S.: Deflating the big bang: fast and scalable deep packet inspection with extended finite automata. In: SIGCOMM 2008, pp. 207–218. ACM (2008)
23. Veanes, M., Bjørner, N.: Symbolic tree transducers. In: Clarke, E., Virbitskaite, I., Voronkov, A. (eds.) PSI 2011. LNCS, vol. 7162, pp. 377–393. Springer, Heidelberg (2012)

24. Veanes, M., Hooimeijer, P., Livshits, B., Molnar, D., Bjørner, N.: Symbolic finite state transducers: algorithms and applications. In: POPL 2012, pp. 137–150 (2012)
25. Wassermann, G., Yu, D., Chander, A., Dhurjati, D., Inamura, H., Su, Z.: Dynamic test input generation for web applications. In: ISSTA (2008)
26. Yu, S.: Regular languages. In: Rozenberg, G., Salomaa, A. (eds.) Handbook of Formal Languages, vol. 1, pp. 41–110. Springer, Heidelberg (1997)
27. Z3. http://research.microsoft.com/projects/z3