Static Analysis of String Encoders and Decoders

Loris D'Antoni[1] and Margus Veanes[2]

[1] University of Pennsylvania*
lorisdan@cis.upenn.edu

[2] Microsoft Research
margus@microsoft.com

Abstract. There has been significant interest in static analysis of programs that manipulate strings, in particular in the context of web security. Many types of security vulnerabilities are exposed through flaws in programs such as string encoders, decoders, and sanitizers. Recent work has focused on combining automata and satisfiability modulo theories techniques to address security issues in those programs. These techniques scale to larger alphabets such as Unicode, that is a de facto character encoding standard used in web software.

One approach has been to use character predicates to generalize finite state transducers. This technique has made it possible to perform precise analysis of a large class of typical sanitization routines. However, it has not been able to cope well with decoders, that often require to read more than one character at a time. In order to overcome this limitation we introduce a conservative generalization of Symbolic Finite Transducers (SFTs) called Extended Symbolic Finite Transducers (ESFTs) that incorporates the notion of a bounded lookahead. We demonstrate the advantage ESFTs on analyzing programs for which previous approaches did not scale.

In our evaluation we use a UTF-16 to UTF-8 translator (*utf8encoder*) and a UTF-8 to UTF-16 translator (*utf8decoder*). We show, among other properties, that *utf8encoder* and *utf8decoder* are functionally correct.

1 Introduction

There has been significant recent interest in decision procedures for solving string constraints. Much of this work has focused on designing domain specific decision procedures for string analysis that use state-of-the art constraint solvers in the backend [17, 4, 19, 20]. Many of the tools use automata based techniques, including JSA [5], and Bek [9]. A comprehensive comparison of various algorithmic design choices in this space is studied in [10]. The growing interest in string analysis has also started a discussion for developing standards for regular expressions modulo alphabet theories [3] to unify some of the notations and underlying theory in the tool support.

* This work was done during an internship at Microsoft Research and this research was partially supported by NSF Expeditions in Computing award CCF 1138996.

One reason for this focus is *security vulnerabilities* caused by strings. Some recent work has studied sanitizer correctness through static analysis based on automata theory [12, 5, 13], including the Bek project and symbolic transducers [9] that our work is based on. Here we extend the analysis of Bek to a richer, more expressive class of problems. In particular we consider string coders that require symbolic *lookahead*. Symbolic lookahead allows programs to read more than one symbol at a time. For example, in order to decode a (html encoded) string `"&#38;"` back to the string `"&"` a lookahead of two digits is needed. Concretely, in the paper we consider unicode encodings UTF-16 and UTF-8, that have emerged as the most commonly used character encodings. UTF-8 is used for representing Unicode text in text files and is perhaps the most widely accepted character encoding standard in the internet today. UTF-16 is used for in-memory representation of characters in modern programming and scripting languages. Transformations between these two encodings are ubiquitous.

Despite the wide adoption of these encodings, their analysis is difficult, and carefully crafted *invalid* UTF-8 sequences have been used to bypass security validations. Several attacks have been demonstrated [16] based on over-encoding the characters '`.`' and '`/`' in malformed URLs. For example, the invalid sequence $[\texttt{C0}_{16}, \texttt{AF}_{16}]$ (that decodes to '`/`') has been used to bypass a literal check in the Microsoft IIS server (in unpatched Windows 2000 SP1) to determine if a URL contains "`../../`" by encoding it as "`..%C0%AF../`". Similar vulnerability exists in Apache Tomcat ($\leq$ 6.0.18), where "`%C0%AE`" has been used for encoding '`.`' [14]. Further attacks use double-encoding [15]. We show how our new extension of symbolic transducers can make analysis of such coding routines possible.

Our analysis starts from a compilation from Bek programs to *symbolic transducers* (ST). In a symbolic transducer, transitions are annotated with logical *formulas* instead of specific characters, and the transducer takes the transition on any input character that satisfies the formula. A symbolic transducer is then transformed to a representation called *extended symbolic finite transducer* (ESFT), that uses lookahead to avoid state space explosion. For example, an ESFT may treat the pattern `"&#[0-9]{6};"` of an html decoder using a *single* transition rather than *100k* transitions required by an SFT (without lookahead). Our representation enables leveraging *satisfiability modulo theories (SMT) solvers*, tools that take a formula and attempt to find inputs satisfying that formula. These solvers have become robust in the last several years and are used to solve complicated formulas in a variety of contexts. At the same time, our representation allows leveraging automata theoretic methods to reason about strings of unbounded length, which is not possible via direct encoding to SMT formulas. One advantage of SMT solvers is that they work with formulas from any theory supported by the solver, while other previous approaches are specialized to specific types of inputs. This is a crucial feature for our algorithms and analysis, in particular we use a combination of theories, involving sequences, numbers, and records.

After the analysis, programs written in Bek can be compiled back to traditional languages such as JavaScript or C#. This ensures that the code analyzed

is functionally equivalent to the code which is actually deployed for sanitization, up to bugs in our compilation. Bek is available online [2].

This paper makes the following contributions:

- it introduces ESFTs as a new effective model for analysis of string coders;
- it presents an algorithm for STs register elimination that improves the efficiency and expressiveness of the previous state of the art technique based on exhaustive exploration;
- it proves *UTF8* encoder and decoder to be correct; and
- it uses realistic coding routines to show how ESFTs scale for big programs.

We first define ESFTs (Section 2) and STs with registers (Section 2.2). Secondly, we provide an algorithm to transform a subclass of STs into ESFTs (Section 3). We then describe UTF-8 encoders and decoders and their Bek implementation (Section 4). We use our technique to prove those coders correct. Finally, we show how our technique scales for bigger programs (Section 5).

2 Extended Symbolic Finite Transducers

We assume a *background structure* that has a recursively enumerable (r.e.) multi-typed carrier set or *background universe* $\mathcal{U}$, and is equipped with a language of function and relation symbols with fixed interpretations. We use τ, σ and γ to denote types, and we write $\mathcal{U}^\tau$ for the corresponding sub-universe of elements of type τ. As a convention, we abbreviate $\mathcal{U}^\sigma$ by Σ and $\mathcal{U}^\gamma$ by Γ, due to their frequent use. The Boolean type is $\mathbb{B}$, with $\mathcal{U}^\mathbb{B} = \{\mathfrak{t}, \mathfrak{f}\}$ and the integer type is $\mathbb{Z}$. Terms and formulas are defined by induction over the background language and are assumed to be well-typed. The type τ of a term t is indicated by $t : \tau$. Terms of type $\mathbb{B}$, or Boolean terms, are treated as formulas, i.e., no distinction is made between formulas and Boolean terms. A *k-tuple type* is a type $\mathbb{T}\langle\tau_0, \ldots, \tau_{k-1}\rangle$ where $k \geq 0$ and all τ_i are types. The 0-tuple type $\mathbb{T}\langle\rangle$ is assumed to be such that $\mathcal{U}^{\mathbb{T}\langle\rangle}$ is the singleton sub-universe $\{()\}$ and the 1-tuple type $\mathbb{T}\langle\tau\rangle \stackrel{\text{def}}{=} \tau$. If τ is a type and $k \geq 0$ then τ^k stands for the type $\mathbb{T}\langle\tau_0, \ldots, \tau_{k-1}\rangle$ of k-way Cartesian product where all $\tau_i = \tau$. For example, $\mathbb{Z}^2$ is $\mathbb{T}\langle\mathbb{Z}, \mathbb{Z}\rangle$. If t is a k-tuple ($k > 1$) then $\pi_i(t)$, also written $t[i]$, projects the i'th element of t for $0 \leq i < k$. The k-tuple constructor for $k > 1$ is simply $(t_0, \ldots, t_{k-1})$.

If τ is a type, then τ^* is the type over finite sequences of elements of type τ. We assume the standard accessors $\mathit{head} : \tau^* \to \tau$ and $\mathit{tail} : \tau^* \to \tau^*$ over sequences and the constructors $\mathit{cons} : \tau \times \tau^* \to \tau^*$ and $[] : \tau^*$. A term $\mathit{cons}(t_0, \mathit{cons}(t_1, \ldots, \mathit{cons}(t_{n-1}, [])))$ of sort τ^* is also denoted by $[t_0, t_1, \ldots, t_{n-1}]$ and is called an *explicit sequence of length n*. We use the following shorthands to access elements of a sequence $t : \tau^*$, $\mathit{tail}^0(t) \stackrel{\text{def}}{=} t$, $\mathit{tail}^{k+1}(t) \stackrel{\text{def}}{=} \mathit{tail}(\mathit{tail}^k(t))$, and for $k \geq 0$, $t[k] \stackrel{\text{def}}{=} \mathit{head}(\mathit{tail}^k(t))$. Given a set S, we write S^* for the *Kleene closure* of S. The justification behind overloading the $*$-operator both as a type annotator and Kleene closure operator is that, for any type τ, we assume $\mathcal{U}^{(\tau^*)} = (\mathcal{U}^\tau)^*$. In particular $\mathcal{U}^{(\sigma^*)} = \Sigma^*$ and $\mathcal{U}^{(\gamma^*)} = \Gamma^*$.

All elements in $\mathcal{U}$ are also used as constant terms. A term without free variables (such as a constant term) is *closed*. Closed terms t have standard Tarski semantics $[\![t]\!]$ over the background structure. Substitution of a variable $x:\tau$ in t by a term $u:\tau$ is denoted by $t[x/u]$.

A *λ-term* is an expression of the form $\lambda\bar{x}.t$, where $\bar{x}$ is a (possibly empty) tuple of distinct variables, and t is a term all of whose free variables occur in $\bar{x}$. It is sometimes technically convenient to view $\bar{x}$ as a single variable of the corresponding product (tuple) type.

To indicate the types, we say *$(\sigma \to \gamma)$-term* for a λ-term $\lambda x.t$ such that $x:\sigma$ and $t:\gamma$. A $(\sigma \to \gamma)$-term $f = \lambda x.t$ denotes the function $[\![f]\!]$ that maps $a \in \Sigma$ to $[\![t[x/a]]\!] \in \Gamma$. We use f, g, h to stand for λ-terms. We do not distinguish between the λ-term $\lambda().t$ and t.

A $(\sigma \to \mathbb{B})$-term is called a *σ-predicate*. We use φ and ψ for σ-predicates and, for $a \in \Sigma$, we write $a \in [\![\varphi]\!]$ for $[\![\varphi]\!](a) = \mathtt{t}$. Given a $(\sigma \to \gamma)$-term $f = (\lambda x.t)$ and a term $u:\sigma$, $f(u)$ stands for the term $t[x/u]$. A σ-predicate φ is *unsatisfiable* when $[\![\varphi]\!] = \emptyset$; φ is *satisfiable*, otherwise. A $(\sigma \to \gamma^*)$-term $f = \lambda x.[t_0, \ldots, t_{n-1}]$ is called a *$(\sigma \to \gamma)$-sequence* and $|f| \stackrel{\text{def}}{=} n$.

A *label theory* is given by an r.e. set Ψ of formulas that is closed under Boolean operations, substitution, equality and if-then-else terms. When talking about satisfiability of formulas, we assume implicit λ-closures. A label theory Ψ is *decidable* when satisfiability for $\varphi \in \Psi$, $\mathit{IsSat}(\varphi)$, is decidable.

Next, we describe an extension of finite state transducers through a symbolic representation of labels and by adding a lookahead component to the rules.

Definition 1. An *Extended Symbolic Finite Transducer (ESFT)* over $\sigma \to \gamma$ is a tuple $A = (Q, q^0, R)$,

- Q is a finite set of *states*;
- $q^0 \in Q$ is the *initial state*;
- R is a finite set of *rules*, $R = \Delta \cup F$, where
- Δ is a set of *transitions* $r = (p, \ell, \varphi, f, q)$, denoted $p \xrightarrow[\ell]{\varphi/f} q$, where
  - $p \in Q$ is the *start* state of r;
  - $\ell \geq 1$ is the *lookahead* of r;
  - φ, the *guard* of r, is a σ^ℓ-predicate;
  - f, the *output* of r, is a $(\sigma^\ell \to \gamma)$-sequence;
  - $q \in Q$ is the *end* state of r.
- F is a set of *final rules* $r = (p, \ell, \varphi, f)$, denoted $p \xrightarrow[\ell]{\varphi/f} \bullet$, with components as above and where ℓ is allowed to be 0.

The *lookahead* of A is the maximum of all lookaheads of rules in R.

We use the following abbreviated notation for rules, by omitting explicit λ's. We write

$$p \xrightarrow[\ell]{t/[u_0,\ldots,u_k]} q \quad \text{for} \quad p \xrightarrow[\ell]{\lambda\bar{x}.t/\lambda\bar{x}.[u_0,\ldots,u_k]} q,$$

where t and u_i are terms whose free variables are among $\bar{x} = (x_0, \ldots, x_{\ell-1})$. Final rules are a generalization of final states. A final rule with lookahead ℓ applies

only when the remaining input has *exactly* ℓ elements remaining as opposed to a transition with lookahead ℓ that applies when the remaining input has *at least* ℓ elements remaining.

The typical case of a final rule that corresponds to the classical final state p is $p \xrightarrow[0]{\mathtt{t}/[]} \bullet$, i.e., p accepts the empty input and produces no additional outputs. But there could be a non-empty output like in $p \xrightarrow[0]{\mathtt{t}/[\text{`\#'}]} \bullet$. There could also be a final rule with a non-zero lookahead. For example, suppose that characters are integers, the state is q, and there are two rules from q, a final rule: if there is a single input character remaining it is output "as is" $q \xrightarrow[1]{\mathtt{t}/[x_0]} \bullet$; and a transition, if there are at least two input characters, their sum is output $q \xrightarrow[2]{\mathtt{t}/[x_0+x_1]} q$. It is not possible to separate these two cases without introducing nondeterminism, if final rules with positive lookahead are not allowed. It is also not practical to lift the input type by adding a new "end of input" symbol as is done in the classical case. Such type lifting non trivially affects properties of the label theory and complicates use of specific theories over a given type such as standard linear arithmetic.

An ESFT with lookahead 1 and whose all final rules have lookahead 0 is an SFT [20]. In the sequel let $A = (Q, q^0, R)$, $R = \Delta \cup F$, be a fixed ESFT over $\sigma \to \gamma$. The semantics of rules in R is as follows:

$$[\![p \xrightarrow[\ell]{\varphi/f} q]\!] \stackrel{\text{def}}{=} \{p \xrightarrow{[a_0,\ldots,a_{\ell-1}]/[\![f]\!](a_0,\ldots,a_{\ell-1})} q \mid (a_0,\ldots,a_{\ell-1}) \in [\![\varphi]\!]\}$$

We write $s_1 \cdot s_2$ for concatenation of two sequences s_1 and s_2.

Definition 2. For $\boldsymbol{a} \in \Sigma^*, \boldsymbol{b} \in \Gamma^*, q \in Q, q' \in Q \cup \{\bullet\}$, define $q \xrightarrow{\boldsymbol{a}/\boldsymbol{b}}\!\!\!\!\rightarrow_A q'$ as follows: there exists $n \geq 0$ and $\{p_i \xrightarrow{\boldsymbol{a}_i/\boldsymbol{b}_i} p_{i+1} \mid i \leq n\} \subseteq [\![R]\!]$ such that

$$\boldsymbol{a} = \boldsymbol{a}_0 \cdot \boldsymbol{a}_1 \cdots \boldsymbol{a}_n, \quad \boldsymbol{b} = \boldsymbol{b}_0 \cdot \boldsymbol{b}_1 \cdots \boldsymbol{b}_n, \quad q = p_0, \quad q' = p_{n+1}.$$

Let also $q \xrightarrow{[]/[]}\!\!\!\!\rightarrow_A q$ for all $q \in Q$.

Does lookahead add expressiveness compared to SFTs? For finite Σ, the answer is *no*, because any concrete transition $p \xrightarrow{[a_0,a_1]/\boldsymbol{b}} q$ can be split into two transitions $p \xrightarrow{[a_0]/\boldsymbol{b}} p' \xrightarrow{[a_1]/[]} q$ where p' is a new (non final) state (as a consequence of the standard form [22, Theorem 2.17]). However, ESFTs are strictly more expressive than SFTs as the following example clearly illustrates. In general, ESFTs with lookahead $k+1$ are strictly more expressive than ESFTs with lookahead k.

Example 1. Let A be following ESFT

$$A = (\{q\}, q, \{q \xrightarrow[2]{(x_0=x_1)/[]} q, q \xrightarrow[0]{\mathtt{t}/[]} \bullet\})$$

Then $q \xrightarrow{\boldsymbol{a}/[]}\!\!\!\!\rightarrow_A \bullet$ iff $\boldsymbol{a}[2*i] = \boldsymbol{a}[2*i+1]$ for all $i \geq 0$. No SFT can express this dependency. ⊠

The above example can be generalized to any k. For a function $\mathbf{f} : X \to 2^Y$, define the *domain of* $\mathbf{f}$ as $\mathscr{D}(\mathbf{f}) \stackrel{\text{def}}{=} \{x \in X \mid \mathbf{f}(x) \neq \emptyset\}$; $\mathbf{f}$ is *total* when $\mathscr{D}(\mathbf{f}) = X$.

Definition 3. The *transduction of* A, $\mathscr{T}_A(\boldsymbol{a}) \stackrel{\text{def}}{=} \{\boldsymbol{b} \mid q^0 \xrightarrow{\boldsymbol{a}/\boldsymbol{b}}\!\!\!\!\rightarrow_A \bullet\}$.

The following subclass of SFTs captures transductions that behave as partial functions from Σ^* to Γ^*.

Definition 4. A is *single-valued* when $|\mathscr{T}_A(\boldsymbol{a})| \leq 1$ for all $\boldsymbol{a} \in \Sigma^*$.

A sufficient condition for single-valuedness is determinism. We define $\varphi \curlywedge \psi$, where φ is a σ^m-predicate and ψ a σ^n-predicate, as the $\sigma^{\max(m,n)}$-predicate $\lambda(x_1, \ldots, x_{\max(m,n)}).\varphi(x_1, \ldots, x_m) \wedge \psi(x_1, \ldots, x_n)$. We define *equivalence of* f *and* g *modulo* φ, $f \equiv_\varphi g$, as: $\mathit{IsValid}(\lambda\bar{x}.(\varphi(\bar{x}) \Rightarrow f(\bar{x}) = g(\bar{x})))$.

Definition 5. A is *deterministic* if and only if for all $p \xrightarrow[\ell]{\varphi/f} q, p \xrightarrow[\ell']{\varphi'/f'} q' \in R$ the following holds:

(a) Assume $q, q' \in Q$. If $\mathit{IsSat}(\varphi \curlywedge \varphi')$ then $q = q'$, $\ell = \ell'$ and $f \equiv_{\varphi \curlywedge \varphi'} f'$.
(b) Assume $q = q' = \bullet$. If $\mathit{IsSat}(\varphi \curlywedge \varphi')$ and $\ell = \ell'$ then $f \equiv_{\varphi \curlywedge \varphi'} f'$.
(c) Assume $q \in Q$ and $q' = \bullet$. If $\mathit{IsSat}(\varphi \curlywedge \varphi')$ then $\ell > \ell'$.

Proposition 1. *If A is deterministic then A is single-valued.*

Determinism is not a necessary condition for single-valuedness. Moreover, deterministic ESFTs with lookahead $k + 1$ are in general more expressive and more succinct than deterministic ESFTs with lookahead k. A few examples of ESFTs are given below to illustrate the definitions.

When A is total and single valued, and $\boldsymbol{a} \in \mathscr{D}(A)$, we write $A(\boldsymbol{a})$ for the value $\boldsymbol{b}$ such that $\mathscr{T}_A(\boldsymbol{a}) = \{\boldsymbol{b}\}$. In other words, we treat A as a function from Σ^* to Γ^*.

Example 2. Consider characters as their integer codes. We construct an ESFT *Decode* over $\mathbb{Z} \to \mathbb{Z}$ that replaces all occurrences of the regex pattern `#[0-9][0-9]` in the input with the corresponding encoded character. For example, since the code `'A'` $= 65$ and `'B'` $= 66$, we have *Decode*(`"##65#66#"`) $=$ `"#AB#"`.[3]

The states are q_0 and q_1, where q_0 is the initial state. The intuition behind the final rules is the following. In q_0 there is no unfinished pattern so the output is $[]$, while in q_1 the symbol `'#'` is the prefix of the unfinished pattern that needs to be output upon reaching the end of the input, and if there is only a single character x_0 remaining in the input then the output is $[$`'#'`$, x_0]$.

[3] A literal string `"ABC"` stands for the sequence [`'A'`, `'B'`, `'C'`], where `'A'` is the character code of letter `A`.

The rules of D are as follows where *IsDigit* is the predicate $\lambda x.\texttt{`0'} \leq x \leq \texttt{`9'}$ (recall that $\texttt{`0'} = 48$ and $\texttt{`9'} = 57$).

$$F = \{\ q_0 \xrightarrow[0]{\mathfrak{t}/[]} \bullet, \qquad q_1 \xrightarrow[0]{\mathfrak{t}/[\texttt{`\#'}]} \bullet, \qquad q_1 \xrightarrow[1]{\mathit{IsDigit}(x_0)/[\texttt{`\#'},x_0]} \bullet\}$$

$$\begin{aligned}\Delta = \{\ & q_0 \xrightarrow[1]{(x_0 \neq \texttt{`\#'})/[x_0]} q_0, \qquad q_0 \xrightarrow[1]{(x_0 = \texttt{`\#'})/[]} q_1, \qquad q_1 \xrightarrow[1]{(x_0 = \texttt{`\#'})/[\texttt{`\#'}]} q_1,\\ & q_1 \xrightarrow[1]{(x_0 \neq \texttt{`\#'} \wedge \neg \mathit{IsDigit}(x_0))/[\texttt{`\#'},x_0]} q_0,\\ & q_1 \xrightarrow[2]{(\mathit{IsDigit}(x_0) \wedge x_1 \neq \texttt{`\#'} \wedge \neg \mathit{IsDigit}(x_1))/[\texttt{`\#'},x_0,x_1]} q_0,\\ & q_1 \xrightarrow[2]{(\mathit{IsDigit}(x_0) \wedge x_1 = \texttt{`\#'})/[\texttt{`\#'},x_0]} q_1,\\ & q_1 \xrightarrow[2]{(\mathit{IsDigit}(x_0) \wedge \mathit{IsDigit}(x_1))/[10*(x_0-48)+x_1-48]} q_0\}\end{aligned}$$

The last three rules have non-overlapping guards because the conditions on x_1 are mutually exclusive. An equivalent SFT would require a state p_d for each $d \in [\![\mathit{IsDigit}]\!]$ and a rule $q_1 \xrightarrow[1]{x_0 = d/[]} p_d$ in order to eliminate the rules with lookahead 2. ⊠

2.1 Composition of ESFTs

For composing ESFTs we first convert them to STs, as explained in Section 2.2, and then convert the resulting ST back to an ESFT using the semi-decision procedure explained in Section 3. In general, ESFTs are not closed under composition, as shown next.

Given $\mathbf{f} : X \to 2^Y$ and $\mathbf{x} \subseteq X$, $\mathbf{f}(\mathbf{x}) \stackrel{\text{def}}{=} \bigcup_{x \in \mathbf{x}} \mathbf{f}(x)$. Given $\mathbf{f} : X \to 2^Y$ and $\mathbf{g} : Y \to 2^Z$, $\mathbf{f} \circ \mathbf{g}(x) \stackrel{\text{def}}{=} \mathbf{g}(\mathbf{f}(x))$. This definition follows the convention in [7], i.e., $\circ$ applies first $\mathbf{f}$, then $\mathbf{g}$, contrary to how $\circ$ is used for standard function composition. The intuition is that $\mathbf{f}$ corresponds to the relation $R_{\mathbf{f}} : X \times Y$, $R_{\mathbf{f}} \stackrel{\text{def}}{=} \{(x, y) \mid y \in \mathbf{f}(x)\}$, so that $\mathbf{f} \circ \mathbf{g}$ corresponds to the binary relation composition $R_{\mathbf{f}} \circ R_{\mathbf{g}} \stackrel{\text{def}}{=} \{(x, z) \mid \exists y(R_{\mathbf{f}}(x, y) \wedge R_{\mathbf{g}}(y, z))\}$.

Definition 6. *A class of transducer C is closed under composition iff for every $\mathscr{T}_1$ and $\mathscr{T}_2$ that are C-definable $\mathscr{T}_1 \circ \mathscr{T}_2$ is also C-definable.*

Theorem 1. *ESFTs are not closed under composition.*

Proof. We show two ESFTs whose composition cannot be expressed by any ESFT. Let A be following ESFT over $\mathbb{Z} \to \mathbb{Z}$

$$A = (\{q\}, q, \{q \xrightarrow[2]{\mathfrak{t}/[x_1,x_0]} q, q \xrightarrow[0]{\mathfrak{t}/[]} \bullet\}).$$

and B be following ESFT over $\mathbb{Z} \to \mathbb{Z}$

$$B = (\{q_0, q_1\}, q_0, \{q_0 \xrightarrow[1]{\mathfrak{t}/[x_0]} q_1, q_1 \xrightarrow[2]{\mathfrak{t}/[x_1,x_0]} q_1, q_1 \xrightarrow[1]{\mathfrak{t}/[x_0]} \bullet\})$$

The two transformations behave as in the following examples:

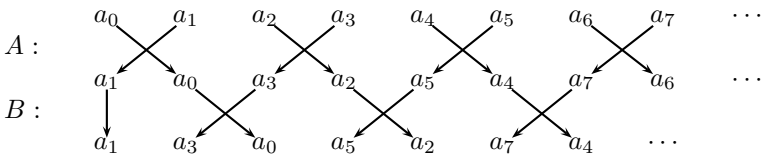
$$\mathscr{T}_A([a_0, a_1, a_2, a_3, a_4, a_5, a_6, \ldots]) = [a_1, a_0, a_3, a_2, a_5, a_4, a_7, \ldots]$$

$$\mathscr{T}_B([b_0, b_1, b_2, b_3, b_4, b_5, \ldots]) = [b_0, b_2, b_1, b_4, b_3, b_6, \ldots]$$

When we compose $\mathscr{T}_A$ and $\mathscr{T}_B$ we get the following transformation:

$$\mathscr{T}_{A\circ B}([a_0, a_1, a_2, a_3, a_4, a_5, a_6, \ldots]) = [a_1, a_3, a_0, a_5, a_2, a_7, \ldots]$$

Intuitively, looking at $\mathscr{T}_{A\circ B}$ we can see that no finite lookahead seems to suffice for this function. The following argument is illustrated by this figure:



Formally, for each a_i such that $i \geq 0$, $\mathscr{T}_{A\circ B}$ is the following function:

- if $i = 1$, a_i is output at position 0;
- if i is even and greater than 1, a_i is output at position $i - 2$;
- if i is equal to $k-2$ where k is the length of the input, a_i is output at position $k - 1$;
- if i is odd and different from $k - 2$, a_i is output at position $i + 2$.

It is easy to see that the above transformation cannot be computed by any ESFT. Let's assume by contradiction that there exists an ESFT that computes $\mathscr{T}_{A\circ B}$. We consider the ESFT C with minimal lookahead (let's say n) that computes $\mathscr{T}_{A\circ B}$.

We now show that on an input of length greater than $n+2$, C will misbehave. The first transition of C that will apply to the input will have a lookahead of size $l \leq n$. We now have three possibilities (the case $n = k - 2$ does not apply due to the length of the input):

$l = 1$**:** before outputting a_0 (at position 2) we need to output a_1 and a_3 which we have not read yet. Contradiction;

l **is odd:** position $l + 1$ is receiving a_{l-1} therefore C must output also the elements at position l. Position l should receive a_{l+2} which is not reachable with a lookahead of just l. Contradiction;

l **is even and greater than** 1**:** since $l > 1$, position l is receiving a_{l-2}. This means C is also outputting position $l - 1$. Position $l - 1$ should receive a_{l+1} which is not reachable with a lookahead of just l. Contradiction;

We now have that n cannot be the minimal lookahead which contradicts our initial hypothesis. Therefore $\mathscr{T}_{A\circ B}$ is not ESFT-definable. ⊠

2.2 Symbolic transducers with registers

Registers provide a practical generalization of SFTs. SFTs with registers are called STs, since their state space (reachable by registers) may no longer be finite. An ST uses a *register* as a symbolic representation of states in addition to explicit (control) states. The rules of an ST are guarded commands with a symbolic input and output component that may use the register. By using Cartesian product types, multiple registers are represented with a single (compound) register. Equivalence of STs is undecidable but STs are closed under composition [20].

Definition 7. A *Symbolic Transducer* or *ST* over $\sigma \to \gamma$ and register type τ is a tuple $A = (Q, q^0, \rho^0, R)$,

- Q is a finite set of *states*;
- $q^0 \in Q$ is the *initial state*;
- $\rho^0 \in \mathcal{U}^\tau$ is the *initial register value*;
- R is a finite set of *rules* $R = \Delta \cup F$;
- Δ is a set of *transitions* $r = (p, \varphi, o, u, q)$, also denoted $p \xrightarrow{\varphi/o;u} q$,
  - $p \in Q$ is the *start* state of r;
  - φ, the *guard* of r, is a $(\sigma \times \tau)$-predicate;
  - o, the *output* of r, is a finite sequence of $((\sigma \times \tau) \to \gamma)$-terms;
  - u, the *update* of r, is a $((\sigma \times \tau) \to \tau)$-term;
  - $q \in Q$ is the *end* state of r.
- F is a set of *final rules* $r = (p, \varphi, o)$, also denoted $p \xrightarrow{\varphi/o} \bullet$,
  - $p \in Q$ is the *start* state of r;
  - φ, the *guard* of r, is a τ-predicate;
  - o, the *output* of r, is a finite sequence of $(\tau \to \gamma)$-terms.

All ST rules in R have lookahead 1 and all final rules have lookahead 0. Longer lookaheads are not needed because registers can be used to record history, in particular they may be used to record previous input characters. A canonical way to do so is to let τ be σ^* that records previously seen characters, where initially $\rho^0 = []$, indicating that no input characters have been seen yet.

An ESFT transition

$$p \xrightarrow[3]{\lambda(x_0,x_1,x_2).\varphi(x_0,x_1,x_2)/\lambda(x_0,x_1,x_2).o(x_0,x_1,x_2)} q$$

can be encoded as the following set of ST rules where p_1 and p_2 are new states

$$p \xrightarrow{(\lambda(x,y).\mathfrak{t})/[];\lambda(x,y).cons(x,[])} p_1 \quad p_1 \xrightarrow{(\lambda(x,y).\mathfrak{t})/[];\lambda(x,y).cons(x,y)} p_2$$
$$p_2 \xrightarrow{(\lambda(x,y).\varphi(y[1],y[0],x))/\lambda(x,y).o(y[1],y[0],x);\lambda(x,y).[]} q$$

Final rules are encoded similarly. The only difference is that q above is $\bullet$ and the register updated is not used in the third rule. An ST rule $(p, \varphi, o, u, q) \in R$ denotes the following set of concrete transitions:

$$[\![(p, \varphi, o, u, q)]\!] \stackrel{\text{def}}{=} \{(p, s) \xrightarrow{a/[\![o]\!](a,s)} (q, [\![u]\!](a, s)) \mid (a, s) \in [\![\varphi]\!]\}$$

A final ST rule $(p, \varphi, o) \in F$ denotes the following set of concrete transitions:

$$[\![(p, \varphi, o)]\!] \stackrel{\text{def}}{=} \{(p, s) \xrightarrow{[]/[\![o]\!](s)} \bullet \mid s \in [\![\varphi]\!]\}$$

The reachability relation $p \xrightarrow{\boldsymbol{a}/\boldsymbol{b}}\!\!\!\!\rightarrow_A q$ for $\boldsymbol{a} \in \Sigma^*$, $\boldsymbol{b} \in \Gamma^*$, $p \in (Q \times \mathcal{U}^\tau)$, $q \in (Q \times \mathcal{U}^\tau) \cup \{\bullet\}$ is defined analogously to ESFTs and $\mathscr{T}_A(\boldsymbol{a}) \stackrel{\text{def}}{=} \{\boldsymbol{b} \mid (q^0, \rho^0) \xrightarrow{\boldsymbol{a}/\boldsymbol{b}} \bullet\}$.

3 Register Elimination

The main advantage of STs is their succinctness and the fact that Bek programs can directly be mapped to STs. The downside of using STs is that many of the desired properties, such as equivalence, idempotence, and commutativity, are no longer decidable. One approach to decide those properties is to transform STs to SFTs by exploring all the possible register values. However, this is only possible for finite alphabets and in general not feasible due to state space explosion. This is where ESFTs play a central role.

In this section we describe an algorithm that allows us to transform a class of STs into ESFTs. The algorithm has several applications.

One application is to eliminate registers at the expense of increasing the lookahead. The algorithm can be applied to a class of *product* STs with *two outputs*. The algorithm is agnostic about how the output looks like. Product STs are used in the single-valuedness checking algorithm of SFTs.

While ESFTs provide a powerful generalization of SFTs, they are unfortunately not closed under composition as shown in Section 2.1. Another application of the transformation algorithm is a semi-decision procedure for composing ESFTs. The technique is to first translate the ESFTs to STs, as outlined in Section 2.2, then compose the STs, and finally apply the register elimination algorithm to convert the composed ST back to an ESFT, if possible. We are currently investigating which subclasses of ESFT are effectively closed under composition.

The core idea that underlies the register elimination algorithm is a symbolic generalization of the classical state elimination algorithm for converting an NFA to a regular expression (see e.g. [22, Section 3.3]), that uses the notion of extended automata whose transitions are labelled by regular expressions. Here the labels of the ST are predicates over sequences of elements of fixed lookahead. Essentially the intermediate data structure of the algorithm is an Extended ST.

Input: ST $A^{\sigma/\gamma;\tau}$.
Output: $\perp$ or an ESFT over $\sigma \to \gamma$ that is equivalent to A.

1. Lift A to use the input sort σ^* by replacing each transition $p \xrightarrow{\varphi/o;u} q$ with the following transition annotated with a lookahead of 1 and with $x : \sigma^*$,

$$p \xrightarrow[1]{\lambda(x,y).x \neq [] \wedge \varphi(x[0],y)/\lambda(x,y).o(x[0],y);\lambda(x,y).u(x[0],y)} q$$

(Apply similar transformation to final rules and give them lookahead 0.)

2. Repeat the steps 2.a-2.c while there exists a state that does not have a self loop (a self loop is a transition whose start and end states are equal).

2.a Choose a state p, such that p is not the state of any self loop and p is not the initial state.

2.b Do for all transitions $p_1 \xrightarrow[k]{\varphi_1/o_1;u_1} p \xrightarrow[\ell]{\varphi_2/o_2:u_2} p_2$ in R:

let $\varphi = \lambda(x,y).\varphi_1(x,y) \wedge \varphi_2(tail^k(x), u_1(x,y))$
let $o = \lambda(x,y).o_1(x,y) \cdot o_2(tail^k(x), u_1(x,y))$
let $u = \lambda(x,y).u_2(tail^k(x), u_1(x,y))$
if $IsSat(\varphi)$ **then let** $r = p_1 \xrightarrow[k+\ell]{\varphi/o;u} p_2$ and add r as a new rule

2.c Delete the state p.

3. If no guard and no output depends on the register, remove the register from all the rules in the ST and return the resulting ST as an ESFT, otherwise return $\perp$.

After the first step, the original ST accepts an input $[a_0, a_1, a_2]$ and produces output v iff the transformed ST accepts $[cons(a_0, \_), cons(a_1, \_), cons(a_2, \_)]$ and produces output v, where the tails $\_$ are unconstrained and irrelevant. Step 2 further groups the inputs characters, e.g., to $[cons(a_0, cons(a_1, \_)), cons(a_2, \_)]$, etc, while maintaining this input/output property with respect to the original ST. Finally, in step 3, turning the ST into an ESFT, leads to elimination of the register as well as lowering of the character sort back to σ, and replacing each occurrence of $tail^k(x)$ with corresponding individual tuple element variable x_k. Soundness of the algorithm follows.

The algorithm omits several implementation aspects that have considerable effect on performance. One important choice is the order in which states are removed. In our implementation the states with lowest total number of incoming and outgoing rules are eliminated first. It is also important to perform the choices in an order that avoids unreachable state spaces. For example, the elimination of a state p in step 2 may imply that φ is unsatisfiable and consequently that p_2 is unreachable if the transition from p is the only transition leading to p_2. In this case, if p is reachable from the initial state, choosing p_2 before p in step 2 would be wasteful.

4 Unicode Case Study

In this section we show how to describe realistic encoding and decoding routines using STs. We use Bek as the concrete programming language for STs.

A *hextet* is a non-negative integer $< 2^{16}$, and an *octet* is a non-negative integer $< 2^8$, i.e., hextets correspond to 16-bit bitvectors and octets correspond to bytes. Hextets are used in modern programming and scripting languages to represent character codes. For example, in C# as well as in JavaScript, string representation involves arrays of characters, where each character has a unique numeric code in form of a hextet. For example, the JavaScript expression `String.fromCharCode(0x48, 0x65, 0x6C, 0x6C, 0x153, 0x21)` equals to the string "`Hellœ!`".

```
program utf8encode(input){
  return iter(c in input)[H:=false; r:=0;]
  {
    case (!H&&(0<=c)&&(c<=0x7F)): yield(c);                          //one octet
    case (!H&&(0x7F<c)&&(c<=0x7FF)):
      yield(0xC0|((c>>6)&0x1F), 0x80|(c&0x3F));                      //two octets
    case (!H&&(0x7FF<c)&&(c<=0xFFFF)&&((c<0xD800)||(c>0xDFFF))):
      yield(0xE0|((c>>12)&0xF), 0x80|((c>>6)&0x3F), 0x80|(c&0x3F)); //three octets
    case (H&&(0xDC00<=c)&&(c<=0xDFFF)):  H:=false; r:=0;            //low surrogate
      yield((0x80|(r << 4))|((c>>6)&0xF), 0x80|(c&0x3F));
    case (!H&&(0xD800<=c)&&(c<=0xDBFF)): H:=true; r:=c&3;           //high surrogate
      yield (0xF0|(((1+((c>>6)&0xF))>>2)&7),(0x80|(((1+((c>>6)&0xF))&3)<<4))|((c>>2)&0xF));
    case (true):  raise InvalidInput;
    end case (H): raise InvalidInput;
  };
}

program utf8decode(input){
  return iter(c in input)[q:=0; r:=0;]
  {
    case ((q==0)&&(0<=c)&&(c<=0x7F)):    yield (c);
    case ((q==0)&&(0xC2<=c)&&(c<=0xDF)): q:=3; r:=(c&0x3F)<<6;
    case ((q==0)&&(c==0xE0)):            q:=7;
    case ((q==0)&&(c==0xED)):            q:=6;
    case ((q==0)&&(0xE1<=c)&&(c<=0xEF)): q:=2; r:=(c&0xF)<<12;
    case ((q==0)&&(0xF1<=c)&&(c<=0xF3)): q:=1; r:=(c&7)<<8;
    case ((q==0)&&(c==0xF0)):            q:=4;
    case ((q==0)&&(c==0xF4)):            q:=5; r:=0x400;
    case ((q==1)&&(0x80<=c)&&(c<=0xBF)): q:=8; r:=0xD800|(((r|((c&0x30)<<2))-0x40)|((c&0x0F)<<2));
    case ((q==4)&&(0x90<=c)&&(c<=0xBF)): q:=8; r:=0xD800|((((c&0x30)<<2)-0x40)|((c&0x0F)<<2));
    case ((q==5)&&(0x80<=c)&&(c<=0x8F)): q:=8; r:=0xD800|(((r|((c&0x30)<<2))-0x40)|((c&0x0F)<<2));
    case ((q==2)&&(0x80<=c)&&(c<=0xBF)): q:=3; r:=r|((c&0x3F)<<6);
    case ((q==6)&&(0x80<=c)&&(c<=0x9F)): q:=3; r:=0xD000|((c&0x3F)<<6);
    case ((q==7)&&(0xA0<=c)&&(c<=0xBF)): q:=3; r:=(c&0x3F)<<6;
    case ((q==8)&&(0x80<=c)&&(c<=0xBF)): q:=3; yield(r|((c>>4)&3)); r:=0xDC00|((c&0xF)<<6);
    case ((q==3)&&(0x80<=c)&&(c<=0xBF)): q:=0; yield(r|(c&0x3F)); r:=0;
    case (true):        raise InvalidInput;
    end case (!(q==0)): raise InvalidInput;
  };
}
```

Fig. 1. UTF-8 encoder and decoder in Bek.

A Unicode *code point* is an integer between 0 and 1,112,064 ($\texttt{10FFFF}_{16}$). *Surrogates* are code points between $\texttt{D800}_{16}$ and $\texttt{DFFF}_{16}$ and are not valid character code points according to the UTF-8 definition.[4]

UTF-16 is the standard character encoding used in modern programming and scripting languages. With UTF-16 format, Unicode symbols are represented either directly by hextets, or as pairs of hextets, so called *surrogate pairs*, that represent symbols in the upper Unicode range, e.g., the musical symbol 𝄵 called "cut time" has Unicode code point $\texttt{1D135}_{16}$ that is encoded in UTF-16 by the surrogate pair $[\texttt{D834}_{16}, \texttt{DD35}_{16}]$.

Not all sequences of hextets represent well-formed UTF-16 strings. Well-formed UTF-16 strings are precisely all those sequences of hextets that match the regular expression and corresponding symbolic finite automaton in Figure 2.

[4] `http://tools.ietf.org/html/rfc3629`.


```
^([\0-\uD7FF\uE000-\uFFFF]|([\uD800-\uDBFF][\uDC00-\uDFFF]))*$
```

Fig. 2. UTF-16 validator. All numbers use hexadecimal notation and the range expression m-n is short for the predicate $\lambda x.m \leq x \leq n$.

Elements in the ranges `[\uD800-\uDBFF]` and `[\uDC00-\uDFFF]` are called *high surrogates* and *low surrogates*, respectively. A surrogate pair $[high, low]$ represents the Unicode symbol whose code point is $((high_{\langle 9,0\rangle} \ll 10)|low_{\langle 9,0\rangle}) + \texttt{10000}_{16}$, where $x_{\langle m,n\rangle}$ extracts bits m through n from x.

UTF-8 UTF-8 uses sequences of one up to four octets to encode single Unicode code points. Let c be a Unicode codepoint, the UTF-8 encoding of c is:

$$Utf8(c) \stackrel{\text{def}}{=} \begin{cases} [c], & \text{if } 0 \leq c \leq \texttt{7F}_{16}; \\ [\texttt{C0}_{16}|c_{\langle 10,6\rangle}, \texttt{80}_{16}|c_{\langle 5,0\rangle}], & \text{if } c \leq \texttt{7FF}_{16}; \\ [\texttt{E0}_{16}|c_{\langle 15,12\rangle}, \texttt{80}_{16}|c_{\langle 11,6\rangle}, \texttt{80}_{16}|c_{\langle 5,0\rangle}], & \text{if } c \leq \texttt{FFFF}_{16}; \\ [\texttt{F0}_{16}|c_{\langle 20,18\rangle}, \texttt{80}_{16}|c_{\langle 17,12\rangle}, \texttt{80}_{16}|c_{\langle 11,6\rangle}, \texttt{80}_{16}|c_{\langle 5,0\rangle}], & \text{otherwise.} \end{cases}$$

Some codepoints are not valid. In particular, in the third case, c may not be a surrogate, i.e., $c < \texttt{D800}_{16}$ or $c > \texttt{DFFF}_{16}$. Also, any number greater than $\texttt{10FFFF}_{16}$ is invalid. The exact details of how the UTF-8 encoding is computed from the UTF-16 encoding follows from the Bek program in Figure 1 and is discussed below.

Conversions A UTF-16 to UTF-8 encoder takes a well-formed UTF-16 encoded string and converts it into the equivalent UTF-8 representation. Figure 1 shows the Bek program of such an encoder. The program makes essential use of bitwise operations over hextets, in particular, it uses bit shifting operations and logical bit operations. The input to the program is a sequence of hextets. The output produced by the encoder is a sequence of octets.

The Bek program represents a symbolic transducer that uses two registers: the Boolean register `H` and the character register `r`. The value of `H` is true if the previous character was a high surrogate, in which case the register `r` contains the two least significant bits of that high surrogate. When a low surrogate is input, it is used together with the value of `r` to yield the remaining two octets of the combined code point (the first two octets were output when the high surrogate was read). Both registers can be effectively eliminated by using an exploration algorithm. The resulting SFT is illustrated in Figure 3, where *HS* is the predicate for high surrogates and *LS* is the predicate for low surrogates. The rules of the SFT correspond to the different branches of the Bek program, where the exception cases correspond to the cases that are *disabled* at the respective states. What is quite remarkable is that the SFT has 11 rules and 5 states in total, compared to an equivalent classical finite state transducer that would require 2^{16} transitions (one transition per hextet).

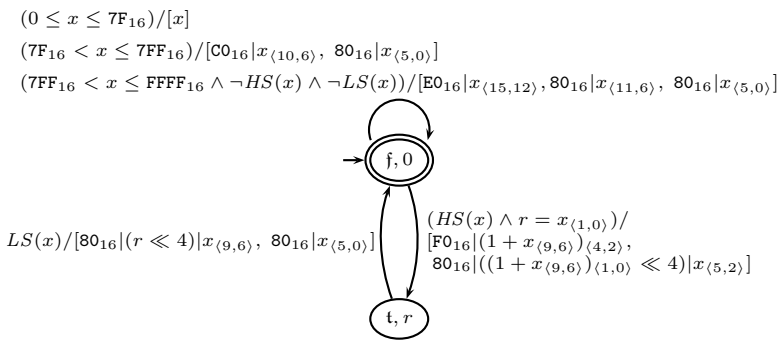


Fig. 3. SFT that is equivalent to the Bek program `utf8encode` in Figure 1. States are labelled by values of $(\mathtt{H}, \mathtt{r})$, there are five states: $(\mathfrak{f}, 0)$, $(\mathfrak{t}, 0)$, $(\mathfrak{t}, 1)$, $(\mathfrak{t}, 2)$, and $(\mathfrak{t}, 3)$.

The program `utf8decode` in Figure 1 provides the inverse conversion from valid UTF-8 encoded sequences to valid UTF-16 encoded sequences. The equivalent SFT has in this case 1284 states and 6371 rules, that is in sharp contrast to the 5 states and 11 rules of the encoder.

5 Experiments and Evaluation

We first verify the functional correctness of *UTF8* encoder and decoder. Secondly, we analyze how the register elimination algorithm of Section 3 scales for different program's sizes.

5.1 Functional correctness of encoders and decoders

The Bek programs in Figure 1 can be analyzed for various properties of interest by first converting them to STs and then to ESFT. While this analysis is very efficient for the ESFT of the encoder in Figure 3 it is more demanding for the decoder because of the size of the state space. As a fundamental correctness criterion, the valid input sequences of *utf8decode* should be the set of all valid UTF-8 sequences: $\mathscr{D}(\mathit{utf8decode}) = \mathscr{D}(\mathit{UTF8})$ where *UTF8* is the UTF-8 validator expressed as a Symbolic Automaton. An inspection of *UTF8* shows that the following validity properties are checked:

1. Octets $\texttt{C0}_{16}$, $\texttt{C1}_{16}$, $\texttt{F5}_{16}, \ldots, \texttt{FF}_{16}$ are disallowed.
2. Invalid combinations of start-octets and continuation-octets are disallowed.
3. Sequences that decode to a value with a shorter encoding (so called "overlong" sequences) are disallowed.
4. Sequences starting with $\texttt{F4}_{16}$ encoding a value $> \texttt{10FFFF}_{16}$ are disallowed.
5. Encodings of surrogates (having start-octet $\texttt{ED}_{16}$) are disallowed.

In particular, overlong encodings, such as the encoding $[\texttt{C0}_{16}, \texttt{AE}_{16}]$ of '.', are disallowed.

Moreover, we expect the decoder to perform the inverse of the encoder and vice versa. Let I be the identity SFT, i.e., I has a single state q_0 and a single rule $q_0 \xrightarrow{\lambda x.\mathrm{t}/[\lambda x.x]} q_0$. Let $E = \mathscr{T}_{utf8encode}$, $D = \mathscr{T}_{utf8decode}$, $U_{\mathrm{utf16}} = \mathscr{D}(UTF16)$, and $U_{\mathrm{utf8}} = \mathscr{D}(UTF8)$. The following must hold:

- $E \circ D \overset{1}{=} I$, $\mathscr{D}(E) = \mathscr{D}(E \circ D) = U_{\mathrm{utf16}}$
- $D \circ E \overset{1}{=} I$, $\mathscr{D}(D) = \mathscr{D}(D \circ E) = U_{\mathrm{utf8}}$

$A \overset{1}{=} B$, iff A and B produce the same output on each of the inputs in their domain intersection (1-equality [20]). Consider $D : U_{\mathrm{utf16}} \to U_{\mathrm{utf8}}$ and $E : U_{\mathrm{utf8}} \to U_{\mathrm{utf16}}$ as functions. Thus D and E are *bijections* and inverses of each other.

5.2 Use of register elimination

In general, an ESFT to SFT conversion is needed for deciding 1-equality. The previous technique eliminated the registers by fully exploring their reachable state space and created an SFT prior to invoking the equivalence algorithm [21]. The technique introduced here takes a step further. It gradually increases the lookahead of a *2-output ST* by shortcutting intermediate states in an attempt to completely eliminate register dependencies from the guards and the output terms. First, we convert ESFTs to STs by introducing registers, as explained in Section 2.2. Second, we compute a 2-output ST as a product of the two STs that has synchronized input and where infeasible guard combinations have been eliminated, corresponding to product-SFTs in [20, Definition 7]. Third, we compute an equivalent 2-output ESFT (when possible) from the 2-output ST using the register elimination algorithm explained in Section 3. Fourth, we convert that 2-output ESFT into a 2-output SFT whose characters are grouped into sequences of characters of given lookahead length and note that this transformation preserves one-equality of the original ESFTs due to the synchronized inputs. Finally, we apply a variation of the one-equality algorithm for SFTs to the 2-output product SFT that is the value of C in the one-equality algorithm [20, Figure 3]. The procedure described above might not terminate. However this has not been the case in our case study. We are currently investigating decidability of one-equality of ESFTs, and a more direct approach.

We illustrate the register elimination algorithm, in the case of composition, using some rules of the case study. Consider the composition $ED = E \circ D$ (encoding followed by decoding). The resulting ST ED uses a register (inherited from D), and has 5 states and 22 rules. Besides the initial state q_0, all other states q are non-final and *intermediate* in the following sense: all paths through q have the form:

$$q_0 \xrightarrow[1]{\varphi(x_0)/([],f(x_0))} q \xrightarrow[1]{\psi(x_0,y)/(\beta(x_0,y),0)} q_0$$

where the first rule is independent of the register (depends only on the input element x_0). The path can be represented with an equivalent rule with a lookahead of 2 elements:

$$q_0 \xrightarrow[2]{\varphi(x_0)\wedge\psi(x_1,f(x_0))/(\beta(x_1,f(x_0)),0)} q_0$$

After the removal of all such intermediate states from ED, the register y can be deleted from all the new rules because no guard or output depends on the register.[5]

Table 1 shows the running times of the operations needed to perform the checks described above.

Operation	Running Time
$\mathscr{D}(E) = U_{\text{utf16}}$	47 ms
$\mathscr{D}(E \circ D) = U_{\text{utf16}}$	109 ms
$\mathscr{D}(D) = U_{\text{utf8}}$	156 ms
$\mathscr{D}(D \circ E) = U_{\text{utf8}}$	320 ms
$E \circ D \overset{1}{=} I$ (exploration)	82,000 ms
$D \circ E \overset{1}{=} I$ (exploration)	134,000 ms
$E \circ D \overset{1}{=} I$ (reg. elim.)	**123** ms
$D \circ E \overset{1}{=} I$ (reg. elim.)	**215** ms

Table 1. Running Times for Functional Correctness

The first four entries of Table 1 show the running time for the domain checks. The times are all under 0.4 seconds. To perform the domain checks we compute the ESFAs (ESFTs with empty outputs) corresponding to domain of the ESFTs, and we check for automata equivalence. In order to improve the efficiency, when possible, we transform ESFAs into equivalent SFAs (SFTs with empty outputs) and use the equivalence algorithm for SFAs [20]. Without this transformation, some of the running times would be above 1 minute.

The next two entries of Table 1 show the running times for 1-equality with the exploration algorithm in [21]. It is clear from the data (> 100 seconds) that the state explosion causes the algorithm to work only on programs of small sizes. Finally, the last two entries of the table represent the running time for the improved algorithm of Section 3 with register elimination. It is important to point out that in this case we do not check for domain equivalence but only for 1-equality. Performing the check after the register elimination algorithm achieves a 600x speed-up against the full exploration version in this particular case study. This is a consequence of the succinctness of ESFTs.

5.3 Running time analysis with register elimination

In this section we run our register elimination algorithm on bigger program instances. Most of the checks performed in this section will time out (longer than 1 hour) when using the full exploration algorithm.

We consider consecutive compositions of encoders and decoders and analyze their correctness using 1-equality for ESFTs. This experiment is motivated by a common form of attack called *double encoding* [15]. This attack technique

[5] The online Bek tutorial `http://www.rise4fun.com/Bek/tutorial/utf8` contains further analysis scenarios.

consists of encoding the user input twice in order to cause unexpected behaviour. We define the following notation for consecutive composition of STs. Given an ST P we define $P^1 \equiv P$ and $P^{i+1} \equiv P \circ P^i$. We verify the following properties and analyze their execution times.

Equivalence for Enc/Dec: $E^i \circ D^i \overset{1}{=} I$ for $1 \leq i \leq 9$, Figure 4(a);
Inequivalence for Enc/Dec: $E^{i+1} \circ D^i \overset{1}{\neq} I$ for $1 \leq i \leq 9$, Figure 4(a);
Equivalence for Dec/Enc: $D^i \circ E^i \overset{1}{=} I$ for $1 \leq i \leq 3$, Figure 4(b);
Inequivalence for Dec/Enc: $D^i \circ E^{i+1} \overset{1}{\neq} I$ for $1 \leq i \leq 3$, Figure 4(b).

Figure 4(a) shows the running times for the case in which we first encode and then decode. The figure plots the following measures where i varies between 1 and 9:

Composition: cost of computing $E^{i+1} \circ D^i$ (we omit the cost of computing $E^i \circ D^i$ since it is almost equivalent);
Equivalence: cost of checking $E^i \circ D^i \overset{1}{=} I$;
Inequivalence: cost of checking $E^{i+1} \circ D^i \overset{1}{\neq} I$.

In this case the algorithm scales pretty well with the number of STs. It is worth noticing that at every i we are analyzing the composition of $2i$ transducers in the case of equivalence and $2i + 1$ transducers in the case of inequivalence.

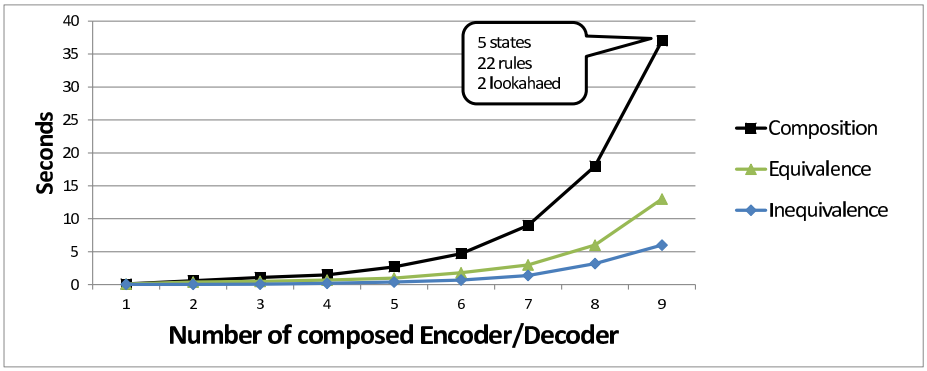
Figure 4(b) shows the running times for the case in which we first decode and then encode. The plot has the same meaning as before, but in this case the running time increases at a faster pace. This happens for two reasons: 1) the state space is bigger, and 2) the lookahead is bigger.

In the case in which we first encode the number of states and transitions does not grow when i increases. However, when we first decode, we early ($i = 3$) reach a big number of states (3645) and transitions (6791). Moreover, while the size of the lookahead in the first case remains the same (it is always 2), it grows exponentially with i when we first decode. Indeed for $i = 1, 2, 3$ we have lookaheads of size $4, 8, 16$ respectively. This causes the register elimination algorithm to explore more paths.

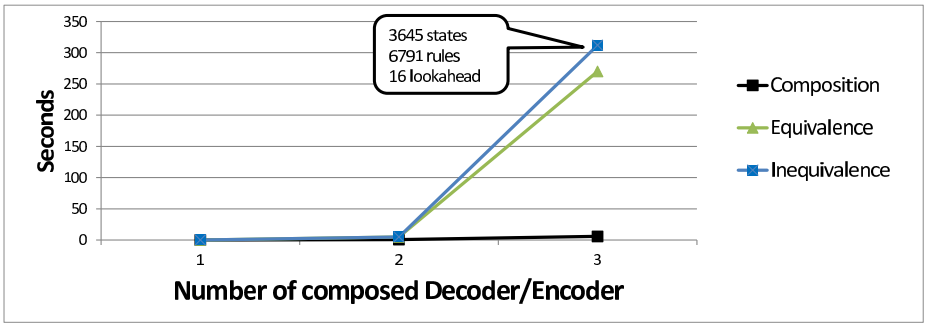
We noticed when we composed the encoder with the decoder, that the lookahead size and the number of states and transitions do not grow when i increases. However, Figure 4(a) shows that the running time grows exponentially in i. The complexity indeed does not only depend on the size of the ESFT, but also on the size of its predicates. The predicate sizes increase when we compose STs, causing the SMT solver to affect the performance of both the composition and equivalence algorithms, which perform several satisfiability checks on predicates.

6 Related Work

Symbolic finite transducers (SFTs) and Bek were originally introduced in [9] with a focus on security analysis of sanitizers. The key properties that are studied in [9] from a practical point of view are idempotence, commutativity and



(a) Encoding then Decoding



(b) Decoding then Encoding

Fig. 4. Running time in *seconds* for equivalence/inequivalence checking of multiple compositions of encoders and decoders.

equivalence checking of sanitizers. The formal foundations and the theoretical analysis of the underlying SFT algorithms, in particular, an algorithm for deciding equivalence of single-valued SFTs, modulo a decidable background theory is studied in [20], including a more general *1-equality* algorithm that factors out the decision problem for single-valuedness, and allows non-determinism without violating single-valuedness. The formalism of SFTs is extended in [20] to Symbolic Transducers (STs) that allow the use of registers. A "brute-force" exploration algorithm for register elimination is analyzed in [21]. However, the algorithm only copes with finite-ranged register updates and generally produces large state spaces. The focus and the motivation of the current paper is *efficient register elimination*. We introduce *extended symbolic finite transducers* (ESFTs) which are strictly more expressive than SFTs. We then propose an algorithm that compiles a subclass of STs to ESFTs and that does not assume the input alphabet to be finite. Finally, the succinctness of ESFTs enables fast analysis of previously intractable (or not expressible) programs.

In recent years there has been considerable interest in automata using infinite alphabets [18], starting with the work on *register automata* [11]. Finite words over an infinite alphabet are often called *data words* in the literature. This line of work focuses on fundamental questions about definability, decidability, complexity, and expressiveness on classes of automata on one hand and fragments of logic on the other hand.

Streaming transducers [1] provide another recent symbolic extension of finite transducers where the label theories are restricted to be total orders, in order to maintain decidability of equivalence. Streaming transducers are largely orthogonal to SFTs or the extension of ESFTs, as presented in the current paper. For example, streaming transducers allow reversing the input, which is not possible with ESFTs, while arithmetic is not allowed in streaming transducers but plays a central role in our applications of ESFTs to string encoders.

We use the SMT solver Z3 [6] for incrementally solving label constraints that arise during the exploration algorithm. Similar applications of SMT techniques have been introduced in the context of symbolic execution of programs by using path conditions to represent under and over approximations of reachable states [8]. The distinguishing feature of our exploration algorithm is that it computes a precise transformation that is symbolic with respect to input labels, while allowing different levels of concretization with respect to the state variables. The resulting extended symbolic finite transducer is not an under or over approximation, but functionally *equivalent* to the original symbolic transducer. This is important for achieving a sound and complete analysis.

Our work is complementary to previous efforts in using SMT solvers to solve problems related to list transformations. Kaluza [17] extends the SMT solver to handle equations over strings and equations with multiple variables. We are not aware of previous work investigating the use of finite transducers for verifying code as complex as *utf8encoder* and *utf8decoder*. One obvious explanation for this is that classical finite transducers are not directly suited for this purpose; indeed, symbolic finite transducers can be exponentially more succinct than classical finite transducers with respect to alphabet size.

7 Conclusions

Several web applications assume the correctness of encoding and decoding functions. However, practical experience shows that writing correct encoders and decoders is a hard task. This paper presents an algorithmic extension of Bek, a language for writing, analyzing string manipulation routines. We introduce extended symbolic finite transducers (ESFTs) to enable analysis of previously intractable programs such as string decoders. We prove correctness of *UTF8* encoder and decoder, even in the case of double encoding. We show that our algorithms are fast in practice, and scale up to 20 encoder/decoder compositions.

References

1. R. Alur and P. Cerný. Streaming transducers for algorithmic verification of single-pass list-processing programs. In *POPL'11*, pages 599–610. ACM, 2011.
2. Bek. `http://research.microsoft.com/bek`.
3. N. Bjørner, V. Ganesh, R. Michel, and M. Veanes. An SMT-LIB format for sequences and regular expressions. In P. Fontaine and A. Goel, editors, *SMT'12*, pages 76–86, 2012.
4. N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In S. Kowalewski and A. Philippou, editors, *TACAS*, volume 5505 of *LNCS*, pages 307–321. Springer, 2009.
5. A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise Analysis of String Expressions. In *SAS*, 2003.
6. L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS'08*, LNCS. Springer, 2008.
7. Z. Fülöp and H. Vogler. *Syntax-Directed Semantics: Formal Models Based on Tree Transducers*. EATCS. Springer, 1998.
8. P. Godefroid. Compositional dynamic test generation. In *POPL'07*, pages 47–54, 2007.
9. P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and precise sanitizer analysis with Bek. In *Proceedings of the USENIX Security Symposium*, August 2011.
10. P. Hooimeijer and M. Veanes. An evaluation of automata algorithms for string analysis. In *VMCAI'11*, volume 6538 of *LNCS*, pages 248–262. Springer, 2011.
11. M. Kaminski and N. Francez. Finite-memory automata. *TCS*, 134(2):329–363, 1994.
12. B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee. Merlin: specification inference for explicit information flow problems. In *PLDI'09*, pages 75–86. ACM, 2009.
13. Y. Minamide. Static approximation of dynamically generated web pages. In *WWW '05: Proceedings of the 14th International Conference on the World Wide Web*, pages 432–441, 2005.
14. NVD. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2008-2938`.
15. OWASP. Double encoding. `https://www.owasp.org/index.php/Double_Encoding`.
16. SANS. Malware faq. `http://www.sans.org/security-resources/malwarefaq/w-nt-unicode.php`.
17. P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. Technical Report UCB/EECS-2010-26, Mar 2010.
18. L. Segoufin. Automata and logics for words and trees over an infinite alphabet. In *CSL*, pages 41–57, 2006.
19. M. Veanes, P. de Halleux, and N. Tillmann. Rex: Symbolic Regular Expression Explorer. In *ICST'10*, pages 498–507. IEEE, 2010.
20. M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjorner. Symbolic finite state transducers: Algorithms and applications. In *POPL'12*, pages 137–150, 2012.
21. M. Veanes, D. Molnar, T. Mytkowicz, and B. Livshits. Data-parallel string-manipulating programs. Technical Report MSR-TR-2012-72, Microsoft Research, 2012.
22. S. Yu. Regular languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 1, pages 41–110. Springer, 1997.