Data-Parallel String-Manipulating Programs

Margus Veanes Todd Mytkowicz David Molnar Benjamin Livshits
Microsoft Research
{margus,toddm,dmolnar,livshits}@microsoft.com

Abstract

String-manipulating programs are an important class of programs with applications in malware detection, graphics, input sanitization for Web security, and large-scale HTML processing. This paper extends prior work on BEK, an expressive domain-specific language for writing string-manipulating programs, with algorithmic insights that make BEK both *analyzable* and *data-parallel*. By analyzable we mean that unlike most general purpose programming languages, many algebraic properties of a BEK program are decidable (i.e., one can check whether two programs commute or compute the inverse of a program). By data-parallel we mean that a BEK program can compute on arbitrary *subsections* of its input in parallel, thus exploiting parallel hardware. This latter requirement is particularly important for programs which operate on large data: without data parallelism, a programmer cannot hide the latency of reading data from various storage media (i.e., reading a terabyte of data from a modern hard drive takes about 3 hours). With a data-parallel approach, the system can split data across multiple disks and thus hide the latency of reading the data.

A BEK program is expressive: a programmer can use conditionals, switch statements, and registers—or local variables—in order to implement common string-manipulating programs. Unfortunately, this expressivity induces data dependencies, which are an obstacle to parallelism. The key contribution of this paper is an algorithm which automatically removes these data dependencies by mapping a BEK program into a intermediate format consisting of symbolic transducers, which extend classical transducers with symbolic predicates and symbolic assignments. We present a novel algorithm that we call *exploration* which performs *symbolic loop unrolling* of these transducers to obtain simplified versions of the original program. We show how these simplified versions can then be lifted to a *stateless* form, and from there compiled to data-parallel hardware.

To evaluate the efficacy of our approach, we demonstrate up to $8x$ speedups for a number of real-world, BEK programs, (e.g., HTML encoder and decoder) on data-parallel hardware. To the best of our knowledge, these are the first data parallel implementation of these programs. To validate that our approach is correct, we use an automatic testing technique to compare our generated code to the original implementations and find no semantic deviations.

POPL '15, January 15–17, 2015, Mumbai, India.

http://dx.doi.org/10.1145/2676726.2677014

Categories and Subject Descriptors D.3.1 [*Formal Definitions and Theory*]: Semantics; D.3.2 [*Language Classifications*]: Specialized application languages; F.1.1 [*Models of Computation*]: Automata; F.4.3 [*Formal Languages*]: Decision problems

Keywords Symbolic Automaton; Symbolic Trandsducer; Satisfiability Modulo Theories; Data-Parallel; State Space Exploration

1. Introduction

BEK is a popular DSL for string-manipulating programs [15]. Because of its expressiveness and popularity, we have chosen to use it for our experiments. While much of our results can be applied broadly to symbolic finite-state transducers, our focus is on providing correct and performant implementations of symbolic finite-state transducers; BEK gives us a platform for doing so. The reader can browse some BEK programs online at http://rise4fun.com/Bek or read the BEK tutorial [23] to build his or her intuition. Typical BEK programs are string encoders and decoders, security sanitizers, etc.

Why BEK: The advantage of a domain-specific language is that, unlike most general-purpose languages, many algebraic properties are *decidable* for basic BEK (BEK programs that do not use registers). In particular, one can check whether such BEK programs commute or are idempotent; one can compute the inverse of a BEK program, and given an output, compute all inputs that lead to it.

While previous work has focused on algorithms for reasoning about basic BEK programs [15, 32], they did not provide a clear strategy for either *running* BEK on a variety of platforms, thus hindering its adoption, or providing algorithms that first transform BEK programs to their basic analyzable form. These are important problems as BEK programs are useful tools that developers use to secure both client-side web browsers and Internet scale web services. The goal of this paper is to demonstrate how to compile a BEK program to their basic form. Then, as a concrete application we show how to compile BEK programs into scalable data-parallel code.

Previous work has shown that special state variables, called *registers* in BEK, are often needed to express common functionality (Section 3). Unfortunately registers introduce data dependencies that are obstacles to parallelism and also make many forms of BEK analysis undecidable. Manually removing registers requires developers to reason about complex state, which is error-prone and time-consuming.

Exploration: A key contribution of this work is a novel *exploration* algorithm that is a form of loop unrolling combined with symbolic function composition. Loops are unrolled symbolically while data dependencies exist between consecutive iterations. Such unrolling is achieved by introducing new states for finite projections of registers. In this process, several characters may be grouped together into tokens by folding some part of a register symbolically into labels, while another part is folded into concrete states. Intuitively,

grouping changes the granularity of what is to be considered as a single input element. For example, if we take a program such as a UTF8 decoder, then grouping would create tokens that correspond to subsequences between 1 and 4 bytes,[1] while for a Base64 decoder it would create subsequences of fixed length 4.

While the algorithm is not guaranteed to work in all cases because it is undecidable, e.g. it is easy to encode reachability of two counter machines which is undecidable [16], when it does work, it produces a transducer that is behaviorally equivalent to the original up to grouping of the input sequence, but *without* register variables. We have not found any typical BEK programs, such as sanitizers, encoders or decoders, for which the technique would fail to terminate. A case when it does not work is a loop that for example counts the number of elements in the input or, in general, has arbitrarily long dependencies between input elements.

Data parallelism: There are two key benefits to a BEK program after exploration. First, registers induce data dependencies which limit parallelism and thus exploration is a necessary first step in exploiting parallelism. Second, after exploration, by construction, a BEK program has little intermediate state (e.g., complexity of a program is pushed to the edges rather than encoded as states). The overhead of our data parallel implementation of a BEK program is linearly related to the size of the BEK program's intermediate state and thus exploration enables the automatic transformation from an expressive BEK program, with registers, into a data parallel one. This ultimately relieves the programmer from the burden of explicitly describing parallelism in a BEK program, an error-prone and difficult task.

1.1 Contributions

This paper makes the following contributions.

- Previous work on BEK introduced an extension to symbolic transducers with *registers* [32]. Use of registers is essential for modeling real sanitizers but makes their analysis difficult. In this paper, we present a novel *exploration* algorithm that is a form of symblic loop unrolling that works modulo *arbitrary decidable* background theories. The algorithm terminates for a class of transduces that only need a bounded lookahead and the algorithm outputs a new transducer that is equivalent to the input transducer but does not need registers.
- We integrated the exploration algorithm into a BEK to symbolic finite transducer (SFT) compiler. It has several interesting features. First, it uses an SMT solver as an oracle during the exploration algorithm, both for satisfiability checking and for model (witness) generation. Second, it combines exploration with *grouping*. Third, it uses the simplification mechanism present in modern SMT solvers for code generation of conditions and update functions.
- We demonstrate how exploration enables data parallelism. In particular, we show how to encode a BEK program, with infinite alphabets and registers, into one with a finite set of equivalence classes and no registers. Further, we demonstrate a data parallel implementation of this registerless intermediate form based on recent work in parallel finite state machines[25]. To the best of our knowledge, this is the first fully-automatic parallelization of string-manipulating code that combines advanced automata theory with state-of-the-art SMT technology to produce a parallel implementation.
- We have experimented with a number of BEK programs. We frequently observed exponential reductions in the number of states due to the application of our exploration algorithm. To validate the correctness of our translation, we used random testing of thousands of strings, observing no semantic differences when compared to independent C# serial implementations. For the range of hardware configurations we have experimented with, we observe a near-linear speedups as significant as $8x$.

[1] Assuming standard Unicode text processing where code points are limited to $\texttt{10FFFF}_{16}$, in which case 4 bytes are enough, otherwise longer encodings up to 6 bytes are possible.

```
private static string EncodeHtml(string t) {
  if (t == null) { return null; }
  if (t.Length == 0) { return string.Empty; }
  StringBuilder builder =
    new StringBuilder("", t.Length*2);
  foreach (char c in t) {
    if (((c>' ')&&(c<'{')) ||
        ((c>'@')&&(c<'[')) || (c=='_') ||
        ((c>'/')&&(c<':')) || (c=='.') ||
        (c==',') || (c=='-') || (c=='_')) {
        builder.Append(c);
      } else {
        builder.Append("&#" +
          ((int)c).ToString() + ";");
      }
    }
    return builder.ToString();
}
```

Figure 1. Code for `AntiXSS.EncodeHtml` version 2.0.

The background theories that were relevant for us during this work, were theories over linear arithmetic, bit-vectors, tuples, and lists. These theories were used to represent characters, tokens and strings. However, there were no dependencies on these particular theories. For example, one could also consider reals, rationals and arrays. Pragmatically speaking, any theory, or combination of theories, that is supported by an SMT solver works.

2. Background and Motivation

String-manipulating programs: Much of the practical motivation for analyzing string-manipulating routines comes from the need to analyze *security sanitizers* [15], such as the one shown in Figure 1. These are key in protecting against cross site scripting ("XSS") attacks, which plague today's web applications. These attacks happen because the applications take data from untrusted users, and then echo this data to other users of the application. Because web pages mix markup and JavaScript, this data may be interpreted as code by a browser, leading to arbitrary code execution with the privileges of the victim.

The first line of defense against XSS is the practice of *sanitization*, where untrusted data is passed through a *sanitizer*, a function that escapes or removes potentially dangerous strings. Multiple widely used Web frameworks offer sanitizer functions in libraries, and developers often add additional custom sanitizers due to performance or functionality constraints.

Sanitizers are typically a small amount of code, perhaps tens of lines. Furthermore, application developers know when they are writing a new, custom sanitizer or set of sanitizers. Experience has shown that if developers are willing to spend a little more time on sanitizers, they can obtain fast and precise analyses of sanitizer behavior, along with actual sanitizer code ready to be integrated into both server- and client-side applications. Our approach here is BEK, a language for modeling string transformations. The language is designed to be (a) sufficiently expressive to model real-world code, and (b) sufficiently restricted to allow fast, precise analysis, without needing to approximate the behavior of the code.

Unfortunately, implementing sanitizers *correctly* is surprisingly difficult. Anecdotally, in dozens of code reviews performed across various industries, just about any custom-written sanitizer was

flawed with respect to security. SANER, for example, shows flaws in custom-written sanitizers used by ten web applications [2]. Incorrect sanitization may be worse than no sanitization by *enabling* rather than *disabling* JavaScript execution [4, 21].

The problem becomes even more complicated when considering that a web application may *compose* multiple sanitizers in the course of creating a web page. In a recent empirical analysis, it has been found that a large web application often applied the same sanitizers twice, despite these sanitizers not being idempotent [29]. This analysis also found that the order of applying different sanitizers could vary, leading to questions about whether composition of sanitizers is commutative.

BEK has been previously used to perform security-specific analyses of sanitizers. For example, one can use BEK to determine whether there exists an input to a sanitizer that yields any member of a publicly available database of strings [27] (XSS Cheat Sheet) known to result in cross site scripting attacks. Our analysis is fast in practice; for example, we take two seconds to check the commutativity of the entire set of Internet Explorer 8 XSS filters, and less than 39 seconds to check an implementation of the `HTMLEncode` sanitization function against target strings from the XSS Cheat Sheet [15].

Beyond strings: BEK has been demonstrated as being useful for applications beyond security sanitization. Other uses have been proposed, as diverse as image blurring and geo-location privacy, which fall outside the scope of this paper. The interested reader can see [32] for more details.

Performance: To realize why performance is a concern in this context, consider large-scale distributed data processing in the cloud. Encoding is paramount in Web applications, especially with large volumes of data. Google Docs or Office Web Apps, for example, store multi-megabyte files from *millions* of users. XSS caused by insufficient data sanitization has been discovered in these large-scale cloud apps before. [2] To avoid XSS, these services must encode all user files before displaying them on the web.

3. Symbolic Transducers

The semantics of BEK programs are given by symbolic transducers. The goal of this section is to recap the BEK language in Section 3.1. In Section 3.2, we develop a formal treatment required to precisely express these semantics. Pragmatically-inclined readers may wish to save Section 3.2 for later reading.

3.1 Intuition

We briefly describe the BEK domain-specific language for writing string transformations such as shown in Figure 1. An example BEK program is in Figure 2. As introduced in [15], the core construct in BEK is iteration over each character in an input string. Programs can then have case statements that describe different behavior for different input characters. Typically the program will perform some local computation, then *yield*, or output, a new character. A new string is built up based on the characters of the input string.

Registers: Programs can also have *register variables* that keep state during the iteration. Figure 2 shows a sample BEK program that checks each character and then updates a register variable r. Depending on input character, the program may then output the contents of the register, or it may simply pass through the character unchanged. An `end case` statement takes care of any remaining final output suffix after the end of the input has been reached For more details on BEK we refer to previous work or to the online BEK evaluator [23]. While the language is limited, it still is expressive enough to capture a wide range of string-manipulating functions, including many of the functions commonly used in Web sanitization and functions used in graphics processing [15, 32].

[2] The reader may consult these links for examples:
https://www.cogmotive.com/blog/office-365-tips/vulnerability-in-office-365-allows-unauthorised-administrator-access,
http://xs-sniper.com/blog/2008/04/14/google-xss/,
http://ha.ckers.org/blog/20070617/another-google-xss-in-google-documents/.

```
function E(x)=
  ite(x<=25,x+65,
    ite(x<=51,x+71,
      ite(x<=61,x-4,ite(x==62,'+','/'))));

program b64e(input){
  return iter(x in input)[s:=0;r:=0;]{
    case(x>0xFF):raise ERROR;
    case(s==2):yield(E((r|(x>>6))), E(x&0x3F));
      s:=0; r:=0;
    case(true):yield(E(ite(s==0,x>>2,r|(x>>4))));
      r:=ite(s==0,(x&3)<<4,(x&0xF)<<2); s:=s+1;
    end case(s==1):yield (E(r),'=','=');
    end case(s==2):yield (E(r),'=');
    end case(true):yield (); //may be omitted
  };
}
```

Figure 2. BEK program for Base64 encoding.

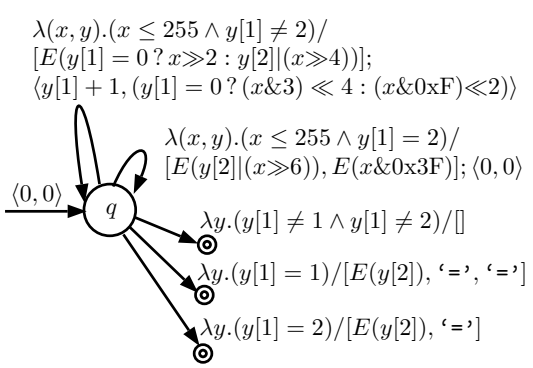


Figure 3. ST of the program `b64e` in Figure 2.

ST representation: The semantics of the program in Figure 2 is captured by a *symbolic transducer* in Figure 3. Symbolic transducers are a generalization of classic finite transducers, which allow arbitrary underlying label theories [32].

Exploration and Data-Parallelism: The register variables in BEK are important for making it easy to translate string-manipulating functions written in C# or other languages to BEK, because existing functions typically keep state through an iteration. These variables are also convenient for writing functions in BEK directly. Unfortunately, these register variables are enemies of data-parallelism, because they introduce control flow that depends on the registers and not on the individual character.

3.2 Formalism

We now formally define symbolic transducers or STs and give examples of how STs capture behavior of programs. We assume a *background structure* that has an effectively enumerable *background universe* $\boldsymbol{U}$, and is equipped with a language of function and relation symbols with fixed interpretations. Definitions below are given with $\boldsymbol{U}$ as an implicit parameter. We assume closure under Boolean operations and equality. Operations that are specific to $\boldsymbol{U}$ do not affect the results. We use λ-expressions for dealing with

anonymous functions that we call λ-*terms*. In general, we use standard first-order logic and follow the notational conventions that are consistent with [32]. The universe is multi-typed with $\boldsymbol{\tau}$ denoting the subuniverse of elements of type τ. We make use of the *empty tuple type* $\mathsf{T0}$ such that $\mathbf{T0} = \{\langle\rangle\}$. We use a variant of the definition of STs where the control state component of STs is explicit. We write $\Psi(\sigma)$ for the set of σ-predicates. We write $\Lambda(\sigma \to \gamma)$ for λ-terms f that denote functions $\boldsymbol{f} : \boldsymbol{\sigma} \to \boldsymbol{\gamma}$. We write $\Lambda(\sigma \to \gamma)^*$ for $\bigcup_{k \geq 0} \Lambda(\sigma \to \gamma^k)$. In other words, for any λ-term $f \in \Lambda(\sigma \to \gamma)^*$ the length of the output sequence may not depend of the input, but must be fixed.

Definition 1: A *Symbolic Transducer* or *ST* with input type σ output type γ and register type τ is a tuple $A = (Q, q^0, r^0, R)$, where Q is a finite set of *states*; $q^0 \in Q$ is the *initial state*; $r^0 \in \boldsymbol{\tau}$ is the *initial register value*; R is a finite set of *rules* $R = \Delta \cup F$ where $\Delta \subseteq (Q \times \Psi(\sigma \times \tau) \times \Lambda(\sigma \times \tau \to \gamma)^* \times \Lambda(\sigma \times \tau \to \tau)$ is a set of *transitions* and $F \subseteq (Q \times \Psi(\tau) \times \Lambda(\tau \to \gamma))$ is a set of *finalizers*. ⊠

A transition (q, φ, o, u, p) is also written $q \xrightarrow{\varphi/o;u} p$ where q is the *start* state, φ the *guard*, o the *output*, u the *update*, and p the *end* state. A finalizer (q, φ, o) is also written $q \xrightarrow{\varphi/o} \circledcirc$ with q as the state, φ as the guard, o as the output. Finalizers generalize final states.

Example 1 Let $\sigma = \gamma$ be unsigned bitvectors (or natural numbers) N and let τ be the type N × N. The operation '≪' is shift-left, '≫' is shift-right, '&' is bit-wise-AND, '|' is bit-wise-OR. The ST in Figure 3 has the above types and is the direct mapping of the BEK program in Figure 2 where $s = y[1]$ and $r = y[2]$. $Q = \{q\}$, $r^0 = \langle 0, 0\rangle$, and the ST has two transitions and three finalizers. It computes Base64 encoding of byte sequences. Base64 is a standard encoding that is used to transfer binary data over textual media. ■

The semantics of A is given by the following concrete transition relation. Let $q, p \in Q$, $r \in \boldsymbol{\tau}$, $a \in \boldsymbol{\sigma}$. Then $(q, r) \xrightarrow{[a]/\boldsymbol{o}(a,r)}_A (p, \boldsymbol{u}(a, r))$ denotes that there exists a transition $q \xrightarrow{\varphi/o;u} p$ such that $\varphi(a, r)$ holds. Similarly, $(q, r) \xrightarrow{\epsilon/\boldsymbol{o}(r)}_A \circledcirc$ denotes that there exists a finalizer $q \xrightarrow{\varphi/o} \circledcirc$ such that $\varphi(r)$ holds.

Now, the *reachability relation* $p \xrightarrow{\bar{a}/\bar{b}}\!\!\!\!\rightarrow_A p'$ for $\bar{a} \in \boldsymbol{\sigma}^*$, $\bar{b} \in \boldsymbol{\gamma}^*$, and $p, p' \in (Q \times \boldsymbol{\tau}) \cup \{\circledcirc\}$ is defined through the closure under the following conditions, where '·' is concatenation of sequences, note that $\epsilon \cdot \bar{x} = \bar{x} \cdot \epsilon = \bar{x}$:

- If $p \xrightarrow{\bar{a}/\bar{b}}_A p'$ then $p \xrightarrow{\bar{a}/\bar{b}}\!\!\!\!\rightarrow_A p'$.
- If $p \xrightarrow{\bar{a}/\bar{b}}\!\!\!\!\rightarrow_A p_1 \xrightarrow{\bar{a}'/\bar{b}'}_A p_2$ then $p \xrightarrow{\bar{a}\cdot\bar{a}'/\bar{b}\cdot\bar{b}'}\!\!\!\!\rightarrow_A p_2$.

Definition 2: The *transduction* of A, denoted $\mathscr{T}_A$, is the following function from $\boldsymbol{\sigma}^*$ to $2^{(\boldsymbol{\gamma}^*)}$:

$$\mathscr{T}_A(\bar{a}) \stackrel{\text{def}}{=} \{\bar{b} \mid (q_A^0, r_A^0) \xrightarrow{\bar{a}/\bar{b}}\!\!\!\!\rightarrow_A \circledcirc\}$$

A is *single-valued* when $|\mathscr{T}_A(\bar{a})| \leq 1$ for all $\bar{a} \in (\boldsymbol{\sigma})^*$ and A is *deterministic* when, for all $\bar{a}, \bar{b}_1, \bar{b}_2, p, p_1, p_2$, if $p \xrightarrow{\bar{a}/\bar{b}_1}_A p_1$ and $p \xrightarrow{\bar{a}/\bar{b}_2}_A p_2$ then $\bar{b}_1 = \bar{b}_2$ and $p_1 = p_2$. We write $\mathscr{T}_A(\bar{a}, \bar{b})$ for $\bar{b} \in \mathscr{T}_A(\bar{a})$. ⊠

Finalizers can be omitted at the expense of allowing nondeterminism. However, nondeterminism is undesired because determinization of STs is in general not possible and analysis of (or even code generation from) nondeterministic STs is very difficult.

It is easy to show that determinism implies single-valuedness. Deterministic STs form a practically important subclass of STs and in the examples and case studies we only consider deterministic STs. For the data-parallel translation explained in Section 5 the STs are required to be deterministic, that is naturally the case for the kinds of string transformations we have in mind with this approach.

Example 2 Let A be the ST in Figure 3. Let $y = \langle 0, 0\rangle$ and $x = \texttt{'A'} = 1000001_2$. So $x \leq 255$ and $y[1] \neq 2$. We have $E(x \gg 2) = E(10000_2) = E(16) = 16 + 65 = \texttt{'Q'}$ and $((x \& 3) \ll 4) = (1 \ll 4) = 10000_2 = 16$, so there is a concrete transition

$$(q, \langle 0, 0\rangle) \xrightarrow{[\texttt{'A'}]/[\texttt{'Q'}]} (q, \langle 1, 16\rangle)$$

If we do one more step from configuration $(q, \langle 1, 16\rangle)$ with input `'B'` we get the concrete transition

$$(q, \langle 1, 16\rangle) \xrightarrow{[\texttt{'B'}]/[\texttt{'U'}]} (q, \langle 2, 8\rangle)$$

Suppose that the input sequence ends here. Then we use the last finalizer that gives us the concrete transition:

$$(q, \langle 2, 8\rangle) \xrightarrow{\epsilon/[\texttt{'I'},\texttt{'='}]} \circledcirc$$

By using the derived reachability relation we have

$$(q, \langle 0, 0\rangle) \xrightarrow{[\texttt{'A'},\texttt{'B'}]/[\texttt{'Q'},\texttt{'U'},\texttt{'I'},\texttt{'='}]}\!\!\!\!\rightarrow \circledcirc$$

Thus, $\mathscr{T}_A(\texttt{"AB"}) = \{\texttt{"QUI="}\}$. ■

4. Compilation of STs to SFTs

We assume that an ST is given. The ST may for example be the result of a translation from a BEK program. Sample BEK program and corresponding ST are illustrated in Figure 2 and Figure 3. Our goal is to compile the ST into an SFT (if possible), where an SFT is an ST *without* registers, or formally, whose register type τ is $\mathsf{T0}$. In the SFT we may group some characters into *combined characters* or *tokens* so that, if the input type of the ST is σ then the input type of the SFT is $\sigma^{\leq k}$ for some $k \geq 1$ (sequences over σ of length l, $1 \leq l \leq k$). In our compilation we maintain the following equivalence between the given *ST* and the resulting *SFT*. We use *flatten*$([u_i]_{i=1}^n)$, where u_i has type $\sigma^{\leq k}$, to denote the sequence $u_1 \cdot u_2 \cdots u_n$ in $\boldsymbol{\sigma}^+$.

Definition 3: *ST* and *SFT* are *equivalent modulo grouping* if $\forall \bar{a}, \bar{b}$, $\mathscr{T}_{ST}(\bar{a}, \bar{b}) \Leftrightarrow \exists \bar{u} : \bar{a} = \mathit{flatten}(\bar{u}), \mathscr{T}_{SFT}(\bar{u}, \bar{b})$. ⊠

In other words, the outputs of *ST* and *SFT* must be equal for some grouping of the input characters in *SFT*. The compiler uses two phases, *grouping* and *exploration*. The phases are interleaved into a single algorithm, but intuitively, they serve two distinct purposes:

Grouping: The idea is as follows. Two consecutive transitions $p \xrightarrow{\varphi/o;u} q' \xrightarrow{\varphi'/o';u'} q$ can be lifted into a single transition that reads more inputs in one atomic step

$$p \xrightarrow{\begin{array}{l}\lambda(x,y).\varphi_1(x[1],y) \wedge \varphi'(x[2], u(x[1],y))/\\ \lambda(x,y).o(x[1],y) \cdot o'(x[2], u(x[1],y));\\ \lambda(x,y).u'(x[2], u(x[1],y))\end{array}} q$$

where $x[i]$ reads the i'th element from x (suppose first element has index 1). Thereby, the state q' becomes eliminated (if this is the only occurrence of q'). The lifted transition may also become *unsatisfiable*, in which case it is eliminated. The input character of the new transition is a pair of the original characters.

Exploration: Exploration starts from the initial configuration (q^0, r^0) where q^0 is the initial state and r^0 is the initial register value of the ST. If an update to a register is a fixed value r that is *independent* of the input values in a given target state q of the ST, then the configuration (q, r) is treated as a state

```
#algo.py
from z3 import *
Z = Solver()

class ST: #encapsulates a symbolic transducer
  def __init__(self, q0, r0, T, F, x, y, ax):
    self.q0 = q0          #initial state
    self.r0 = r0          #initial register
    self.T = T            #transitions
    self.F = F            #finalizers
    self.x = x            #label variable
    self.y = y            #register variable
    self.ax = ax          #background axioms

  def Delta(self,q):      #transitions from q
    for t in self.T:
      if t[0] == q: yield t

  def Fin(self,q):        #finalizers from q
    for f in self.F:
      if f[0] == q: yield f

def IsSat(f):
  Z.push();Z.add(f);res=Z.check();Z.pop();
  return res != unsat

def Choose(f, t):
  Z.push();Z.add(f);Z.check();
  v = Z.model().evaluate(t, True);
  Z.pop();return v

def GetUniqueValue(st, phi, t, k):
  x_ = lambda i:Const('x%d'%(i+1),st.x.sort())
  z_ = lambda i:Const('z%d'%(i+1),st.x.sort())
  theta = [(x_(i),z_(i)) for i in range(k)]
  t1 = substitute(t, *theta)
  phi1 = substitute(phi, *theta)
  if IsSat(And(phi,phi1,t!=t1)): return None
  else: return Choose(phi, t)
```

```
def Group(st, q, r, k, phi, out):
  for (qS, G, O, U, qE) in st.Delta(q):
    xk = Const('x%d'%k, st.x.sort())
    theta = [(st.x, xk),(st.y, r)]
    c1 = And(phi, substitute(G, *theta))
    if IsSat(c1):
      o1 = [substitute(o, *theta) for o in O]
      r1 = substitute(U, *theta)
      v1 = GetUniqueValue(st, c1, r1, k)
      if v1 != None:
        yield (c1, out+o1, k, qE, v1)
      else:
        for tr in Group(st,qE,r1,k+1,c1,out+o1):
          yield tr
  for (qF, G, O) in st.Fin(q):
    cF = And(phi, substitute(G,(st.y, r)))
    if IsSat(cF):
      oF = [substitute(o,(st.y,r)) for o in O]
      yield (cF, out+oF, k-1, None, None)

def Explore(st): #Exploration algorithm
  Z.add(st.ax) #add the axioms to the solver
  W,sft = {0},[]
  nextStateId = 1
  stateMap = {0 : (st.q0, st.r0)}
  stateIdMap = {(st.q0, str(st.r0)) : 0}
  while len(W) > 0:
    qS = W.pop()
    (q, r) = stateMap[qS]; tt = BoolVal(True)
    for (g,o,k,q1,r1) in Group(st,q,r,1,tt,[]):
      qE = None
      if q1 != None:
        if (q1,str(r1)) in stateIdMap:
          qE = stateIdMap[(q1, str(r1))]
        else:
          qE = nextStateId; nextStateId += 1
          stateMap[qE] = (q1,r1)
          stateIdMap[(q1, str(r1))] = qE
          W.add(qE)
      sft.append((qS, g, o, k, qE))
  return sft
```

Figure 4. ST exploration algorithm in Z3 Python.

of the explored SFT and the configuration (if new) is pushed to the search stack for continued exploration. If an update does not have a fixed value then grouping is invoked in an attempt to localize the use of the register update.

The overall effect is that registers which are used to encode local dependencies between consecutive characters are redundant and thus can be eliminated. We refer to the full algorithm also as *exploration* when it is clear from the context that the combined algorithm is meant. The complete self-contained algorithm is given by the (executable) Python script in Figure 4. Although the algorithm is given in a concrete executable form in Python (using the Z3 module) it is virtually identical to an abstract mathematical formulation and we have therefore decided to present only the concrete version, although it requires a bit of extra effort to get used to the notational differences compared to the formalism above. We illustrate the exploration algorithm in Example 3, where, for explanatory purposes, grouping and exploration are presented as two separate phases. The function `GetUniqueValue`, called on line 49, shows how Z3 is used to decide if the register is going to have a concrete value, i.e., if the term r_1 representing the value is constant with respect to the constraint c_1. If not then grouping is invoked recursively.

Example 3 Take the ST in Figure 3. Project τ into $\tau_1 = \mathrm{N}$ as the first element $y[1]$ of y and $\tau_2 = \mathrm{N}$ as the second element $y[2]$ of y. This projection happens automatically in the combined exploration algorithm. Exploration will now partially evaluate all the rules, by using depth first search, and integrate $y[1]$ into the states $q_{y[1]}$ in

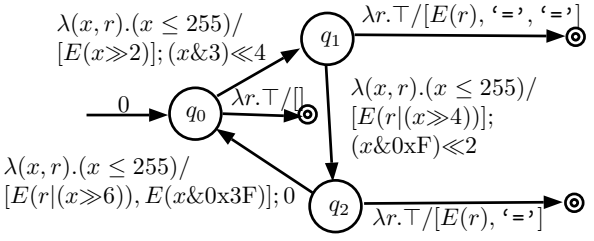


Figure 5. ST after exploration of ST in Figure 3.

such a way that a rule $q \xrightarrow{\lambda(x,y).\ell(x,y)} q$ is replaced by one or more rules of the form $q_a \xrightarrow{\lambda(x,z).\ell_{ab}(x,z)} q_b$ where $a, b \in \boldsymbol{\tau_1}$ and the register z has type τ_2. The main technical insight is that it is possible to compute such b and ℓ_{ab} for the given a through *finite* case analysis over x even though σ, γ and τ_2 may be infinite.

If we apply exploration to the ST in Figure 3 we get the equivalent ST in Figure 5. We can now apply grouping to this ST. To illustrate grouping, consider the path $q_0 \xrightarrow{f} q_1 \xrightarrow{g} q_2 \xrightarrow{h} q_0$ of transitions. First apply grouping to $q_0 \xrightarrow{f} q_1 \xrightarrow{g} q_2$ to obtain $q_0 \xrightarrow{f\circ g} q_2$ and then we apply grouping to $q_0 \xrightarrow{f\circ g} q_2 \xrightarrow{h} q_0$ to obtain $p_0 \xrightarrow{f\circ g\circ h} p_0$. At this point the composed label $f \circ g \circ h$ be-

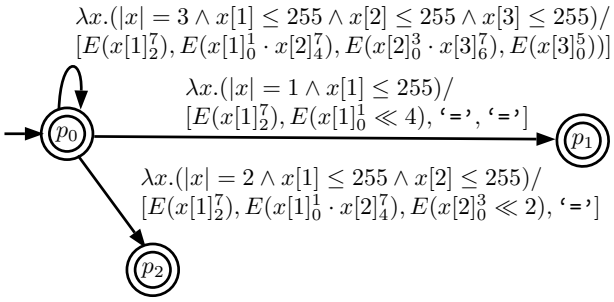


Figure 6. SFT after grouping of ST in Figure 5; y_n^m denotes extraction of bits from m to n from y; $y \cdot z$ denotes bit-append.

comes independent of the register r because r is only used to glue f with g and g with h.

A similar step is applied to transitions followed by finalizers to get composed transitions that lead from p_0 to *sink* states p_1 and p_2 (states without outgoing transitions and with trivial finalizers that yield empty outputs). Finally, all resulting rules are independent of the register and so the register can be eliminated and we have constructed an SFT illustrated in Figure 6. An important point is that the construction of the SFT would not be possible by only using grouping or only using naive exploration. The new combined exploration algorithm is needed. ■

The first point we address next is: *The reason we are interested in SFTs.* The second point is: *When does the exploration algorithm terminate?* The third point is: *Does the algorithm meet our needs for parallelization?* As it turns out, the third question needs more work – because, except for some special cases when the lookahead is fixed, the assumption that the input is tokenized up front is not fully realistic in a parallel setting because then there is no way to calculate where the next token starts without having processed the previous tokens.

4.1 Succinctness

Figure 7 compares the sizes of the state machines needed to achieve the different encoding and decoding tasks.

	Size		
Encoder	**SFT**	**SFT$^+$**	**k-SFT**
`UTF8encode`	16	6	9
`UTF8decode`	6371	10	27
`BASE64encode`	105	6	14
`BASE64decode`	1161	6	23
`HTMLencode`	3	3	7
`HTMLdecode`	113	10	31

Figure 7. Exploration sizes in total nr of transitions. SFT is naive exploration. SFT$^+$ is grouped exploration. k-SFT is creation of k-SFT after SFT$^+$.

The table indicates the advantages of using the exploration algorithm. Without this dramatic reduction in the size of the SFT, we would not be able to compile the SFT to exploit data parallel finite state machines presented below, where the approach requires that the number of states in the transducer is *small*, which our compilation scheme provides, while the label theory may potentially allow infinite character domains. In particular, it is shown in [32, Figure 8(a)], that in HTML decoding, the number of states in the SFT grows exponentially in the length of k for supporting decoding of patterns $\&\#[0-9]\{k\}$; that makes naive exploration to SFTs impractical in this case. This exponential blowup is avoided with the new exploration algorithm. The final column in the table shows the size of the k-SFT obtained after the final phase of the algorithm discussed below.

Definition 4: A *k-SFT* has a finite state space Q and has transitions $p \xrightarrow{\varphi/f} q$ where $p, q \in Q$, φ is a σ-predicate, and $f \in \Lambda(\sigma^m \to \gamma^n)$ is a λ-term for some fixed $m \geq 0$ and $n \geq 0$. We assume that the k-SFT is *deterministic* meaning that if there are two transitions $p \xrightarrow{\varphi_1/f_1} q_1$ and $p \xrightarrow{\varphi_2/f_2} q_2$ such that $\varphi_1 \wedge \varphi_2$ is satisfiable, then $f_1 = f_2$ and $q_1 = q_2$. A k-SFT has also finalizers of the form $q \xrightarrow{f} \circledcirc$ with f as above and where (to maintain determinism) there is at most one finalizer for each state q. The case $m = 0$ means that there is no input dependence, so f is a fixed output sequence. ⊠

The semantics of a transition $p \xrightarrow{\varphi/f} q$ is that φ is applied in state p to the current character in the input sequence $\bar{c}$, say c_i. There is an implicit well-definedness criterion that, if the state is p and $\varphi(c_i)$ is true then $i \geq m$, where m is the arity of f. If $\varphi(c_i)$ holds then f is applied to $(c_{i-m+1}, c_{i-m+2}, \ldots, c_{i-1}, c_i)$ to produce the next subsequence of the output. For finalizers the semantics is that f is applied to $(c_{l-m+1}, c_{l-m+2}, \ldots, c_{l-1}, c_l)$ to produce the suffix of the output, where c_l is the last character.

4.2 Termination and Limitations

The exploration algorithm does not terminate if there are unbounded dependencies between successive characters in the input. In other words, if the register is needed to remember some input element for arbitrarily many following input elements. For example, if the register is used to sum all "digits" seen in the input, then the exploration algorithm does not terminate.

An *m-grouping* of a nonempty sequence $\bar{a}$ over $\boldsymbol{\sigma}$ is a sequence $\bar{u}$ over $\boldsymbol{\sigma}^{\leq m}$ whose flattened form equals $\bar{a}$. For example, the sequence $[[1,2],[3,4,5],[6]]$ is a 3-grouping of the sequence $[1,2,3,4,5,6]$.

Definition 5: ST A has the *bounded lookahead property* if there is a finite subset $P \subseteq Q \times \boldsymbol{\tau}$, $p_0 = (q^0, r^0) \in P$, and there exists $m > 0$ such that for all input sequences $\bar{a}$ over $\boldsymbol{\sigma}$ accepted by A there exists an m-grouping $[u_i]_{i=1}^n$ of $\bar{a}$ such that $p_{i-1} \xrightarrow{u_i/} p_i$ for some $p_i \in P$ for $i < n$ and $p_n = \circledcirc$. ⊠

Theorem 1: *Exploration algorithm terminates iff the ST has the bounded lookahead property. The output SFT is equivalent modulo grouping to the input ST.*

We write *SFT$^+$* for the intermediate result that the exploration algorithm produces. Namely, SFT$^+$ stands for the SFT whose input characters have been grouped, i.e., whose character type is $\sigma^{\leq m}$, for some $m > 0$, where σ is the character type of the input SFT.

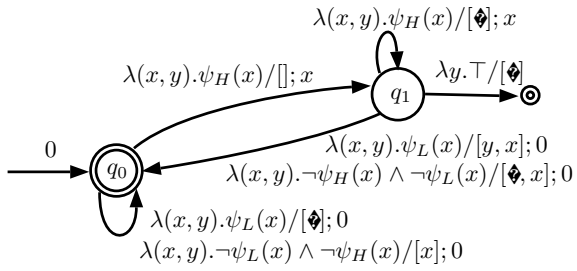
There is a class of STs for which the exploration algorithm does not terminate although an equivalent k-SFT exists. Roughly speaking, this situation arises when *bounded lookback* cannot be turned into *bounded lookahead*. Thus, even if the ST could, in principle, be compiled into an equivalent k-SFT, there exists no intermediate SFT$^+$ over grouped characters.

The following example illustrates such a case.

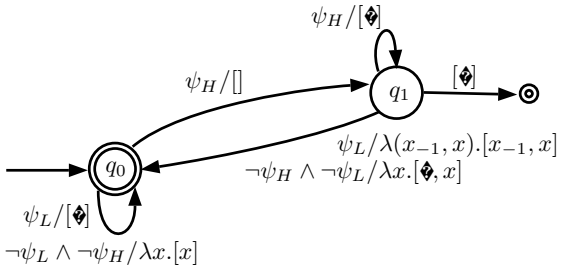
Example 4 Consider standard `HtmlEncode` over full Unicode. An input to the encoder is a UTF-16 encoded string. An input is *malformed* if it contains a *bad surrogate* b:

- b is a high surrogate (in range $\psi_H \stackrel{\text{def}}{=} \lambda x.55296 \leq x \leq 56319$) that is not immediately followed by a low surrogate (in range $\psi_L \stackrel{\text{def}}{=} \lambda x.56320 \leq x \leq 57343$), or
- b is a low surrogate that is not immediately preceded by a high surrogate.

Some encoders use the following *correcting rule* as a robustness feature rather than rejecting malformed strings (e.g., `HtmlEncode` in `System.Net.WebUtility` in .NET Framework 4.5). The Unicode replacement character `&#xFFFD;` (`\uFFFD`) is used to replace bad surrogates so that the string is not malformed. Such a correcting phase can be described by the ST in Figure 8(a), say `Rep`. For efficiency,



(a) Using an ST



(b) Using a 2-SFT

Figure 8. Replacing bad surrogates with `&#xFFFD;`.

existing hand-optimized encoders integrate `Rep` into the logic for the encoder, but here we think of it as a separate preprocessing step over the input. In some versions of `HtmlEncode` the replacement character `&#xFFFD;` itself is also encoded, in others it is not.

The difficulty with `Rep` is that there is no fixed length lookahead window to decide, if the last character in the window is the last consequtive high surrogate. However, the 2-SFT in Figure8(b) is equivalent to `Rep` and is parallelizable with the approach described below. The variable x_{-1} refers to the previous character in the input while x refers to the current character, so the lookback window size for determining the output is 2. ■

To be completely accurate, the exploration algorithm *does* terminate for the `Rep` ST in Figure 8(a) because there are a finite number of high surrogates (1024). But this is not the point. The point is that the exploration creates a state space that is unreasonably large. Moreover, if $\psi_H(x)$ and $\psi_L(x)$ were defined as $(x \bmod 2 = 0)$ and $(x \bmod 2 = 1)$ then the algorithm would not terminate because `Rep` would not have the bounded lookahead property: suppose there exists m and P as in Definition 5 and consider the input $[c]_{i=1}^{m}$ for some c such that $\psi_H(c)$ holds. Then $(q_1, c) \in P$, but there are infinitely many such c, contradicting that P is finite.

It is an open problem how to decide such cases. The particular case of `Rep` in Figure 8(a) seems easy to detect, but consider the small variant of `Rep` where the q_1-loop is replaced by $q_1 \xrightarrow{\psi_H(x)/[]} q_1$, i.e., the intermediate high surrogates are ignored. Then there exists no fixed lookback $k+1$ for the symbol x_{-k} in the corresponding transition from q_1 to q_0 in the $(k+1)$-SFT in Figure 8(b).

The way we deal with this problem is by using composition. The encoder STs we consider, are defined under the assumption that the input string is valid, i.e., it has no bad surrogates. The exploration algorithm is used to produce a SFT$^+$ that is then transformed into an equivalent k-SFT, say M. Finally, M is *composed* with `Rep` (the 2-SFT in Figure 8(b)) to produce $M' = \lambda x.M(\texttt{Rep}(x))$. Although k-SFTs are, in general, not closed under compositon, in this particular case they are. So the actual input to parallelization is M'. Not only is this easier to program, but it is also less error prone because the corner cases of what to do with bad surrogates are completely avoided. Moreover, there is no loss in efficiency because the two transformations are automatically composed into one, essentially eliminating the intermediate corrected string, as a form of deforestation [34].

Example 4 illustrates a case when it is not possible to construct *SFT*$^+$ from *ST* because there is no grouping bound, although an equivalent *k-SFT* exists. Another difficulty is that it is not always possible to turn an *SFT*$^+$ into a *k-SFT* (that is required for parallelization unless all groups have a fixed length). As discussed below, such a transformation requires *monadic decomposition* of predicates [33] in order to decompose n-ary relations used as guards in the *SFT*$^+$ into Boolean combinations of monadic relations that will guard the transitions in the *k-SFT*. In general this is not possible, e.g., if the predicate is $x = y$. Moreover, the problem of deciding if such a decomposition exists is, in general, undecidable even when assuming (as we do) that the theory over labels is decidable. The latter undecidability follows from a result in [20, Proposition 2.b].

4.3 Parallelizability

The exploration algorithm presented above transforms an ST into an SFT at the expense of grouping several input characters into one token. Thus, formally the input language of the SFT is a nested sequence where the inner sequences or character groups are tokens. If all tokens have the same length k, as in Base64 decoding where $k = 4$, then it is possible to process the original input in parallel by reading the input tokens from offsets km for $m > 0$. In general, as for example with HTML encoding or UTF8 encoding, the tokens have variable length and this simple solution does not work. Also, storing inputs in registers breaks parallelizability and takes us back to square one.

Instead, we transform the SFT$^+$ into a k-SFT with *lookback* building on the idea of k-SLTs introduced in [7]. The algorithm that we use does not introduce registers and maintains the property that transition guards only reference the current character, but output functions may refer back to k prior input characters. The resulting k-SFT is a special kind of a k-SLT. The main benefit is that the transition graph of the domain of a k-SFT is an SFA over the original characters. This enables the parallelization approach presented in Section 5 and the output computation still works because the lookback is bounded and can be applied in parallel.

We illustrate the algorithm below using an example. The input to the algorithm is an SFT over tokens or subsequences of characters. This SFT is first decomposed so that all transitions are split into predicates over singleton characters. To achieve this, the algorithm uses *monadic decomposition* of predicates from [33]. Then the split transitions are merged back together into a deterministic k-SFT. Finally, the k-SFT is compiled into C# code that is used by the subsequent parallelization step.

Example 5 We consider here a cut down version of a UTF8 decoder to illustrate k-SFT construction. Suppose that the SFT$^+$ produced by exploration has a single state q_0 and three transitions: $q_0 \xrightarrow{\psi/\lambda x.[x]} q_0$, $q_0 \xrightarrow{\varphi/f} q_0$, and $q_0 \xrightarrow{\gamma/g} q_0$, where

- ψ reads *one* byte and checks that it is in the ASCII range and the character is output as is,
- φ reads *two* bytes and checks that they form a valid 2-byte encoding, f computes the code point from the two bytes,

- γ reads *three* bytes and checks that they form a valid 3-byte encoding, g computes the code point from the three bytes.

The first transition is kept as is. The second and the third transitions are split into transitions that read one character at a time and the output calculation is delayed to the last transition. In order to do so, the predicates are first decomposed by using monadic decomposition. The decomposed forms of the predicates are as follows (using hexadecimal notation for numbers and interval notations)

$$\begin{aligned} dec(\varphi(\bar{x}))&=x_1 \in \texttt{C2-DF} \wedge x_2 \in \texttt{80-BF} \\ dec(\gamma(\bar{x}))&=(x_1 = \texttt{E0} \wedge x_2 \in \texttt{A0-BF} \wedge x_3 \in \texttt{80-BF}) \vee \\ &\quad (x_1 = \texttt{ED} \wedge x_2 \in \texttt{80-9F} \wedge x_3 \in \texttt{80-BF}) \vee \\ &\quad (x_1 \in \texttt{EE|EF|E1-EC} \wedge x_2 \in \texttt{80-BF} \wedge x_3 \in \texttt{80-BF}) \end{aligned}$$

The k-SFT construction algorithm introduces intermediate states for each transition after reading each symbol, considering one disjunct at a time in the DNF. It then merges the transitions into a single deterministic k-SFT. In this case the result is a 3-SFT shown in Figure 9. There would be additional states for the case of 4 byte

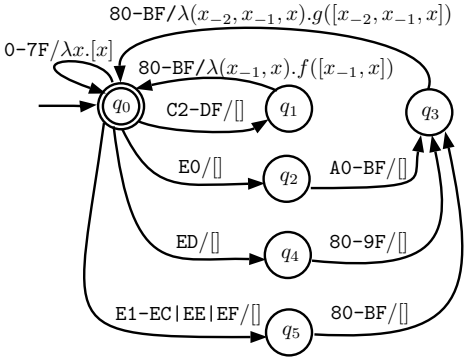


Figure 9. 3-SFT constructed from a UFT8 decoder that decodes up to three byte encodings.

encodings resulting in a 4-SFT, and more states and larger lookback if even longer encodings (up to 6 bytes) were allowed. ■

5. Data-Parallel Translation

In the prior sections we demonstrated how to remove registers from ST and turn them into k-SFTs. In this section we describe how to run k-SFTs on data-parallel hardware. In particular, we demonstrate an end to end compilation of k-SFTs to a data parallel version capable of exploiting multiple cores via threads.

Here we frame the evaluation of finite state transducers as associative operations over vectors and matrices. Because these operations are associative, they can take advantage of data-parallel hardware. The number of states in the SFT must be small for efficiency. Our key insight that allows us to combine STs with the data-parallel approach is that we first transform the ST into a k-SFT, using the pipeline of algorithms discussed above. In effect, ST to k-SFT compilation pushes the complexity of the ST into the edges, which in turn allows us to efficiently target data-parallel hardware.

In the sections that follow, we demonstrate an automatic approach that compiles a BEK program into a SFT and then down into the data parallel formulation that runs on multicore hardware.

5.1 Data-Parallel Operators

To aid our discussion, we introduce two higher-order data-parallel primitives. First, `zip` takes a binary function and maps that function over two sequences of equal sized length. For example, to pairwise add the numbers in two sequences we could use

$$\texttt{zip}(+, [0,1,2], [3,4,5]) = [3,5,7]$$

Second, `scan` applies a binary *associative* function, $\oplus$, over every prefix of a sequence, i.e., given a sequence $[x_1, x_2, \ldots, x_n, x_{n+1}]$, and an identity element $\imath$ such that $\imath \oplus x = x$, `scan` produces

$$[\imath, x_1, (x_1 \oplus x_2), (x_1 \oplus x_2 \oplus x_3), \ldots, (x_1 \oplus \cdots \oplus x_n)]$$

For explanatory purposes we use a related function `scanf` that takes a function $\oplus : Q \times \boldsymbol{\sigma} \to Q$, a value $q \in Q$, a sequence $[s_1, \ldots, s_n, s_{n+1}] \in \boldsymbol{\sigma}^*$ and produces the sequence

$$[q, q \oplus s_1, ((q \oplus s_1) \oplus s_2), \ldots, (((q \oplus s_1) \oplus s_2) \cdots \oplus s_n)]$$

in Q^* of length $n+1$. Finally, we use the function `split` to split a given sequence $\boldsymbol{s}$ according to a sequence $\boldsymbol{m}$ of lookback offsets as follows into a sequence of tuples (or subsequences):

$$\texttt{split}([s_i]_{i=1}^n, [m_i]_{i=1}^n) = [(s_{i+1-m_i}, s_{i+2-m_i}, \ldots, s_i)]_{i=1}^n$$

We next show how to define SFTs in terms of these primitives.

5.2 Describing SFTs With Higher-Order Functions

Recall that a k-SFT M is a tuple (Q, q^0, R) where q^0 is the initial state, and $R = \Delta \cup F$ is a finite set of rules, consisting of *transitions* Δ and *finalizers* F. Here we require M to be deterministic. Moreover, for ease of presentation and without loss of generality, we assume that all finalizers of M are trivial in the sense that they produce the empty output and are thus treated as final states in the classical sense. For example the 2-SFT in Figure 8(b) is deterministic. We assume that we work with an extended alphabet where there is an end-of-input (`EOI`) symbol and all valid input sequences end with `EOI`, so that finalizers are not needed. For each transition $p \xrightarrow{\varphi/f} q$ we refer to the arity of f as the *lookback* of the transition, that is denoted by $\ell(p,a)$ where $a \in [\![\varphi]\!]$. We represent normal characters by their code points as (non-negative) integers. `EOI` has code -1. We refer to f by $\phi(p,a)$, i.e., $\phi(p,a)$ takes $\ell(p,a)$ arguments.

Thus, the rules define a transition *function* from a given input and state to an output and a new state. Let $\delta(q,a)$ be the *state function*, implicitly defined by the rule set R, which takes as arguments a state q and an input a and produces a new state.

To transduce an input sequence s, M starts in state $q_1 = q^0$ and sequentially reads the symbols of s. When M reads the i'th symbol s_i from s in state q_i (first symbol being s_1), it enters state $q_{i+1} = \delta(q_i, s_i)$ and calls the *output function* $\phi(q_i, u)$, where $u = [s_{i+1-\ell(q_i,s_i)}, s_{i+2-\ell(q_i,s_i)}, \ldots, s_i]$, which maps to a finite (possibly empty) sequence of symbols in the output alphabet. We call the algorithm to transduce a string by M, `Transduce`, it takes as input M and a sequence s and produces the output sequence.

$$\begin{aligned} &\texttt{Transduce}(M, s) = \\ &\quad \textit{let } \boldsymbol{q} = \texttt{scanf}(\delta, q^0, s) \textit{ in} \\ &\quad \textit{let } \boldsymbol{m} = \texttt{zip}(\ell, \boldsymbol{q}, s) \textit{ in} \\ &\quad \texttt{flatten}(\texttt{zip}(\phi, \boldsymbol{q}, \texttt{split}(s, \boldsymbol{m}))) \end{aligned}$$

where `flatten` is defined in Section 4.

Example 6 Consider the 2-SFT `Rep` in Figure 8(b). We first extend `Rep` so that it has the final state q_2 and we assume that all valid inputs end with `EOI`. We write h for ψ_H. We write \$ for $\lambda x.(x = \texttt{EOI})$ and we write $\hbar$ for $\lambda x.\neg\$(x) \wedge \neg h(x)$. We also add a fourth state q_3 that corresponds to an *error state* (in practice this state may often be omitted, but we include it here for clarity). Let $? \stackrel{\text{def}}{=} [\blacklozenge]$, $f \stackrel{\text{def}}{=} \lambda x.[\textit{if } \psi_L(x) \textit{ then } \blacklozenge \textit{ else } x]$ and let $g \stackrel{\text{def}}{=} \lambda(x_{-1}, x).[(\textit{if } \psi_L(x) \textit{ then } x_{-1} \textit{ else } \blacklozenge), x]$. The extended 2-SFT is depicted in Figure 10. Let $a = \texttt{D83D}_{16}$ and $b = \texttt{DE0A}_{16}$. So $h(a)$ holds and $\psi_L(b)$ (thus also $\hbar(b)$) holds. Take the input

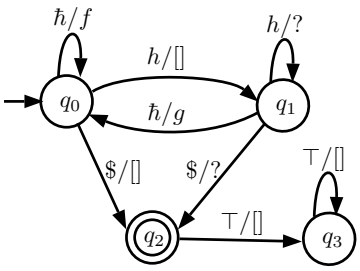


Figure 10. 2-SFT `Rep` extended with `EOI`. The state q_2 is the final state, meaning that it has the trivial finalizer $q_2 \xrightarrow{[]} \circledcirc$.

sequence $s = [b, a, a, b, a, \texttt{EOI}]$. We get that

$$\begin{aligned} s &= [b \;, a \;, a \;, b \;, a \;, \texttt{EOI}] \\ \texttt{scanf}(\delta, q_0, s) = \boldsymbol{q} &= [q_0 \;, q_0 \;, q_1 \;, q_1 \;, q_0 \;, q_1 \;] \\ \texttt{zip}(\ell, \boldsymbol{q}, s) = \boldsymbol{m} &= [1 \;, 0 \;, 0 \;, 2 \;, 0 \;, 0 \;] \\ \texttt{split}(s, \boldsymbol{m}) = \boldsymbol{u} &= [(b) \;, () \;, () \;, (a, b) \;, () \;, () \;] \\ \texttt{zip}(\phi, \boldsymbol{q}, \boldsymbol{u}) = \boldsymbol{v} &= [[\texttt{\textbf{\textit{?}}}] \;, [] \;, [\texttt{\textbf{\textit{?}}}] \;, [a, b] \;, [] \;, [\texttt{\textbf{\textit{?}}}] \;] \end{aligned}$$

and so $\texttt{flatten}(\boldsymbol{v})$ is the valid Unicode string $[\texttt{\textbf{\textit{?}}}, \texttt{\textbf{\textit{?}}}, a, b, \texttt{\textbf{\textit{?}}}]$ that would be displayed similar to "??☺?" where (a, b) is the surrogate pair UTF16 encoding of the code point $\texttt{1F60A}_{16}$ that is a smiley in the emoticon alphabet.[3] ■

5.3 Data-Parallel k-SFT

The prior section formalized k-SFT in terms of higher-order data parallel primitives. If the function on which these primitives operate are not *associative*, they must execute sequentially. If the BEK code contains registers, then in general it is not possible to directly write the resulting k-SFT with associative generalization of δ. Fortunately, as we saw in the previous section, our compilation algorithm can remove registers in many cases.

We compile δ into an associative operation on matrices. Because matrix multiplication is an associative operation that encodes graph traversals, this representation is amenable to data parallelism. The main idea is to lift `scanf` above to its associative counterpart `scan` by representing both states and symbols as matrices (so that we can talk about associativity to begin with). We build on known techniques from [14, 19, 25].

Graph Traversals with Matrix Multiplication: A convenient way to view k-SFT is as a graph where nodes in the graph are states and there exists an edge from state i to state j on symbol s if $\delta(i, s) = j$. A graph is simple to represent as an adjacency matrix: the set of allowed transitions for each symbol s in our input alphabet can be described by M_s, a $n \times n$ adjacency matrix, where n is the number of states, such that $(M_s)_{ij} = 1$ if state i transitions to state j on symbol s, and $(M_s)_{ij} = 0$, otherwise. In other words, an adjacency matrix is a symbolic representation of how a symbol from the input alphabet transitions *every* state in a k-SFT.

In order to deal with a potentially infinite alphabet $\boldsymbol{\sigma}$, we first divide the input alphabet $\boldsymbol{\sigma}$ into a finite set of equivalence classes $\boldsymbol{\sigma}_\sim$ such that the state transitions are invariant under $\sim$. Thus, we end up with one matrix M_C per equivalence class $C \in \boldsymbol{\sigma}_\sim$. We write $\widetilde{a}$ for the equivalence class containing the symbol a. But how do we actually compute the equivalence relation $\sim$? Let all the guards of the k-SFT be $\Psi = \{\varphi_1, \ldots, \varphi_m\}$. The equivalence relation $\sim$ can be effectively obtained by constructing all the *satisfiable* Boolean combinations over Ψ, for this we use the *minterm* algorithm from [9].

[3] See http://unicode.org/charts/PDF/U1F600.pdf.

Example 7 Consider the 2-SFT `Rep` in Figure 10. `Rep` uses four guards. The minterms are $\{h, \hbar, \$\}$ because they are all pairwise disjoint and their union equals $\top$. The SFA corresponding to `Rep` looks exactly the same as `Rep` but with output functions omitted. Since there are three equivalence classes, say $A = [\![h]\!]$, $B = [\![\hbar]\!]$, and $C = [\![\$]\!]$, we have three adjacency matrices (M_A, M_B and M_C) that describe how every state in the SFA transitions when reading symbols $a \in A$, $b \in B$ and $c \in C$, respectively.

$$M_A = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad M_B = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad M_C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Given a and b as in Example 6, we have $a \in A$ and $b \in B$. ■

Given this formulation, we use matrix multiplication '$\cdot$' as a mechanism for graph traversal. Assume the identity matrix I encodes the initial state of the SFA. Then adjacency matrix that encodes the state of the SFA after reading the first symbol a in an input sequence s is $I \cdot M_{\widetilde{a}}$. Further, the adjacency matrix that encodes the state of the SFA after reading the second symbol, b of s is $I \cdot M_{\widetilde{a}} \cdot M_{\widetilde{b}}$, etc.

From k-SFTs to matrices: To transform a k-SFT to operations on vectors and matrices, we define the following two functions, `inflate` and `project`; `inflate` generates the matrix $M_{\widetilde{a}}$ for each symbol $a \in \boldsymbol{\sigma}$; `project` extracts from matrix M_w the state of the (underlying) SFA, where M_w is obtained after reading the input sequence w, starting from state $q^0 = 0$:

$$\texttt{project}(M_w) = V_0 \cdot M_w \cdot V_F$$

where V_0 is an n-component *row* vector

$$V_0 = (1, 0, \ldots, 0)$$

and V_F is an n-component *column* vector

$$V_F^T = (0, 1, \ldots, n-1).$$

So $V_0 \cdot M_w$ extracts the first row, say X, of M_w, where X has exactly one element j that is 1 while all the other elements are 0 because the SFA is deterministic, where j is the state reached from q^0 after reading w. Thus $X \cdot V_F$ equals that state j.

Given this formulation, we implement an associative version $\hat{\delta}$ of the transition function δ that uses "inflated" symbols instead of actual symbols. Thus, instead of using the actual alphabet, the transduction uses the inflated alphabet $\{M_{\widetilde{a}} \mid a \in \boldsymbol{\sigma}\}$ and the input sequence is mapped to the inflated alphabet through the *initial inflation* of the input sequence $\texttt{map}(\texttt{inflate}, s)$:

$$\hat{\delta}(M, M_{\widetilde{a}}) \stackrel{\text{def}}{=} M \cdot M_{\widetilde{a}}$$

where M is a matrix that encodes the state of the SFA, and $M_{\widetilde{a}}$ is the inflated symbol a. Thus, the function $\hat{\delta}$ is associative by associativity of matrix multiplication and enables us to implement a data parallel version of `Transduce` as:

$$\begin{aligned} &\texttt{Transduce}(k\text{-SFT}, s) \stackrel{\text{def}}{=} \\ &\quad \textit{let } S = \texttt{map}(\texttt{inflate}, s) \textit{ in} \\ &\quad \textit{let } \boldsymbol{q} = \texttt{map}(\texttt{project}, \texttt{scan}(\hat{\delta}, S)) \textit{ in} \\ &\quad \textit{let } \boldsymbol{m} = \texttt{zip}(\ell, \boldsymbol{q}, S) \textit{ in} \\ &\quad \texttt{flatten}(\texttt{zip}(\phi, \boldsymbol{q}, \texttt{split}(s, \boldsymbol{m}))) \end{aligned}$$

An important point to note is that ϕ must be applied to the *original* (non-inflated) sequence s because the output functions of the rules of the SFT depend on the actual input, while both ℓ and δ are lifted to work directly on the equivalence classes.

5.4 An Efficient Matrix Representation

While matrix multiplication is associative, it unfortunately has a high overhead. Even an optimized implementation requires a significant amount of computation and memory—matrix multiplication runs in $O(m^3)$ where m is the number of states in a finite state automaton and requires m^2 memory.

To reduce this overhead such that it is linear in the number of states (both time and space), we exploit the fact that all of our adjacency matrices are square binary matrices with exactly one 1 in each row (and zeros elsewhere). We follow recent work and succinctly represent this type of adjacency matrix as an array where the value of the array at index i (staring from 0) is equal to the index of the column that contains a 1 for row i [25]. This is possible because each row contains *exactly* one 1.

In other words, the matrix is encoded as a one-dimensional array indexed by states. For example, the matrices M_A and M_B from the prior section, represented as arrays are $[1, 1, 3, 3]$ and $[0, 0, 3, 3]$, respectively. With this flattened representation, matrix multiplication of M_A with M_B reorders (with replacement) the elements of M_B according to M_A: $M_A \cdot M_B = [0, 0, 3, 3]$

To implement this efficient form of matrix multiplication is simple; we re-order the elements of one array by the elements of another:

```
void Multiply(char Ma[K], char Mb[K], char Mab[K]) {
  for(int i = 0; i < K; i++) Mab[i] = Mb[Ma[i]]; }
```

Because multiplying two matrices of this form produces another matrix of this form, we always represent matrices as linear arrays indexed by state which requires only m memory locations and a running time of $O(m)$.

5.5 Translating k-SFT to δ, ℓ and ϕ

To summarize, we produce the following pipeline. First, a programmer writes string manipulating functions in BEK. Next, we compile from BEK into an ST and then compile the ST into a k-SFT. Finally, we compile from the k-SFT to C# functions which encode ϕ, δ and ℓ.

These functions can then be applied as part of our data-parallel computation. As an integrated part of the pipeline, we also need a constraint solver, e.g., an SMT solver [3, 11], to construct the finite equivalence relation $\sim$ and to construct a classifier that decides membership of actual inputs in the $\sim$-equivalence classes.

Example 8 Consider again the 2-SFT in Figure 10. Recall the matrices M_A, M_B and M_C from Example 7. Let s be as in Example 6. So $S = \texttt{map}(\texttt{inflate}, s) = [M_B, M_A, M_A, M_B, M_A, M_C]$ and

$$\begin{aligned}&\texttt{map}(\texttt{project}, \texttt{scan}(\hat{\delta}, S))\\ &\quad = \texttt{map}(\texttt{project}, [I, M_B, M_B \cdot M_A, M_B \cdot M_A \cdot M_A, \ldots])\\ &\quad = [q_0, q_0, q_1, q_1, q_0, q_1]\end{aligned}$$

The rest of the output computation is the same as illustrated in Example 6. ■

The actual code generation to `C#` is discussed below. The code template for the generated `C#` is shown in Figure 13. The field `delta` corresponds to $\hat{\delta}$, the field `lookback` corresponds to ℓ (lifted to symbols), and the field `output` corresponds to ϕ. The semantics of `Transduce` is the implementation of `Encoder.Apply`. The implementation of `GetSymbol` uses predicates that have been generated by Z3 and then mapped to `C#` expressions to decide which symbol a concrete character maps to. The symbols are unique identifiers for the equivalence classes of $\sim$ as discussed above.

6. Evaluation

This section demonstrates the efficacy of our algorithms by investigating their correctness and performance. Section 6.1 explains the setup of the evaluation experiment. Section 6.2 empirically demonstrates that our generated programs produce consistent outputs compared to those available in existing system libraries. Section 6.3 provides a baseline performance comparison of our generated programs with existing library functions in the sequential case, demonstrating that our generated programs are comparable in performance. Finally, Section 6.4 demonstrates multi-factor speedups on a variety of real-world string transformation programs.

We use three encoder and decoder pairs in the following experiments, `HtmlEncode` and `HtmlDecode`, `Utf8Encode` and `Utf8Decode`, and finally, `Base64Encode` and `Base64Decode`. Each pair performs the logical inverse of the other (i.e., the output of an encoder is the input to the corresponding decoder). We also use sanitizers from the `AntiXss` library [22] in our sequential baseline comparison. All system encoders that we use are listed in Figure 11. There are several variations of some of them and they

Routine	Library
`HtmlEncode`	`AntiXss.AntiXssEncoder`
`HtmlDecode`	`AntiXss.AntiXssEncoder`
`CssEncode`	`AntiXss.AntiXssEncoder`
`JavaScriptStringEncode`	`AntiXss.AntiXssEncoder`
`HtmlEncode`	`System.Net.WebUtility`
`HtmlDecode`	`System.Net.WebUtility`
`HtmlEncode`	`System.Web.HttpUtility`
`HtmlDecode`	`System.Web.HttpUtility`
`JavaScriptStringEncode`	`System.Web.HttpUtility`
`ToBase64String`	`System.Convert`
`FromBase64String`	`System.Convert`
`UTF8.GetBytes` (encoder)	`System.Text.Encoding`
`UTF8.GetString` (decoder)	`System.Text.Encoding`

Figure 11. Pre-existing encoders used for comparison. All encoders are from .NET Framework 4.5.

differ behaviorally. For example `HtmlEncode` in `AntiXss` is different than both the one in `HttpUtility` and the one in `WebUtility`. In particular, the latter two do not encode control characters or the replacement character `&#xFFFD;` while the former does. However, the latter two encode all non-ASCII characters such as `ä` while the former does not encode all of them. In total, the latter two encode 101 characters out of all of the 256 extended ASCII characters, while the former encodes only 72. Since we have no bias in selecting one before the other, we compare against all of them in our experiments.

6.1 Setup

The concrete input to each individual experiment is the intermediate representation of the BEK program under consideration. It is an ST, as an instance of the `ST` python class in Figure 4. For example, the BEK intermediate representation of `Base64Encode` is shown in Figure 12. It corresponds to the ST shown in Figure 3. It has a single state $q = 0$, a register that is a pair of numbers, two transitions $\{r_1, r_2\}$ and three finalizers $\{f_1, f_2, f_3\}$. The initial register value is the pair $(0, 0)$. Notice that the function E is defined as a Z3 axiom. Such axioms are either inlined or used to define corresponding methods during code generation.

The python function `st2cs` in Figure 12 is the entry point that generates the final `C#` code. First, it invokes `Explore` (as defined in Figure 4) to construct the equivalent (modulo grouping) SFT$^+$. It then decomposes the transitions of the SFT$^+$ by introducing lookback and constructs a deterministic k-SFT. In the case of encoders whose input is assumed to be a valid UTF16 string, the resulting k-SFT is composed with the 2-SFT `Rep`, as shown in Figure 8(b), to lift the input strings to arbitrary sequences of characters (as opposed to valid UTF16 strings only, recall the discussion at the end

```
#genBase64Encode.py
from st2cs import *

def Base64Encode():
  bvs = BitVecSort(16); q = 0
  C = lambda c:BitVecVal(ord(c),bvs)
  Register = Datatype('Register')
  Register.declare('Pair',('Fst',bvs),('Snd',bvs))
  Register = Register.create()
  P=Register.Pair; F=Register.Fst; S=Register.Snd
  Edef = lambda x: If(x <= 25, x + 65,
           If(x <= 51, x + 71,
             If(x <= 61, x - 4,
               If(x == 62, C('+'), C('/')))))
  E = Function('Encode',bvs,bvs)
  v = Const('v',bvs)
  axiom = ForAll(v,(E(v)==Edef(v)),patterns=[E(v)])
  x = Var(1,bvs) #input character variable
  y = Var(0,Register) #register variable
  #transitions
  r1 = (q, And(x <= 255,F(y)!=2),
       [If(F(y)==0,E(x >> 2),E(S(y)|(x>>4)))],
       P(F(y)+1,If(F(y)==0,(x&3)<<4,(x&0xF)<<2)),q)
  r2 = (q, And(x <= 255,F(y)==2),
       [E(S(y)|(x>>6)),E(x&0x3F)],P(0,0),q)
  #finalizers
  f1 = (q, And(F(y) != 1, F(y) != 2), [])
  f2 = (q, F(y)==1,[E(S(y)),C('='),C('=')])
  f3 = (q, F(y)==2,[E(S(y)),C('=')])
  st = ST(q,P(0,0),[r1,r2],[f1,f2,f3],x,y,[axiom])
  return st

st2cs("Base64Encode",Base64Encode())
```

Figure 12. `Base64Encode` intermediate representation.

of Section 4.2). The generated `C#` is taylored for the subsequent parallelization step.

6.2 Consistency

To demonstrate that our synthesized encoders are correct, we checked the consistency of the BEK-generated output against independent implementations with the same functionality. We generate a set of 100 1-MB strings and evaluate both the independent output and the BEK generated code on each input.

The strings are chosen randomly. We used independently-produced implementations listed in Figure 11 for comparison. In each case we implemented a corresponding equivalent BEK program and automatically generated the code as explained above. For each individual input we compared that the output of the BEK generated program is equal to the output returned by the corresponding system encoder. We found no discrepancies. The system encoders, listed in Figure 11, came from .NET 4.5 core libraries and the `AntiXss` encoder library.

6.3 Sequential BEK

The generic code template for the generated sequential `C#` code for a given BEK encoder is shown in Figure 13. It exercises the same interface of the k-SFT that is used by the parallel implementation. The method `Encoder.Apply` implements the sequential routine. Concrete characters x are mapped to a finite set of symbols through `GetSymbol`. Each symbol identifies the equivalence class in which the input behaves similarly (recall the discussion from Section 5.3). The maximal lookback is k. The initial state is 0. Given a state q and a symbol s, `delta`$[q][s]$ is the next state. Output is computed from a given state q and symbol s by invoking `output`$[q][s](x_1,\ldots,x_m)$ where $(x_1,\ldots,x_m)$ is the subsequence of m previous characters, $m =$ `lookback`$[q][s]$, note that $m \leq k$. We have not optimized for the case when the input grouping size

```
namespace Experiments
{
 public static class Encoder
 {
  public const int finalSymbol = ...;
  public const int finalState = ...;
  public const int k = ...;
  public static int[][] delta = ...;
  public static int[][] lookback = ...;
  public static Func<int[],int>[][][] output = ...;
  public static int GetSymbol(int x) {...};

  public static string Apply(string s)
  {
   int L = s.Length;
   StringBuilder sb = new StringBuilder();
   var xs = new int[k];
   int state = 0;
   for (int i = 0; i < L; i++)
   {
    var x = s[i];
    var symbol = GetSymbol(x);
    int n = output[state][symbol].Length;
    if (n > 0)
    {
     var m = lookback[state][symbol];
     for (int j = 0; j < m; j++)
      xs[j] = s[i - m + 1 + j];
     for (int j = 0; j < n; j++)
      sb.Append((char)output[state][symbol][j](xs));
    }
    state = delta[state][symbol];
   }
   var m_end = lookback[state][finalSymbol];
   for (int j = 0; j < m_end - 1; j++)
    xs[j] = s[L - m_end + 1 + j];
   var n_end = output[state][finalSymbol].Length;
   for (int j = 0; j < n_end; j++)
    sb.Append((char)output[state][finalSymbol][j](xs));
   return sb.ToString();
  }
 }
}
```

Figure 13. Code template for generated `C#`.

is fixed (as with `Base64Decode`) but iterate one character at a time over the input.

For each of the *string to string* encoders in Figure 11 (the top 9 rows) we compared the running time with the corresponding BEK generated encoder `Encoder.Apply` over 100 randomly generated 1-megabyte strings (over extended ASCII). To get statistically significant results, we ran each experiment 10 times and report the mean m and the standard error e of the running time as $m(\pm e)$. For `HtmlEncode` we use a different version of BEK encoder in the case of `AntiXss` to match the semantics, while we use another BEK encoder for the other two libraries. For `HtmlDecode` we used the same BEK encoder in all three instances. As input to the decoder we used the output from the corresponding encoder, i.e., from the random inputs we generate inputs to the decoders using the corresponding encoders: inputs for `HtmlDecode` in `AntiXss` are generated with `HtmlEncode` in `AntiXss`, inputs for `HtmlDecode` in `WebUtility` are generated with `HtmlEncode` in `WebUtility`, etc. The Html decoder in BEK supports up to 3 digit decimal encodings, which is why we restricted this experiment to extended ASCII. We see no reason why supporting longer encodings would affect the sequential running time in any significant manner. The experiments were carried out on a Lenovo X1 Carbon laptop with Intel i7 2GHz processor having 8GB of RAM and running 64-bit Windows 8.1.

The results are shown in Figure 14. The sequential performance of our generated encoders are on par with existing hand-optimized codes. Our code is sometimes faster and sometimes slower than

Routine	Library	t(sec)	t_{Bek}(sec)	t_{Bek}/t
HtmlEncode	AntiXss	2.7(±.02)	4.6(±.1)	1.7
HtmlDecode	AntiXss	8.0(±.1)	3.5(±.03)	0.4
CssEncode	AntiXss	3.4(±.03)	9.5(±.05)	2.8
JSSEncode	AntiXss	3.6(±.1)	3.7(±.03)	1.0
HtmlEncode	WebUtility	6.9(±.1)	4.6(±.07)	0.7
HtmlDecode	WebUtility	11(±.2)	4.4(±.06)	0.4
HtmlEncode	HttpUtility	6.8(±.07)	4.5(±.04)	0.7
HtmlDecode	HttpUtility	11(±.1)	4.4(±.04)	0.4
JSSEncode	HttpUtility	3.6(±.07)	3.8(±.05)	1.0
Geomean :				0.92

Figure 14. Sequential comparison over 100x1MB strings.

these standard libraries. Consider `HtmlDecode`: BEK is significantly faster than the `AntiXss` baseline as this decoder has many corner cases which makes it difficult to hand-optimize (and thus points to the benefits of an automated compiler). However, consider `CssEncode` where BEK is almost 3x slower `AntiXss`. This is because all encodings have the form `\0000`XY and use the loop over function calls in `output` in Figure 13 to append '`\`' and four '`0`'s to `sb`, one at a time. We could amortize this huge overhead through simple optimizations (i.e., inlining function calls instead of indirectly calling them through a table), however, because this paper is about extracting data parallelism from BEK codes with registers, we left this simple optimization for future work.

The encoder producing inputs for `HtmlDecode` in the `AntiXss` experiment encodes less data than in the other two `HtmlDecode` experiments, as was explained above, which explains the BEK time difference of almost 1 second in those experiments.

To summarize, the geometric mean of all speedup experiments is 0.92 and at the 95% confidence interval, ranges from 0.57 to 1.46. So our approach is neither statistically faster nor slower than these hand-optimized baselines.

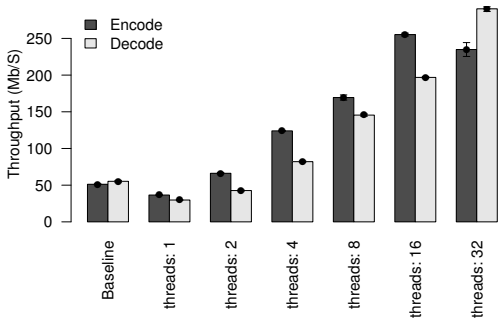
6.4 Data Parallel BEK

In this section we demonstrate multi-factor performance improvements over sequential baselines by utilizing multiple threads on multi-core hardware. It is worth noting our approach is not limited to multiple cores as it is data parallel and thus it can target many forms of data parallel hardware (e.g., GPUs or Hadoop clusters).

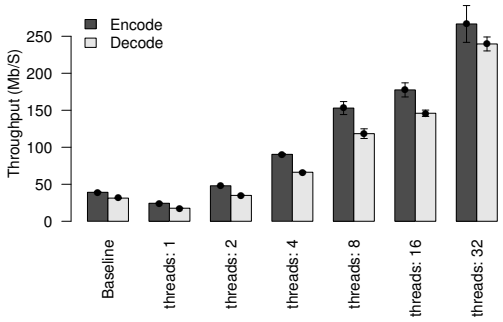
Platform: We conducted all experiments on an unloaded Intel 2 GHz (E5-2650) workstation with 64 GB of RAM and 32 logical processors (2 sockets with 16 physical cores).

Inputs: For both the `HTML` and `Utf8` encoder/decoder pair, we encode (and then decode) a 1GB subset of Wikipedia's HTML while for the `Base64` encoder/decoder pair, we encode (and then decode) a 100 MB SQL server executable. We measured the time it takes to execute a BEK program *after* reading the input from disk. Many of our performance numbers are faster than a commodity disk and thus we did not want our experiments to be IO bound. To compute throughput numbers, we measure both bytes read *and* bytes written as both numbers are required to understand a BEK program's performance. To get statistically significant results, we ran each experiment 10 times and reported the mean and 95% confidence interval of the mean.

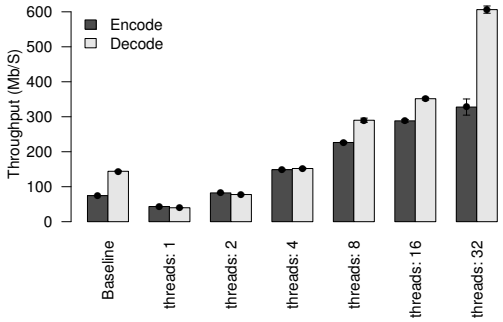
Throughput: Our data-parallel BEK programs exploit parallel hardware and are fast. As shown in Figure 15, with 32 logical threads, our encoders and decoders are able to process data at speeds of anywhere from 250 to 600 MB/sec. In contrast, the serial baselines are anywhere from 30 to 100 MB/sec. Secondly, even with small input data (i.e., `Base64` encoder only encodes 100 MB) we still see significant throughput. This implies that the cost to



(a) HTML Encode/Decode



(b) Base64 Encode/Decode

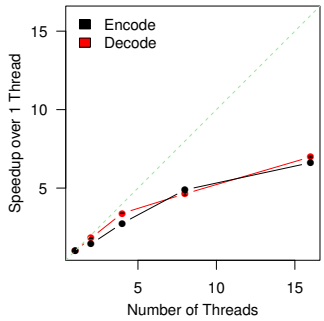


(c) UTF8 Encode/Decode

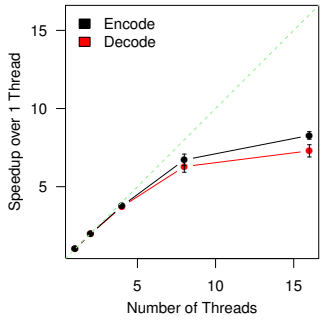
Figure 15. Throughput of various BEK encoders shown in MB/sec on the y axis, as function of threads.

create and manage threads does not dominate the computation. Lastly, our single-threaded implementation is *slower* than the baseline. This result is pedagogical: our implementation requires each thread do two passes over the input and with a single-thread our implementation does twice as much work as the sequential baseline. A real user of our system would never run with a single thread.

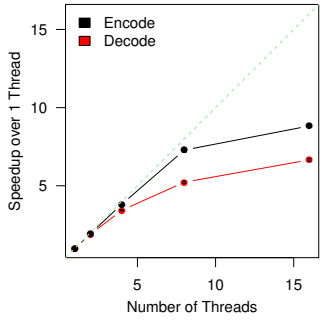
Speedup: Our data-parallel BEK programs provide nearly linear speedups for up to 8 physical cores and moderate speedups after that, up through 16 cores. Figure 16 details the speedup of x processors over a single processor. With 16 threads, our implementation is just shy of $8x$ speedup across the various encoders and decoders. Our approach saturates at 16 physical threads because the latency of reading data from shared caches and RAM becomes a bottleneck. Because our approach is data parallel, we could amortize this latency by using more physical machines such that each machine more effectively utilizes both caches and RAM. To summarize, these order-of-magnitude throughput improvements and speedups across a wide range of BEK programs are significant and enable data parallel processing of large amounts of data.



(a) HTML Encode/Decode



(b) Base64 Encode/Decode



(c) UTF8 Encode/Decode

Figure 16. Speedup of various BEK encoders, relative to a sequential baseline, as a function of threads.

7. Related Work

Symbolic finite transducers (SFTs) and BEK were originally introduced in [15] with a focus on security analysis 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 [32], where Symbolic Transducers (STs) are also introduced as an extension of SFTs with registers, but exploration of STs and code generation are not studied in [15, 32]. In contrast, the focus of this paper and its motivation is efficient transformation from STs to SFTs and k-SFTs with the particular application of *code generation* that supports efficient parallel execution. Another recent extension of SFTs, *extended* SFTs [8], is SFTs with *lookahead* where the primary motivation is to allow lookahead without introducing registers; unlike SFTs, ESFTs are not closed under composition. The grouping idea was originally exploited in [8] where it is used as a heuristic for lifting ESFTs to SFTs, but it is not integrated with exploration. A model similar to k-SFTs is k-SLTs, k-SLTs are introduced in [7], and use lookback instead of lookahead and, unlike ESFTs, step one symbol at a time. Our model of k-SFTs can be seen as a special class of parallelizable k-SLTs. A key property that we use here is monadicity of the predicates [33], which is needed for a k-SLTs to be translatable into a k-SFTs. The general case of deciding if a predicate is monadic is undecidable, which follows from [20, Proposition 2.b].

In recent years there has been considerable interest in automata over infinite languages [17, 30]. Finite words over an infinite alphabet are often called *data words* in the literature. Other related automata models are *pebble automata* [26] and *data automata* [6]. Several characterizations of logics with respect to different models of data word automata are studied in [5]. 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. A different line of work on automata with infinite alphabets introduces *lattice automata* [12] that are finite state automata whose transitions are labeled by elements of an atomic lattice with motivation coming from verification of symbolic communicating machines. *Streaming transducers* [1] is another recent symbolic extension of finite transducers. We do not know of prior work that has investigated the use of symbolic extensions of transducers for code generation.

In our implementation we use the off-the-shelf SMT solver Z3 [10] 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 [13]. The distinguishing feature of our exploration algorithm is that it computes a transformation that is behaviorally *equivalent* to the original ST with respect to the transduction semantics, which is important for correct code generation.

Finite state transducers have been used for dynamic and static analysis to validate sanitization functions in web applications in [2]. Other types of security analyses use string analysis [24, 35]. Yu et.al. show how multiple automata can be composed to model looping code [36]. Our work is complementary to previous efforts in using SMT solvers to solve problems related to list transformations. HAMPI [18] and Kaluza [28] extend the STP solver to handle equations over strings and equations with multiple variables. We are not aware of previous work investigating the use of finite state transducers for efficient code generation. One explanation for this is that classical finite state transducers are not directly suited for this purpose because SFTs can be exponentially more succinct with respect to the alphabet size.

Parallel algorithms for finite state machines, akin to the one presented in this work, have been known for a long time [14, 19]. Like this paper, Ladner and Fischer build a parallel implementation by phrasing finite state machine computation as matrix multiplication. Their approach targets applications with small finite state machines and thus they ignore the resulting cubic (in the number of states) overhead [19]. Hillis and Steele describe an improved algorithm based on parallel prefix sum which reduces this overhead to linear in the number of states [14]. To the best of our knowledge, none of these approaches allow infinite alphabets, neither were implemented on real hardware nor are they used with a larger framework for transducing. This paper builds on these theoretical insights, fits into a larger compilation framework which lets programmers use an expressive language to write encoders and decoders, and then evaluates those programs on large scale, parallel machines.

Finally, stream processing is a programming model which organizes programs by independent filters that communicate over data channels. Programmers need not think about parallelism as it is implicit in the programming model; a streaming program is the successive composition of independent filters and the runtime maps these independent computations to hardware parallelism [31, 37].

Unlike the programming model in this paper, streaming based programming models are not analyzable nor do they *break* data dependencies to expose data parallelism like we do in this work. In particular, streaming languages can only run as fast as hardware can expose a stream. Thus, it is not clear how they can hide I/O from reading large files from multiple disks.

8. Conclusions

This paper demonstrates how to compile an expressive domain-specific language, BEK to produce consistent and fast encoders and sanitizers. We use symbolic transducers as an intermediate formalism, and then introduce a novel algorithm which performs a symbolic partial exploration of these transducers to obtain simplified, stateless versions of the original BEK program.

We then show how to compile the resulting stateless transducers to data-parallel hardware. Our compilation results indicate significant runtime improvements: our data-parallel compilation speedups are up to $8x$ faster compared to a sequential implementation, when run on a 16-node commodity desktop.

References

[1] R. Alur and P. Cerný. Streaming transducers for algorithmic verification of single-pass list-processing programs. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11. ACM, 2011.

[2] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *IEEE Oakland Security and Privacy*, 2008.

[3] C. Barrett and C. Tinelli. Satisfiability modulo theories. In E. Clarke, T. Henzinger, and H. Veith, editors, *Handbook of Model Checking*. Springer, 2014. (to appear).

[4] D. Bates, A. Barth, and C. Jackson. Regular expressions considered harmful in client-side xss filters. In *Proceedings of the 19th international conference on World wide web*, WWW '10. ACM, 2010.

[5] M. Benedikt, C. Ley, and G. Puppis. Automata vs. logics on data words. In *Proceedings of the 24th international conference/19th annual conference on Computer Science Logic (CSL'10/EACSL'10)*, volume 6247 of *LNCS*, pages 110–124. Springer, 2010.

[6] M. Bojańczyk, A. Muscholl, T. Schwentick, L. Segoufin, and C. David. Two-variable logic on words with data. In *Proceedings of the 21st Annual IEEE Symposium on Logic in Computer Science (LICS'06)*, pages 7–16. IEEE, 2006.

[7] M. Botinčan and D. Babić. Sigma*: symbolic learning of input-output specifications. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL'13, pages 443–456. ACM, 2013.

[8] L. D'Antoni and M. Veanes. Static analysis of string encoders and decoders. In *14th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2013*, volume 7737 of *LNCS*, pages 209–228. Springer, 2013.

[9] L. D'Antoni and M. Veanes. Minimization of symbolic automata. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'14, pages 541–553. ACM, 2014.

[10] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *LNCS*. Springer, 2008.

[11] L. De Moura and N. Bjørner. Satisfiability modulo theories: Introduction and applications. *Commun. ACM*, 54(9):69–77, 2011.

[12] T. L. Gall and B. Jeannet. Lattice automata: A representation for languages on infinite alphabets, and some applications to verification. In *SAS 2007*, volume 4634 of *LNCS*, pages 52–68, 2007.

[13] P. Godefroid. Compositional dynamic test generation. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL'13, pages 47–54, 2007.

[14] W. D. Hillis and G. L. Steele. Data parallel algorithms. In *Commun. ACM*, volume 29, pages 1170–1183, Dec 1986.

[15] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and precise sanitizer analysis with Bek. In *USENIX Security*, August 2011.

[16] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 1979.

[17] M. Kaminski and N. Francez. Finite-memory automata. In *31st Annual IEEE Symposium on Foundations of Computer Science*, volume 2 of *FOCS'90*, pages 683–688. IEEE, 1990.

[18] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: a solver for string constraints. In *International Symposium on Software Testing and Analysis*, 2009.

[19] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.

[20] L. Libkin. Variable independence for first-order definable constraints. *ACM Transactions on Computational Logic*, 4(4):431–451, 2003.

[21] D. Lindsay and E. V. Nava. Universal XSS via IE8's XSS filters. In *Black Hat Europe*, 2010.

[22] *Anti-Cross Site Scripting Library*. Microsoft Corporation, . http://msdn.microsoft.com/en-us/security/aa973814.aspx.

[23] *Bek guide*. Microsoft Research, . http://rise4fun.com/bek/tutorial.

[24] Y. Minamide. Static approximation of dynamically generated web pages. In *WWW '05*, pages 432–441, 2005.

[25] T. Mytkowicz, M. Musuvathi, and W. Schulte. Data-parallel finite-state machines. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 529–542, New York, 2014. ACM.

[26] F. Neven, T. Schwentick, and V. Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. CL*, 5:403–435, 2004.

[27] *XSS Filter Evasion Cheat Sheet*. OWASP. http://ha.ckers.org/xss.html.

[28] P. Saxena, D. Akhawe, S. Hanna, S. McCamant, F. Mao, and D. Song. A symbolic execution framework for JavaScript. In *IEEE Security and Privacy*, 2010.

[29] P. Saxena, D. Molnar, and B. Livshits. Scriptgard: Preventing script injection attacks in legacy web applications with automatic sanitization. Technical Report MSR-TR-2010-128, Microsoft Research, August 2010.

[30] L. Segoufin. Automata and logics for words and trees over an infinite alphabet. In Z. Ésik, editor, *EACSL Annual Conference on Logic in Computer Science, CSL*, volume 4207 of *LNCS*, pages 41–57, 2006.

[31] W. Thies, M. Karczmarek, and S. P. Amarasinghe. Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, volume 2304 of *LNCS*, pages 179–196. Springer, 2002.

[32] M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjorner. Symbolic finite state transducers: Algorithms and applications. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12. ACM, 2012.

[33] M. Veanes, N. Bjørner, L. Nachmanson, and S. Bereg. Monadic decomposition. In *International Conference on Computer Aided Verification*, volume 8559 of *LNCS*, pages 628–645. Springer, 2014.

[34] P. Wadler. Deforestation: transforming programs to eliminate trees. In *Proceedings of the Second European Symposium on Programming*, pages 231–248, 1988.

[35] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su. Dynamic test input generation for web applications. In *International Symposium on Software Testing and Analysis*, 2008.

[36] F. Yu, T. Bultan, and O. H. Ibarra. Relational string verification using multi-track automata. In *Conference on Implementation and Application of Automata*, CIAA'10, pages 290–299, 2011.

[37] D. Zhang, Q. J. Li, R. Rabbah, and S. Amarasinghe. A lightweight streaming layer for multicore execution. *SIGARCH Comput. Archit. News*, 36(2):18–27, May 2008.