Data-Parallel String-Manipulating Programs

Microsoft Research Technical Report MSR-TR-2012-72

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

Abstract

Applications ranging from malware detection to graphics to Web security sanitization depend on string transformations, but writing such transformations is a challenge. Making these transformations run in parallel on a cluster of machines or special hardware, as often required for scalability, is an even greater challenge. We answer this challenge with fast, parallel string manipulating code compiled from BEK, a domain-specific language for writing complex string manipulation routines [9].

First, our new compilation pipeline maps a BEK program to an 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 a *symbolic partial evaluation* of these transducers to obtain simplified, stateless versions of the original program. These simplified versions can be lifted back to BEK, and from there compiled to C#, C, or JavaScript. Next, we show how the resulting transducers, post-exploration, fit into a recent advance in data-parallel compilation of finite state machines. Finally, we describe a concrete implementation built on the Windows High Performance Computing framework in a cluster.

We have implemented our code generation pipeline for BEK code corresponding to several real string manipulating functions, such as security sanitizers for Web applications. We use an automatic testing approach to compare our generated code to the original C# implementations and found no semantic deviations. Our generated C# code out-performs handwritten code by a factor of up to 3 and we generate code in C that is a factor of 5 faster. For a cluster with 32 nodes, we see speedups of 13.7 times compared to sequential C# code for an HTML sanitizer over 32 GB of data.

1. Introduction

We produce an *optimizing data-parallel code pipeline* for BEK, a domain-specific language for writing string manipulating programs. The original motivation for BEK came from Web programs called *sanitizers* that are evaluated on inputs from untrusted sources [9]. The language can further express a variety of string-manipulating programs. Previous work described how image blurring, GPS trace anonymization, and malware fingerprinting can be expressed in this way [20].

Unfortunately, previous work did not show how to compile BEK to standard languages, even for sequential code. We demonstrate compilation that generates faster code than manually-written string-manipulating routines. We then turn to parallelism. Previous work has shown that special state variables called *registers* are needed to express common functions. Unfortunately registers introduce data dependencies that are obstacles to parallelism. Manually removing registers requires reasoning about which state values will cause different behavior, which is difficult and time consuming.

We fix this problem with a novel *exploration algorithm* that attempts to remove registers from BEK programs. The exploration algorithm uses SMT solvers to determine ranges of state values that matter, then creates a new, equivalent program that has the register values removed. We combine this with recent advances in data-parallel compilation to obtain a *data-parallel back-end* for BEK programs. By data-parallel we mean that the computation can be distributed across the data with minimal need for cross-communication between threads. Data-parallelism enables processing gigabytes of text in a short time using multiple cores or multiple machines, as we show in Section 5. Our end result is a *fully-automatic compilation* from programs in an expressive language, BEK, to a LINQ-to-HPC data-parallel framework running on a cluster.

We do this without compromising the core strength of BEK: fast and precise analysis. Previous work has showed how to perform composition, equivalence checking, and commutativity checking that scale quadratically in the number of states, and which are fast in practice. Simply put, BEK is a tool for programmers to specify arbitrary *finite state* transformations. Before this work, there was no easy way to deploy the results, much less in a data-parallel way.

1.1 The BEK Language

We briefly describe the BEK domain-specific language for writing string transformations. As introduced in [9], the core construct in BEK is an 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. In this way a new string is built up based on the characters of the input string.

Programs can also have *register variables* that keep state during the iteration. Figure 1 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. For more details on BEK we refer to previous work or to the online BEK evaluator at `http://rise4fun.com/bek`. 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 [9, 20].

```
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)[q:=0;r:=0;]{
    case (x>0xFF): raise InvalidInput;
    case (q==0): yield (E(x>>2));
                 q:=1; r:=((x&3)<<4);
    case (q==1): yield (E(r|(x>>4)));
                 q:=2; r:=((x&0xF)<<2);
    case (q==2): yield (E((r|(x>>6))),E(x&0x3F));
                 q:=0; r:=0;
    end case (q==1): yield (E(r),'=','=');
    end case (q==2): yield (E(r),'=');
  };
}
```

Figure 1: Sample Bek program that does base64 encoding.

1.2 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.

The key contribution of this work is a novel *partial symbolic evaluation* algorithm that enables *removing registers* from Bek programs. Without this algorithm, we would not be able to exploit recent advances in data parallel finite transducers. We describe the algorithm in detail in Section 3. While the algorithm is not guaranteed to work in all cases (much like type checking, the algorithm is solving an undecidable problem), when it does work it produces a transducer that is semantically equivalent to the original, but *without* register state variables. We then put this algorithm into a fully-automatic pipeline that compiles Bek into a parallel implementation in the C# LINQ-to-HPC framework. This relieves the programmer from the burden of explicitly describing parallelism in Bek programs and allows them to use the full expressibility that registers provide. Removing registers manually is a difficult task because it requires reasoning about the different behavior exhibited by the program for different values of the register and so an automated solution is preferred.

1.3 Paper Contributions

This paper makes the following contributions.

- Previous work on Bek introduced an extension to symbolic transducers with *registers* [20]. Registers can be used to remember small amounts of state and are essential for modeling real sanitizers. In this paper, we present a novel *partial-evaluation algorithm modulo theories* that is complete for finite-valued register update functions and works modulo arbitrary background theories. The algorithm, if it terminates, outputs new transducers that are equivalent to the input but have no registers present.
- Our algorithm has several interesting features. First, the algorithm uses an SMT solver to eliminate registers (by folding them into control states) while maintaining symbolic representation from input sequences to output sequences. Next, the algorithm uses the model-generation feature of state-of-the-art SMT solvers to compute a finite control state partitioning as a dynamic forward reachability analysis. Finally, it uses unsatisfiability checks to prune provably unreachable states as a dynamic backward reachability analysis.

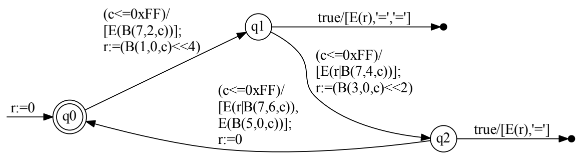


Figure 2: Symbolic transducer of the program in Figure 1.

- We show how to compile from Bek into C#, JavaScript, and C. We show how to check the resulting code for semantic differences from the original code. For server-side C# and C code, we achieve significant speedups: our transformation from Bek to C# results in code that outperforms production hand-written versions of the same function by as much as $3x$ and our C code is up to $5x$ faster. For JavaScript, our compiled code is sometimes faster and sometimes slower than common Web libraries, but our JavaScript comes with a guarantee that today's libraries cannot match: that the code will have the same semantics as the server-side filter.
- As our main application of the partial-evaluation or exploration algorithm, we show how to combine it with a recent advance by Mytkowicz and Schulte [14] to obtain a fully-automatic data-parallel compilation of Bek programs into the LINQ-to-HPC framework. 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. We achieve between 8.7x and 13.7x speedup on a 32 GB file transformed with representative benchmarks on a 32-machine cluster.

1.4 Paper Organization

The rest of this paper is organized as follows. Section 3 presents algorithms for transducer exploration. Section 4 describes the Bek back-end, focusing on the translation process for C#, C++, and JavaScript. Section 5 provides our experiment evaluation. Finally, Sections 6 and 7 describe related work and conclude.

2. Symbolic Transducers

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* $\mathcal{U}$, and is equipped with a language of function and relation symbols with fixed interpretations. Definitions below are given with $\mathcal{U}$ as an implicit parameter. We assume closure under Boolean operations and equality. Operations that are specific to $\mathcal{U}$ 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 [20]. The universe is multi-typed with $\mathcal{U}^\tau$ denoting the subuniverse of elements of type τ. We make use of the *empty tuple type* $\mathsf{T0}$ such that $\mathcal{U}^{\mathsf{T0}} = \{\langle\rangle\}$. While the definition below is consistent with the definition of STs as originally introduced in [20], here we use a variant where the control state component is explicit. This variant in better suited for presenting and explaining the key results of this paper.

Definition 1: A *Symbolic Transducer* or *ST* with input type σ output type γ and register type τ is a tuple $A = (Q, q^0, r^0, R)$,

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

If τ is the empty tuple type $\mathtt{T0}$ then A is called a *Symbolic Finite Transducer* or *SFT*.[1] ⊠

We write $[e_1, \ldots, e_k]$ or $[e_i]_1^k$ for finite sequences of length $k \geq 0$ and if $\boldsymbol{f}$ is a sequence $[f_i]_1^k$ of λ-terms of type $\sigma \to \gamma$ then $[\![\boldsymbol{f}]\!]$ is the function that maps $a \in \mathcal{U}^\sigma$ to $[[\![f_1]\!](a), \ldots, [\![f_k]\!](a)] \in \mathcal{U}^{\gamma *}$, where X^* denotes the set of all finite sequences over X. The empty sequence is $[]$ or ϵ.

The semantics of A is given by the following concrete transition relation. Let $q, q' \in Q$, $r, r' \in \mathcal{U}^\tau$, $a \in \mathcal{U}^\sigma$, $\boldsymbol{b} \in \mathcal{U}^{\gamma *}$. Then $(q, r) \xrightarrow{[a]/\boldsymbol{b}}_A (q', r')$ denotes that there exists a transition $q \xrightarrow{\varphi/o;u} q'$ such that $\varphi(a, r)$ holds, the output sequence $\boldsymbol{b}$ is $[\![o]\!](a, r)$ and the new register r' is $[\![u]\!](a, r)$. Similarly, $(q, r) \xrightarrow{\epsilon/\boldsymbol{b}}_A \bullet$ denotes that there exists a finalizer $q \xrightarrow{\varphi/o} \bullet$ such that $\varphi(r)$ holds and $\boldsymbol{b}$ is $[\![o]\!](r)$.

The *reachability relation* $p \overset{\boldsymbol{a}/\boldsymbol{b}}{\twoheadrightarrow}_A p'$ for $\boldsymbol{a} \in \mathcal{U}^{\sigma *}$, $\boldsymbol{b} \in \mathcal{U}^{\gamma *}$, and $p, p' \in (Q \times \mathcal{U}^\tau) \cup \{\bullet\}$ is defined through the closure under the following conditions, where '$\cdot$' is concatenation of sequences, note that $\epsilon \cdot \boldsymbol{x} = \boldsymbol{x} \cdot \epsilon = \boldsymbol{x}$:

- If $p \xrightarrow{\boldsymbol{a}/\boldsymbol{b}}_A p'$ then $p \overset{\boldsymbol{a}/\boldsymbol{b}}{\twoheadrightarrow}_A p'$.
- If $p \overset{\boldsymbol{a}/\boldsymbol{b}}{\twoheadrightarrow}_A p_1 \overset{\boldsymbol{a}'/\boldsymbol{b}'}{\twoheadrightarrow}_A p_2$ then $p \overset{\boldsymbol{a}\cdot\boldsymbol{a}'/\boldsymbol{b}\cdot\boldsymbol{b}'}{\twoheadrightarrow}_A p_2$.

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

$$\mathscr{T}_A(\boldsymbol{a}) \stackrel{\text{def}}{=} \{\boldsymbol{b} \mid (q_A^0, r_A^0) \overset{\boldsymbol{a}/\boldsymbol{b}}{\twoheadrightarrow}_A \bullet\}$$

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

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 4 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 1 The BEK program in Figure 1 does base64 encoding of byte sequences. Base64 is a standard used to transfer binary data over textual media. In the program, q is a state variable and r is a register. The input type, output type and register type is `int`. The ST, shown in Figure 2, has 3 states, $Q = \{\mathtt{q0}, \mathtt{q1}, \mathtt{q2}\}$, initial state is $\mathtt{q0}$, and the initial register value is 0. There are 3 transitions, and 3 finalizers. For example, the transition from state $\mathtt{q0}$ to state $\mathtt{q1}$ is

$$\mathtt{q0} \xrightarrow{\lambda(c,r).(c \leq \mathrm{FF}_{16})/[\lambda(c,r).E(\mathit{Bits}(7,2,c))];\ \lambda(c,r).\mathit{Bits}(1,0,c) \ll 4} \mathtt{q1}$$

and the finalizer from state $\mathtt{q0}$ is $\mathtt{q0} \xrightarrow{\lambda r.\mathit{true}/\epsilon} \bullet$, i.e., state $\mathtt{q0}$ is final in the classical sense. Fix $r = 0$ and $c = \text{'A'} = 1000001_2$. Clearly $c \leq \mathrm{FF}_{16}$. We have $E(\mathit{Bits}(7, 2, c)) = E(10000_2) = E(16) = 16 + 65 = \text{'Q'}$ and $(\mathit{Bits}(1, 0, c) \ll 4) = (1 \ll 4) = 10000_2 = 16$, so the concrete transition is

$$(\mathtt{q0}, 0) \xrightarrow{[\text{'A'}]/[\text{'Q'}]} (\mathtt{q1}, 16)$$

If we do one more step from configuration $(\mathtt{q1}, 16)$ with input 'B' we get the concrete transition

$$(\mathtt{q1}, 16) \xrightarrow{[\text{'B'}]/[\text{'U'}]} (\mathtt{q2}, 8)$$

Suppose that the input sequence ends here. Then we use the finalizer from state $\mathtt{q2}$ for the concrete input-ϵ transition:

$$(\mathtt{q2}, 8) \xrightarrow{\epsilon/[\text{'I'},\text{'='}]} \bullet$$

By using the derived reachability relation we have

$$(\mathtt{q0}, 0) \overset{[\text{'A'},\text{'B'}]/[\text{'Q'},\text{'U'},\text{'I'},\text{'='}]}{\twoheadrightarrow} \bullet$$

Thus, $\mathscr{T}_{\mathtt{b64e}}(\texttt{"AB"}) = \{\texttt{"QUI="}\}$. The base64 example is available in an online tutorial of BEK[2]. ■

3. Partial Exploration of STs

In this section we develop an algorithm that allows us to concretize either all or some part of the the register used in a Symbolic Transducer A. To simplify the discussion, we assume, without loss of generality, that the register type τ comes with projection functions $\pi_1 : \tau \to \tau_1$, $\pi_2 : \tau \to \tau_2$, and pairing function $\lceil\_,\_\rceil : \tau_1 \times \tau_2 \to \tau$. The aim is to compute a new ST A' that is equivalent to A but whose register type is τ_2 and the first projection of the register values has been folded into the state space of A'. Equivalence of A and A' means that $\mathscr{T}_A = \mathscr{T}_{A'}$. If we want to eliminate the register completely, we take τ_2 to be $\mathtt{T0}$.

$\mathcal{E}_{(\pi_1,\pi_2)}(A)$ The basic algorithm uses a least fixpoint computation that constructs a set of transitions Δ' and finalizers F' over a state space $Q' \subseteq Q_A \times \mathcal{U}^{\tau_1}$ such that the following conditions are met:

1. $\langle q_A^0, \pi_1(r_A^0)\rangle \in Q'$;
2. for each $\langle q, r\rangle \in Q'$:
   (a) If $q \xrightarrow{\varphi/[o_i]_1^k;u} q' \in \Delta_A$ and there is $r' \in \mathcal{U}^{\tau_1}$ s.t.
      i. $\varphi' = \lambda(x, y).(\varphi(x, \lceil r, y\rceil) \wedge r' = \pi_1(u(x, \lceil r, y\rceil)))$ is satisfiable;
      ii. let, for $1 \leq i \leq k$, $o_i' = \lambda(x, y).o_i(x, \lceil r, y\rceil)$;
      iii. let $u' = \lambda(x, y).\pi_2(u(x, \lceil r, y\rceil))$;

[1] In other words, an SFT is an ST without registers.

[2] `http://www.rise4fun.com/Bek/tutorial/base64`

then $\langle q', r'\rangle \in Q'$, $\langle q, r\rangle \xrightarrow{\varphi'/[o'_i]_1^k; u'} \langle q', r'\rangle \in \Delta'$.

(b) If $q \xrightarrow{\varphi/[o_i]_1^k} \bullet \in F_A$ and

i. $\varphi' = \lambda y.\varphi(\lceil r, y\rceil)$ is satisfiable;

ii. let, for $1 \leq i \leq k$, $o'_i = \lambda y.o_i(\lceil r, y\rceil)$;

then $\langle q, r\rangle \xrightarrow{\varphi'/[o'_i]_1^k} \bullet \in F'$.

The result is $(Q', \langle q_A^0, \pi_1(r_A^0)\rangle, \pi_2(r_A^0), \Delta' \cup F')$.

The calculation of values for r' (in step 2(a)) in the algorithm uses iterated model search modulo a decision procedure for $\mathcal{U}$ (such as, a backend SMT solver). The search starts with the quantifier free formula $\varphi(x, \lceil r, y\rceil) \wedge z = \pi_1(u(x, \lceil r, y\rceil))$, say ψ, in a logical context where x, y and z are uninterpreted constants. While ψ is satisfiable with model $M \models \psi$, the value $r' = z^M$ is stored, ψ is updated to be $\psi \wedge z \neq r'$, and the search is repeated.

Theorem 1: If $\mathcal{E}_{\bar{\pi}}(A)$ terminates then $\mathscr{T}_{\mathcal{E}_{\bar{\pi}}(A)} = \mathscr{T}_A$.

We omit the formal proof of the theorem but note that termination of the algorithm depends on two factors: *decidability* of the background theory, and *finiteness* of the reachable subset P of $\mathcal{U}^{\tau_1}$. Then there are finitely many transitions computed in step 2(a), and since $Q \times P$ is finite there can be no infinite computation paths (because no pair $\langle q, r\rangle$ is repeated on an exploration path). Thus, the computation terminates by virtue of Königs Lemma.

The basic algorithm is subject to several optimizations, such as *deadend elimination* (a deadend is a state q from which there is no path to $\bullet$, here ignoring registers), *incremental model search* (taking advantage of logical contexts of modern SMT solvers), and *register restriction* (program transformation intended to limit the possible range of values for r' in step 2(a)).

Example 2 To illustrate the idea of register restriction consider the following transitions (here $\top = \lambda(x, y).\mathit{true}$ and all types are int):

$$q_0 \xrightarrow{\top/\_;\ \lambda(x,y).x} q_1 \xrightarrow{\top/[\lambda(x,y).(x+\underline{(y \bmod 4)})];\ \lambda(x,y).v(x)} q_2$$

Then, the transformed transitions

$$q_0 \xrightarrow{\top/\_;\ \lambda(x,y).\underline{(x \bmod 4)}} q_1 \xrightarrow{\top/[\lambda(x,y).(x+y)];\ \lambda(x,y).v(x)} q_2$$

are equivalent (with respect to the transduction semantics) but restrict the distinct register values assigned from state q_0 (underlined) to only 4 possible values, as opposed to unboundedly many in the original case. ■

Example 3 As a concrete example of full register elimination we use the ST $\mathsf{b64e}$ in Figure 2. The function $B(m, n, c)$ extracts bits m through n from c. For example $B(7, 4, 10110110_2) = 1011_2$. Here π_1 is $\lambda x.x$, π_2 is $\lambda x.\langle\rangle$, and $\lceil x, y\rceil \stackrel{\text{def}}{=} x$.

Start with $Q' = \{\langle \mathsf{q0}, 0\rangle\}$, $\Delta' = \emptyset$, $F' = \emptyset$. Let $q = \mathsf{q0}$ and $r = 0$ and consider the transition from $\mathsf{q0}$ to $\mathsf{q1}$ in 2(a). Let ψ be the predicate $(r' = (B(1, 0, x) \ll 4))$. Iterated model search of ψ provides four possible distinct satisfying assignments for r': $\{0, 10_{16}, 20_{16}, 30_{16}\}$. Q' is updated to

$$\{\langle \mathsf{q0}, 0\rangle, \langle \mathsf{q1}, 0\rangle, \langle \mathsf{q1}, 10_{16}\rangle, \langle \mathsf{q1}, 20_{16}\rangle, \langle \mathsf{q1}, 30_{16}\rangle\}$$

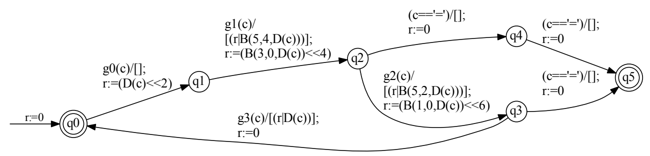


Figure 3: ST of a base64 decoder, $\mathsf{b64d}$, guards $\mathsf{g2}$ and $\mathsf{g3}$ do not accept **'='** (so $\mathsf{b64d}$ is deterministic). Function D is the inverse of E from Figure 1.

Δ' is updated to (we omit the register)

$$\begin{aligned}
\{&\langle \mathsf{q0}, 0\rangle \xrightarrow{\lambda x.(0=(B(1,0,x)\ll 4))/[E(B(7,2,x))]} \langle \mathsf{q1}, 0\rangle,\\
&\langle \mathsf{q0}, 0\rangle \xrightarrow{\lambda x.(10_{16}=(B(1,0,x)\ll 4))/[E(B(7,2,x))]} \langle \mathsf{q1}, 10_{16}\rangle,\\
&\langle \mathsf{q0}, 0\rangle \xrightarrow{\lambda x.(20_{16}=(B(1,0,x)\ll 4))/[E(B(7,2,x))]} \langle \mathsf{q1}, 20_{16}\rangle,\\
&\langle \mathsf{q0}, 0\rangle \xrightarrow{\lambda x.(30_{16}=(B(1,0,x)\ll 4))/[E(B(7,2,x))]} \langle \mathsf{q1}, 30_{16}\rangle\}
\end{aligned}$$

Now consider 2(b). We have $\mathsf{q0} \xrightarrow{\lambda y.\mathit{true}/\epsilon} \bullet$ and thus F' becomes $\{\langle \mathsf{q0}, 0\rangle \xrightarrow{/\epsilon} \bullet\}$. Step 2 is repeated for the new states in Q'. The algorithm terminates with $|Q'| = |F'| = 21$, and $|\Delta'| = 84$. ■

From our experience, full exploration works well for string encoders (such as sanitizers). A direct application of full exploration is less suited for *decoders* that require a lookahead of more than 2 characters (such as HtmlDecoder), where full exploration may lead to state space explosion. The following example is a border line case.

Example 4 Similar to the base64 encoder, a base64 *decoder* $\mathsf{b64d}$ can also be described as an ST, see Figure 3. Full exploration of $\mathsf{b64d}$ produces an SFT with 87 states, 1159 transitions and 2 finalizers. ■

A unique advantage of the symbolic approach is that it allows us to "adjust" the granularity of characters and to use more powerful character theories without sacrificing precision. The following example illustrates this aspect of STs.

Example 5 Consider the ST if Figure 3. Notice that the lengths of *all* sequences accepted by $\mathsf{b64d}$ are multiples of 4. Using this observation, we can symbolically compose all possible paths of length 4 from state $\mathsf{q0}$ and lift the input type to int^4. Now, in this case, trivial application of full exploration (because register updates are 0 in the composed transitions) will eliminate the register and we are left with 2 states and 3 transitions. ■

Notice that in Example 5 we *changed* the input type from int to int^4 while the output type remained the same. For the purposes of the data-parallel translation this transformation is fine because it does not affect the semantics of the input-output behavior. Formally it means that the initial input sequence is passed through a function that first groups individual elements into 4-tuples that are then consumed by the transducer.

Impact of Exploration Algorithm Figure 4 compares the sizes of the state machines needed to achieve the different encoding and decoding tasks. The table indicates the advantages of using the exploration algo-

	Rules		
Coder	**FSM**	**SFT**	**SFT$^+$**
`UTF8Encode`	2^{16}	12	5
`UTF8Decode`	2^{16}	6371	5
`Base64Encode`	5397	105	4
`Base64Decode`	5445	1161	5

Figure 4: Exploration sizes in total nr of rules. FSM is the classical explicit representation. SFT$^+$ is SFT + grouping.

rithm, in particular, in combination with grouping the succinctness is quite remarkable. While the grouping used in Example 5 was an easy one (a fixed length one), in the other cases the groupings are variable length sequences (ranging between 1 and 4 characters).

Without this dramatic reduction in the size of the SFT, we would not be able to compile the SFT to exploit Mytkowicz and Schulte's data parallel finite state machines. In particular, their approach requires that the number of states in the transducer be *small*, which our exploration algorithm provides.

4. Data-Parallel Translation

In the prior sections we demonstrated how to remove registers from ST and turn them into SFTs. In this section we describe how to compile ST into SFT so it can exploit data-parallel hardware. In particular, we demonstrate an end to end compilation of SFT to a large cluster running LINQ-to-HPC [12].

Recently, Mytkowicz and Schulte framed the evaluation of finite state transducers as associative operations over vectors and matrices [14]. Because their approach uses associative operations, it can take advantage of data-parallel hardware. Their approach, however, requires that the number of states in the ST be small in order to be efficient.

Our key insight that allows us to combine SFT and the approach of Mytkowicz and Schulte is that SFT exploration removes registers and at the same time reduces the number of states in the SFT. In effect, SFT exploration pushes the complexity of the SFT 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 described by Mytkowicz and Schulte that runs on a large cluster.

4.1 Data-Parallel Operators

To aid our discussion, we introduce two higher-order data-parallel primitives.

`zipwith` 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

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

`scan` applies a binary *associative* function, $\oplus$, over every prefix of a sequence. For example, given a sequence of n elements

$$[x_0, x_1, x_2, \ldots, x_n]$$

`scan` produces a new sequence

$$[x_0, (x_0 \oplus x_1), (x_0 \oplus x_1 \oplus x_2), \ldots, (x_0 \oplus x_1 \oplus \ldots x_n)]$$

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

4.2 Describing SFT With Higher-Order Functions

Recall that a SFT is a tuple $(Q, q^0, R \cup F)$ where q^0 is the initial state, $R \cup F$ is a finite set of *input* and *finalizer* rules. Each rule defines a transition from a pair of input symbol and state to an output symbol and a new state. Let $\delta(q, s)$ be the (flattened) transition function, implicitly defined by the rule sets R and F, which takes as arguments a state and a symbol and produces a new state. (In this view we assume that that the final rule from a state is triggered by a special "end-of-input" symbol and leads to a unique final state.)

To transduce a string s by a SFT, the SFT starts in state q^0 and sequentially reads the symbols of s. When the SFT reads the i'th symbol, s_i from s, it enters state $q = \delta(q, s_i)$ and calls function $\phi(q, s_i)$ with state q and symbol s_i, which maps to a sequence of symbols in the output alphabet.

We call the algorithm to *transduce* a string by a SFT, `Transduce` which takes as input a SFT and a string s and produces a new string s' which is the result of applying ϕ to each state of the SFT, after the SFT reads the ith symbol in s. Using the higher-order functions introduced in Section 4.1, we can write `Transduce`[3] as:

$$\texttt{Transduce}(\text{SFT}, s) = \texttt{zipwith}(\phi, \texttt{scan}(\delta, q_0, s), s)$$

4.3 Translating SFT to δ and ϕ

We then produce the following pipeline. First, a programmer writes string manipulating functions in Bek. Next, we compile from Bek into an ST and then to an SFT by using the exploration algorithm. Finally, we compile from the SFT to C# functions which encode ϕ and δ. These functions can then be applied as part of our data-parallel computation.

For example, consider a simple Bek program (here chosen so that exploration is not needed):

```
program sample(t) {
 return iter(c in t)[state := 0;]{
  case(state==0):
    if (c=='6'){state:=2;}
    else if (c=='7'){state:=1;}
    else {state:=0; yield(c);}

  case(state==1):
    if ((c=='6')||(c=='7')){state:=0; yield(22+c);}
    else {state:=0; yield('7'); yield(c);}

  case(state==2):
    if ((c=='6')||(c=='7')){state:=0; yield(12+c);}
    else {state:=0; yield('6'); yield(c);}
 };
}
```

A simple syntax-directed translation produces the following sequential C# implementation which takes state as an out parameter and the current character and (i) updates the state of the SFT and (ii) returns an enumeration of output characters based on both the current state and the character passed in.

```
IEnumerator<char> Apply(out int state, char c) {
  switch (state){
    case (0): {
      if (c == '6') {state = 2; yield break;}
      else if (c == '7') {state = 1; yield break;}
      else {state = 0; yield return c;}
    }
    case (1): {
      if (c == '6'||c == '7') {
        state=0; yield return 22+c;
      }
      else {
        state=0; yield return '7'; yield return c;
      }
    }
    case (2): {
      if (c == '6'||c == '7') {
        state=0; yield return 12+c;
      }
      else {
        state=0; yield return '6'; yield return c;
```

[3] To aid discussion and demonstrate `Transduce` type-checks visit `http://pastebin.com/axXZcw3f`

```
      }
}}}
```

To build a data-parallel version of this SFT we need two functions δ and ϕ. We alter the syntax directed translation of a SFT to C# to produce `Delta` such that we keep the control flow during the translation but remove the statements in `Apply` that generate output (e.g. the `yield` statements).

```
int Delta(int state, char c) {
  switch (state){
    case (0): {
      if (c == '6') { return 2;}
      else if (c == '7') { return 1;}
      else { return 0;}
    }
    case (1): {
      return 0;
    }
    case (2): {
      return 0;
    }
}}
```

Likewise, to produce ϕ, we no longer update the state variable, we use the current state and character to produce an enumeration of output characters.

```
IEnumerable<char> Phi(int state, char c) {
  switch (state){
    case (0): {
      if (c == '6' || c == '7') { yield break; }
      else { yield return c; }
    }
    case (1): {
      if (c == '6' || c == '7') {
        yield return (char)(22+((int)c));
      }
      else {
        yield return '7', yield return c;
      }
    }
    case (2): {
      if (c == '6' || c == '7') {
        yield return (char)(12+((int)c));
      }
      else {
        yield return '6', yield return c;
      }
}}}
```

4.4 Data-Parallel SFT

The prior section formalized SFT in terms of higher-order data parallel primitives. If the function on which these primitives operate (e.g. δ and ϕ) 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 SFT with an associative δ and ϕ. Fortunately, as we saw in the previous section, the exploration algorithm can remove registers in many cases.

Following Mytkowicz and Schulte, we compile δ and ϕ into associative operations on vectors and matrices. Because matrix multiplication is an associative operation that encodes graph traversals, this representation is amenable to data parallelism.

Graph Traversals with Matrix Multiplication: A convenient way to view 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 an SFT.

More precisely, first the universe $\mathcal{U}^\sigma$ of input symbols s is divided into finitely many equivalence classes $\hat{s}$ where each equivalence class is defined by a *minterm* of all guards of transitions that occur in the SFT. There are *finitely* many such minterms, independent of the size of $\mathcal{U}^\sigma$ and for all $e \in \hat{s}$, $M_s = M_e$. So we have finitely many such matrices (one per minterm). In this way we lift the technique in [14] to work here over arbitrary input types. This is important for dealing with infinite alphabets and grouped symbols (such as the one used in Example 5) even when $\mathcal{U}^\sigma$ is finite but large.

Given this formulation, we use matrix multiplication as a mechanism for graph traversal; if the identity matrix (M_I) encodes the initial state of the SFT then the adjacency matrix that encodes the state of the SFT after reading the first symbol s_o in an input s is $M_I \cdot M_{s_0}$. Further, the adjacency matrix that encode the state of the SFT after reading the second symbol, s_1 in an input s is $M_I \cdot M_{s_0} \cdot M_{s_1}$, and so on.

From SFT **to Matrices:** To transform a SFT to operations on vectors and matrices, we follow Mytkowicz and Schulte and define the following two functions, `inflate` and `project`. `inflate` generates a matrix from each symbol in the input alphabet. Given a symbol s, `inflate` returns a $n \times n$ matrix M_s such that:

$$\texttt{inflate}(s) = (M_s)_{ij} = \begin{cases} 1 & \text{if } \delta(i, s) = j \\ 0 & \text{otherwise} \end{cases}$$

Next, `project` extracts from matrix M_s the state of the SFT after reading symbol s, starting from state q^0:

$$\texttt{project}(M_s) = V_{q^0} \cdot M_s \cdot V_F$$

where V_{q^0} is an n-component *row* vector

$$V_{q^0} = \begin{cases} 1 & \text{if } i = q_0 \\ 0 & \text{otherwise} \end{cases}$$

and V_F, an n-component *column* vector

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

Given this formulation, we implement an associative version of the transition function, δ:

$$\hat{\delta}(M, s_i) = M \cdot \texttt{inflate}(s_i)$$

where M is a matrix that encodes the state of the SFT, and s_i is the ith symbol in string s.
With an associative version of δ, we implement a data parallel version of `Transduce` as:

$$\texttt{Transduce}(\text{SFT}, s) =$$
$$\texttt{zipwith}(\phi, \texttt{map}(\texttt{project}, \texttt{scan}(\hat{\delta}, M_I, s)), s)$$

To increase efficiency (e.g. remove a pass over the input) we take advantage of the fact that functions compose. For example:

$$\texttt{map}(f, \texttt{map}(g, list)) = \texttt{map}(f \cdot g, list)$$

and thus we can compose ϕ with `project` to rewrite `Transduce` as:

$$\texttt{Transduce}(\text{SFT}, s) = \texttt{zipwith}(\phi \cdot \texttt{project}, \texttt{scan}(\hat{\delta}, M_I, s), s)$$

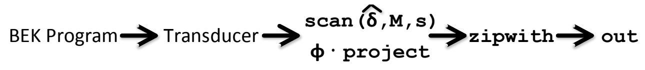
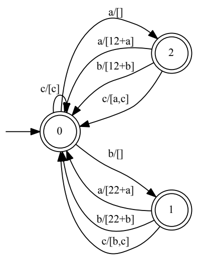


Figure 5: The pipeline for compiling from BEK to a data-parallel implementation.

In other words, given an SFT, this section describes an automatic method to compile an SFT into an implementation that is suitable to data-parallel hardware. The end-to-end pipeline is shown in Figure 5. We start with a BEK program, then compile the program to a symbolic transducer. From the transducer we derive the associative operators ϕ and $\hat{\delta}$. Finally we feed these to the `zipwith` primitive which in turn yields the output `out`. After a brief example, we demonstrate how we implement `Transduce` on a LINQ-to-HPC cluster.

A Concrete Example: In this section we walk through how to build a data-parallel implementation of the simple BEK program introduced in Section 4.3.



In order to simplify the exposition we use a for the digit `6`, we use b for the digit `7`, and we use c for any other character besides a and b. More precisely, a, b, and c are character predicates that partition the alphabet $\mathcal{U}^{\mathsf{bv}7}$, but it is convenient for the exposition to view them as three distinct characters.

In this view, the SFT has control states $\{0, 1, 2\}$, input alphabet $\Sigma = \{a, b, c\}$, initial state 0, and whose δ and ϕ are shown to the right.

There are three symbols in our input alphabet a, b and c: thus we have three adjacency matrices (M_a, M_b and M_c) that describe how every state in the SFT transitions when reading symbols a, b and c, respectively.

$$M_a = \texttt{inflate}(a) = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

$$M_b = \texttt{inflate}(b) = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

$$M_c = \texttt{inflate}(c) = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

Suppose we are given the input $s = \epsilon ab$. To `Transduce`(s), we first calculate a `scan` over the symbols in s which produces the sequence

$$[(M_I), (M_I \cdot M_a), (M_I \cdot M_a \cdot M_b)]$$

With matrices:

$$[\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix}]$$

We then compute the higher-order functions

$$\texttt{zipwith}(\phi \cdot \texttt{project}, [(M_I), (M_I \cdot M_a), (M_I \cdot M_a \cdot M_b)], \epsilon ab)$$

to produce the output string: $[12 + b]$

4.5 Implementing SFT on Parallel Hardware

Mytkowicz and Schulte describe their approach in terms of a modern multicore desktop. In this section, we demonstrate how to extend that work to a large cluster of machines using LINQ-to-HPC. LINQ-to-HPC is a data-parallel framework that translates declarative SQL like queries into a dataflow graph, which it then compiles to run on a large cluster [12]. Unfortunately, LINQ-to-HPC does not implement the data-parallel primitives `scan` and `zipwith` and thus we were forced to implement these primitives.

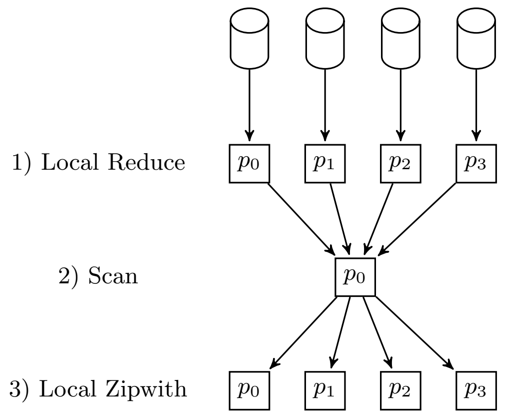
In particular, `scan` is non-trivial to implement efficiently[16, 19]. Our first implementation in LINQ-to-HPC that was based off the implementations detailed in prior work had terrible performance; in these implementations, at each step of the parallel algorithm, each processor has to communicate with another and thus parallel performance is dominated by communication. In a GPU, where on-chip memory is used for communication between threads this may be sufficient to get good parallel performance. However, in a LINQ-to-HPC cluster where communication occurs over the network, we were unable to get good performance.

Given this first failed attempt, our second and final implementation was optimized to reduce the amount of communication required during `scan` to a minimum. We did this by making each machine perform a *sequential* `scan` on large amounts of data before communicating the result of that `scan` to other processors in the cluster.

Our implementation is a few hundred lines of C#. The intuition behind our approach is that we can break the input up into sections, so each processor in a cluster works on an isolated contiguous section of the input. *If* each processor knew the starting state of the SFT for its section of the input, the problem would be embarrassingly parallel (e.g. each processor works in isolation over its part of the input). Our implementation of `scan` is designed to efficiently calculate the starting state of the SFT for each of the P processors in the cluster. Our implementation has the following steps:

1. `Local Reduce:` Given an input string s with $N = |s|$ symbols and P processors, each processor computes a local sequential reduction of $\hat{\delta}$ over N/P consecutive symbols in s. This results in P matrices where M_p is the partial reduction of $\hat{\delta}$ over processor p's section of the symbols in s.
2. `Scan:` One processor does a scan, using $\hat{\delta}$, of the P matrices computed in the prior step. After this scan, matrix M_p encodes the starting state of the SFT for processor p.
3. `Local Zipwith:` The problem has now become embarrassingly parallel: the prior two steps calculated the starting state of the SFT for each of the P processors in the cluster. Each of the P processors takes a second pass over its section of the input, calling $\phi \cdot project$ for each matrix

The following picture shows the communication pattern:



Note that our approach has little communication; each processor works in isolation in both the first and third steps. The only serialization of the algorithm occurs when a single processor does a scan over the partial reductions computed in the Step 1. If p is the number of processors, and n is the size of the input, then when $p << n$ our implementation will have good performance and scaling because p bounds the amount of communication—and thus performance-killing serialization—in the cluster.

5. Evaluation

This section is organized as follows. Section 5.1 talks about the exploration overhead. Section 5.2 discusses consistency of our Bek encoders and those from other libraries. Section 5.3 focuses on compiling to JavaScript, C, and C# from Bek. Finally, Section 5.4 talks about compiling Bek programs to run on a data-parallel cluster and discusses the significant throughput improvements achieved with this approach.

5.1 Exploration Overhead

Figure 6 shows the number of states in some representative encoders/decoders before exploration and with full exploration. These encoders match those extracted from Microsoft AntiXSS and other similar sanitization libraries [9]. Their Bek translations can be found online at `http://rise4fun.com/bek`. The speed of the exploration algorithms depends on the size of the reachable state space. For our examples, in most cases, this is small due to the restricted range of the values stored in the registers. The time to do full exploration is less than .1 seconds for the first six coders in Figure 6 while it takes around a second to fully explore `UTF8Decode`.

The case of `HtmlEncode` is special in the sense that the ST does not need a register and there is a simple loop over characters, so SFT generation is trivial in this case.

	States	
Coder	Original	Explored
`HtmlEncode`	1	1
`UTF8Encode`	2	5
`CssEncode`	2	5
`Base64Encode`	3	21
`Base64Decode`	6	87
`HtmlDecode`	4	113
`UTF8Decode`	9	1284

Figure 6: Input statistics on number of control states in unexplored STs and fully-explored SFTs. `HtmlDecode` here is the restricted version from [20].

First, we discuss the cost of our exploration algorithm in terms

Impl. Language	Routine	Running time 50	500
	`UTF8Encode` *implementations compared*		
C#	.NET	2	16
	Bek	2	14
	Bek explored	1	10
C	Bek explored	1.06	3.35
	`CSSEncode` *implementations compared*		
C#	AntiXSS hand-written	7	73
	Bek default	2	21
	Bek explored	2	25
C	Bek	2.44	14.28
	Bek explored	2.41	15.07
	`HtmlDecode` *implementations compared*		
C#	Bek default	<1	5
	Bek explored	<1	6
	`HtmlEncode` *implementations compared*		
C#	AntiXSS hand-written	6	55
	Bek default	3	25
	Bek explored	3	30

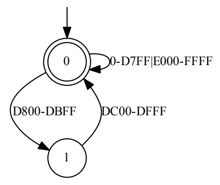
Figure 8: Running times for inputs of size 50 and 500.

of the speed to perform exploration and the number of states added.

SFTs often provide an exponential reduction, in terms of the size of the alphabet, compared to classical finite state transducers. Classical transducers would need in the order of 2^{16} transitions in all cases except for base64 coding where they need around 2^{12} transitions.

5.2 Consistency of Encoders

Our approach to checking the consistency of the Bek-generated sanitizers with the original versions relies on large-scale testing. We generate a set of 1,000 strings and evaluate both the original sanitizer and the generated code on each input. The strings are chosen randomly and then checked to ensure that they are accepted by the finite state automaton to the right to ensure that the inputs are legal. (The automaton represents all valid sequences of UTF16 encoded strings, that is the standard for in-memory representation of Unicode strings.)



We used independently-produced implementations in C# listed in Figure 7 for comparison. The independent implementations came from .NET 4.5 core libraries and the AntiXSS encoder library.

Routine	Lib.	Ver.	LOC
`CSSEncode`	AntiXSS	2.4	206
`UTF8Encode`	.NET	4.5	310
`HTMLEncode`	AntiXSS	2.4	110

Figure 7: Pre-existing coders used for comparison.

5.3 Serial Execution

We discuss client-side compilation to JavaScript and server-side compilation to C and C# in turn. We evaluated the running time of our client-side JavaScript implementations obtained from Bek using Google Chrome version 20.0.1132.47, build 144678, with the V8 JavaScript engine version 3.10.8.19. We run for 100 iterations each on two sets consisting of 1,000 randomly generated test strings. The first set contains strings of 50 characters while the second contains strings of 500 characters.

Client-side: compiling to JavaScript: The results for JavaScript compilation are shown in Figure 9. Overall, the running times for the Bek-provided implementation in JavaScript are comparable to those for other libraries. In some cases (`UTF8Encode`) we run slower, in some cases (`CSSEncode`) faster. We believe that part of our speed difference may arise from edge case checks performed by Bek code that are not in the other implementations. In practice, having encoders that run the same no matter where the code is executed is necessary to fluidly migrate code between the server and the client. The main advantage of using Bek is the ability to achieve parity with server-side implementations in JavaScript.

Impl. Language	Routine	Running time 50	500
`UTF8Encode` *implementations compared*			
JavaScript	PHP.JS	0.79	5.39
	WebTK	2.34	15.17
	Bek	9.83	88.45
`CSSEncode` *implementations compared*			
JavaScript	OWASP	196.73	1,976.02
	Bek	9.47	80.68
`HtmlDecode` *implementations compared*			
JavaScript	Bek	9.45	81.16
`HtmlEncode` *implementations compared*			
JavaScript	Bek	21.66	201.05

Figure 9: Client-side running times for inputs of size 50 and 500.

Server-side: Compiling to C and C#: Next, we focus on compiling to C# and C so that our encoders can run on the server. Figure 8 shows a speed comparison.

The C# running times are competitive with other library implementations, beating AntiXSS 3-fold for `CSSEncode` and 2-fold for `HtmlEncode`. Translating to C gives a considerable boost in terms of execution time, especially noticable for larger inputs (strings of length 500). Speed improvements range from $2x$ to about $5x$. These increases in execution time are consistent with the overall speed of a managed runtime such as .NET compared to a C version. We hypothesize that in our case, for small inputs, the overhead of explicit memory allocation calls to `malloc` and `free` dominates the execution time for `UTF8Encode` at length 500, the built-in .NET version is about 60% slower than the Bek-generated version and is almost 5 times slower than the C version.

5.4 Cloud: Compiling to a Data-Parallel Cluster

While sequential performance is of great practical value, our translation from Bek really shines when we use parallel hardware as a translation target. In this section we demonstrate our translation approach on a small LINQ-to-HPC cluster and show *multi-factor* performance improvements over a sequential baseline.

Platform: We conducted all experiments on an unloaded cluster of 32-machines. Each machine is an Intel 2 GHz (L5420) workstation with 16 GB of RAM per machine. During our experiments, one machine of the 32 is used as a "head node" to coordinate communication and schedule jobs across the cluster. This means we have 31 machines for core computation.

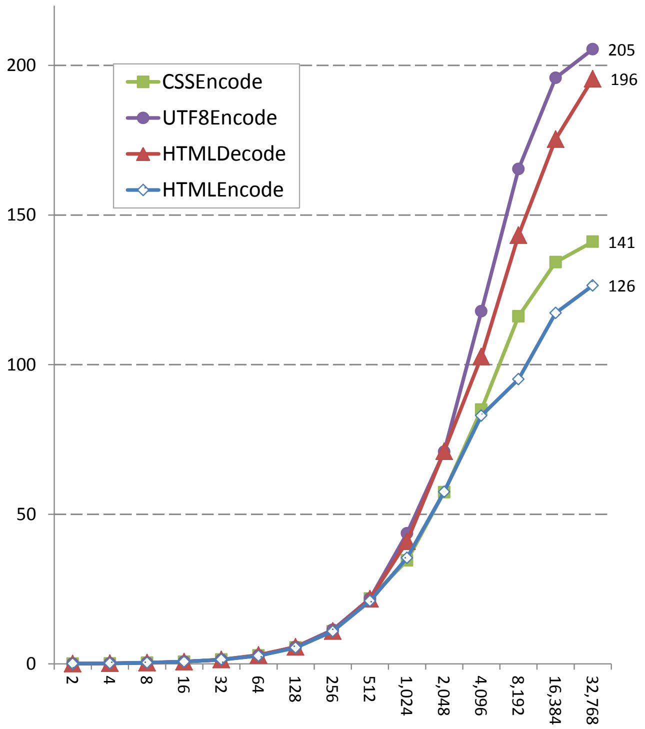


Figure 10: The throughput of various Bek encoders on large data, shown in GB/s on the y axis, as function on input size on the x axis.

Distributed Measurement: For our experiments in this section, we used a very large 32 GB HTML file, obtained from the web. We distributed the 32 GB of data to each of the 31 machines; each machine stored a little over 1 GB of HTML locally on its hard disk. We measured the time it takes to complete a single transduction of the HTML (e.g. reading HTML from disk, computing the transduction, and finally writing the transformed input to disk). We then computed the Bek program's throughput by dividing the number of bytes encoded by the time it takes to do the encoding. To get statistically significant results, we ran each experiment 10 times and reported the mean and 95% confidence interval of the mean.

Throughput of Data-Parallel Bek Programs: We evaluate the performance of the four Bek encoders introduced in Section 5.1: `CSSEncode`, `UTF8Encode`, `HTMLDecode`, `HTMLEncode`.

In Figure 10 we show our data-parallel implementations are much faster at $13x$, $9.5x$, $13.7x$, and $8.7x$ respectively, for `UTF8Encode`, `CSSEncode`, `HTMLDecode`, and `HTMLEncode` than their sequential C# implementations. To compute a baseline for each encoder, we ran the sequential C# encoders over a 32 GB data file obtained from the Bing search engine. We then ran each data-parallel version of the encoder on a 32 node cluster running the LINQ-to-HPC framework.

There are two interesting features of this graph.

- Our data-parallel Bek programs are fast: the throughput for 32 GB of data (far right point of x-axis) are 141, 205, 196, and 126 megabytes per second, respectively, for `CSSEncode`, `UTF8Encode`, `HTMLDecode`, and `HTMLEncode`, respectively. The reason for the difference is due to the amount of data each encoder writes. For example, most HTML input is already in UTF8 so for every byte in the input, the encoder writes a single byte. In contrast, HTML encoding sometimes writes more than one byte for any input byte (e.g. to encode the `&` character the encoder writes `&amp;`). Furthermore, `CSSEncode` does the most encoding and thus has the lowest throughput.

- We see nice scaling as we increase the size of the input from 1 megabyte up to 5,000 megabytes (i.e. as we move along the x axis). At this point, throughput of the algorithm ceases to scale. We suspect this is due to disk IO being the bottleneck in the computation. Because our approach is data-parallel, we expect that if we increase the size of the cluster, we can amortize this IO across more machines and thus get more scaling.

Overall, these order-of-magnitude throughput improvements across the board are significant and enable considerably larger amounts of cloud-based data processing than would be possible on a single machine.

6. Related Work

Symbolic finite transducers (SFTs) and Bek were originally introduced in [9] 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 [20], 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 [9, 20]. In contrast, the focus of the current paper and its motivation is efficient transformation from STs to SFTs with the particular application of *code generation* that supports efficient parallel execution. Another recent extension of SFTs, *extended* SFTs [5], is SFTs with "lookahead" where the primary motivation is to overcome limitations of SFTs in order to support analysis of decoders, unlike SFTs, ESFTs are *not closed* under composition.

In recent years there has been considerable interest in automata over infinite languages [18], starting with the work on *finite memory automata* [10], also called *register automata*. Finite words over an infinite alphabet are often called *data words* in the literature. Other automata models over data words are *pebble automata* [15] and *data automata* [4]. Several characterizations of logics with respect to different models of data word automata are studied in [3]. 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* [7] 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] provide another recent symbolic extension of finite transducers where the label theories are restricted to be total orders, in order to maintain decidability of equivalence. To the best of our knowledge, we do not know of prior work that has investigated the use of extensions of transducers for code generation.

In our implemenation we use the off-the-shelf SMT solver Z3 [6] for incrementally solving label constraints that arise during the exploration algorithm. Similar applications of SMT techniques have been introduced in the context of symbolic execution of programs by using path conditions to represent under and over approximations of reachable states [8]. The distinguishing feature of our exploration algorithm $\mathcal{E}$ is that it computes a transformation $\mathcal{E}(A)$ that is *equivalent* to the original ST A with respect to the transduction semantics, which is important for correct code generation, as opposed to other applications such as test case generation, where under-approximations are used, or verification where over approximations are used.

Finite state transducers have been used before for dynamic and static analysis to validate sanitization functions in web applications in [2], by an over-approximation of the strings accepted by the sanitizer using static analysis of existing PHP code. Other security analysis of PHP code, e.g., SQL injection attacks, use string analyzers to obtain over-approximations (in form of context-free grammars) of the HTML output by a server [13, 21]. Yu et.al. show how multiple automata can be composed to model looping code [22]. Our work is complementary to previous efforts in using SMT solvers to solve problems related to list transformations. HAMPI [11] and Kaluza [17] 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; as we have demonstrated, SFTs can be exponentially more succinct than classical finite transducers with respect to the alphabet size.

7. Conclusions

This paper demonstrates how to compile a domain-specific language called Bek to produce consistent and fast sanitizers across a range of languages, some server- and others client-based. We use symbolic finite state transducers as an intermediate language. We then introduce a novel algorithm which performs a symbolic partial evaluation of these transducers to obtain simplified, stateless versions of the original Bek program. We showed how to compile the resulting stateless transducers to both sequential and data-parallel hardware. Our compilation results in significant runtime improvements: our generated C# code outperforms the previous hand-tuned code by a factor of up to 3. Our data-parallel compilation achieves more impressive results of up to 13.7 times speedup on a 32-node cluster, compared to a sequential implementation.

References

[1] R. Alur and P. Cerný. Streaming transducers for algorithmic verification of single-pass list-processing programs. In *POPL'11*, pages 599–610. ACM, 2011.

[2] 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] M. Benedikt, C. Ley, and G. Puppis. Automata vs. logics on data words. In *CSL*, volume 6247 of *LNCS*, pages 110–124. Springer, 2010.

[4] M. Bojańczyk, A. Muscholl, T. Schwentick, L. Segoufin, and C. David. Two-variable logic on words with data. In *LICS*, pages 7–16. IEEE, 06.

[5] L. D'Antoni and M. Veanes. Static analysis of string encoders and decoders. In *VMCAI'13*, LNCS. Springer, 2013.

[6] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS'08*, LNCS, 2008.

[7] 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.

[8] P. Godefroid. Compositional dynamic test generation. In *POPL'07*, pages 47–54, 2007.

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

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

[11] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: a solver for string constraints. In *ISSTA*, 2009.

[12] Microsoft Corporation, 2011. http://msdn.microsoft.com/en-us/library/hh378101.aspx.

[13] Y. Minamide. Static approximation of dynamically generated web pages. In *WWW '05: Proceedings of the 14th International Conference on the World Wide Web*, pages 432–441, 2005.

[14] T. Mytkowicz and W. Schulte. Maine: a library for data parallel finite automata. Technical report, Microsoft Research, 2012.

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

[16] P. Sanders and J. L. Träff. Parallel prefix (scan) algorithms for MPI. In *Proceedings of the 13th European PVM/MPI User's Group conference on Recent advances in parallel virtual machine and message passing interface*, EuroPVM/MPI'06, pages 49–57, Berlin, Heidelberg, 2006. Springer-Verlag.

[17] 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.

[18] L. Segoufin. Automata and logics for words and trees over an infinite alphabet. In Z. Ésik, editor, *CSL*, volume 4207 of *LNCS*, pages 41–57, 2006.

[19] S. Sengupta, A. E. Lefohn, and J. D. Owens. A work-efficient step-efficient prefix sum algorithm, in: Workshop on edge computing using new commodity architectures, 2006.

[20] M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjorner. Symbolic finite state transducers: Algorithms and applications. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'12)*, 2012.

[21] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su. Dynamic test input generation for web applications. In *ISSTA*, 2008.

[22] F. Yu, T. Bultan, and O. H. Ibarra. Relational string verification using multi-track automata. In *Proceedings of the 15th international conference on Implementation and application of automata*, CIAA'10, pages 290–299, 2011.