Data-Parallel String-Manipulating Programs

Margus Veanes David Molnar Todd Mytkowicz Benjamin Livshits
Microsoft Research

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 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 finite state transducers, which extend classical transducers with symbolic predicates. 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 outperforms the previous 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 32GB 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, however, is more general that Web sanitization and can 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 [21].

The strength of BEK is that it enables 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. Combined with the analyses possible for BEK programs, the language can be applied widely.

After analysis, we need to actually execute the program. Generating efficient code from BEK is a novel challenge and the subject of this work. Through a series of case studies, we demonstrate a compilation step that generates faster code than manually-written string-manipulating routines.We combine a novel *exploration algorithm* described in this paper 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 BEK to the LINQ-to-HPC data-parallel framework running on a cluster.

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, the key point is that BEK 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, 21].

1.2 Symbolic Branching Transducers

Previous work provided a semantics for BEK in terms of symbolic finite transducers (SFTs). An SFT is an extension of a traditional finite transducer in which edges may be labeled with formulas instead of concrete characters. Recall that a finite transducer is a generalization of deterministic

```
program Decode67(t) {
  return iter(c in t)[r := 0;]{
     case((r==0)&&(c>='6')&&(c<='7')):
        r := c;

     case((r!=0)&&(c>='6')&&(c<='7')):
        yield(10*(r-48)+c-48);
        r:=0;

     case((r==0)&&(c!='6')&&(c!='7')):
        yield(c);

     case(true):
        yield(r,c);
        r:=0;
  };
}
```

Figure 1. Sample Bek program.

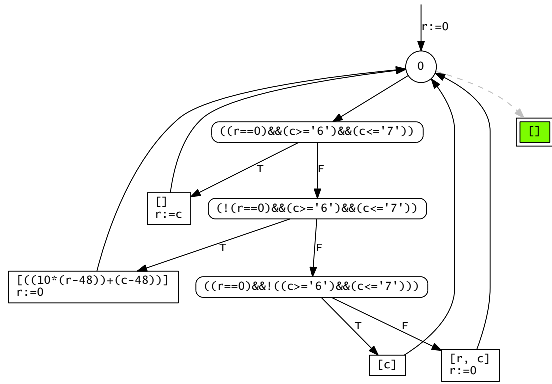


Figure 2. Depiction of the ST^b corresponding to the program in Figure 1. The dashed arrow corresponds to the "final rule," which terminates execution when fired. Oval nodes correspond to branch conditions and rectangular nodes correspond to basic rules.

finite automaton that allows transitions from one state to another to be annotated with *outputs*: if the input character matches the transition, the automaton outputs a specified sequence of characters. In symbolic finite transducers [21], transitions are annotated with logical *formulas* instead of specific characters, and the transducer takes the transition on any input character that satisfies the formula.

We introduce a novel semantics for Bek programs in terms of *symbolic branching transducers*, an extension of SFTs that allow for additional structure. Our symbolic representation allows us to use *satisfiability modulo theories (SMT) solvers*, tools that take a formula and attempt to find inputs satisfying the formula. These solvers have become robust in the last several years and are used to solve complicated formulas in a variety of contexts. At the same time, our representation allows leveraging automata-theoretic methods to reason about strings of unbounded length, which is not possible via direct encoding to SMT formulas. SMT solvers allow working with formulas from any theory supported by the solver, while other previous approaches, such as using binary decision diagrams, are specialized to specific types of inputs.

We extend SFTs with *branching rules* and *exception rules*, called ST^bs. These branching rules allow us to precisely model exceptions, which was not possible with SFTs. At the same time, the extension is a conservative extension of SFTs. Branching rules do not increase expressiveness, but they allow code generation to preserve the original branching structure and avoid duplication of branch conditions. For analysis tasks, such as domain equivalence and partial equivalence, we first need to eliminate registers in order to use the SFT algorithms introduced in [21]. We introduce an algorithm for partial evaluation of ST^bs that preserves the branching structure of rules. As a special case, the algorithm also works for symbolic transducers without branching rules. As an example, Figure 2 shows the ST^bcorresponding to the Bek program in Figure 1.

1.3 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. Recent advances in data-parallel compilation of finite transducers allow us to automatically translate Bek programs without registers into parallel implementations, which we describe in Section 4.

Fortunately, a key contribution of this work is a novel *partial symbolic evaluation* algorithm that enables *removing registers* from Bek programs. We describe the algorithm in detail in Section 3. While the algorithm is not guaranteed to work in all cases, when it does work it produces a ST^bthat 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. In particular, eliminating register state is difficult to perform manually because it requires reasoning about the different behavior exhibited by the program for different values of the register.

1.4 Paper Contributions

This paper has the following contributions.

- We extend symbolic transducers with branching rules. The purpose of this extension is to more faithfully support code generation by preserving branching structure and avoiding duplication of branch conditions.
- Previous work on Bek introduced an extension to symbolic transducers with *registers* [21]. Registers, such as `r` in Figure 1, 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* for ST^bs that is complete for finite-valued register update functions and works modulo arbitrary background theories. The algorithm, if it terminates, outputs new ST^bs 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 unsatisfi-

ablity checks to prune provably unreachable states as a dynamic backward reachability analysis.

- 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 to obtain a fully-automatic data-parallel compilation of BEK programs into the LINQ-to-HPC framework. To the best of out 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.

1.5 Paper Organization

The rest of this paper is organized as follows. Section 2 introduces symbolic branching transducers (ST$^{\rm b}$s). 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 Branching Transducers

We now formally introduce symbolic branching transducers or ST$^{\rm b}$s and give examples of how ST$^{\rm b}$s 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.

We use τ, σ and γ to denote types, and we write $\mathcal{U}^\tau$ for the corresponding sub-universe of elements of type τ. The Boolean type is bool, with $\mathcal{U}^{\mathsf{bool}} = \{\mathsf{t}, \mathsf{f}\}$, the integer type is int, and the type of k-bit bit-vectors is $\mathsf{bv}k$. The Cartesian product type of types σ and γ is $\sigma \times \gamma$. The type σ^* is the type for finite sequences of elements of type σ. The universe $\mathcal{U}^{(\sigma^*)}$ is the Kleene closure $(\mathcal{U}^\sigma)^*$ of the universe $\mathcal{U}^\sigma$. We also write type σ^k as a semantic subtype of σ^* of sequences of elements of length *at most* $k \geq 0$.

Terms and formulas are defined by induction over the background language and are assumed to be well-typed. The type τ of a term t is indicated by $t:\tau$. Terms of type bool, or Boolean terms, are treated as formulas, i.e., no distinction is made between formulas and Boolean terms. All elements in $\mathcal{U}$ are also assumed to have corresponding constants in the background language and we use elements in $\mathcal{U}$ also as constants. The set of free variables in a term t is denoted by $FV(t)$, t is *closed* when $FV(t) = \emptyset$, and closed terms t have Tarski semantics $[\![t]\!]$ over the background structure. Substitution of a variable $x:\tau$ in t by a term $u:\tau$ is denoted by $t[x/u]$. In the following we let $\Sigma = \mathcal{U}^\sigma$, $\Gamma = \mathcal{U}^\gamma$ and $Q = \mathcal{U}^\tau$.

A *λ-term* f is an expression of the form $\lambda x.t$, where $x:\sigma$ is a variable, and $t:\gamma$ is a term such that $FV(t) \subseteq \{x\}$; the type of f is $\sigma \to \gamma$; $[\![f]\!]$ denotes the function that maps $a \in \Sigma$ to $[\![t[x/a]]\!] \in \Gamma$. As a convention, we use f and g to stand for λ-terms. A λ-term of type $\sigma \to \mathsf{bool}$ is called a *σ-predicate*. We write φ and ψ for σ-predicates and, for $a \in \Sigma$, we write $a \in [\![\varphi]\!]$ for $[\![\varphi]\!](a) = \mathsf{t}$. We often treat $[\![\varphi]\!]$ as a *subset* of Σ. Given a λ-term $f = (\lambda x.t):\sigma \to \gamma$ and a term $u:\sigma$, $f(u)$ stands for $t[x/u]$. A predicate φ is *unsatisfiable* when $[\![\varphi]\!] = \emptyset$; *satisfiable*, otherwise.

The main building block of an ST$^{\rm b}$ is a *rule*. A rule is an expression that denotes a partial function corresponding to a straight-line conditional statement of a program that may *yield outputs*, *produce updates*, and *raise exceptions*. We first provide an inductive definition of rules that omits type annotations. We then define additional well-formedness criteria and the semantics for rules.

- $\bot$ is the *exception rule*.
- If f is a λ-term then $br(f)$ is a *basic rule*.
- If φ is a predicate and r_1, r_2 are rules then $ite(\varphi, r_1, r_2)$ is an *if-then-else (ite) rule*.

We say that a rule r is *well-formed* with respect to the type $\sigma \to \gamma$, denoted $r : \sigma \to \gamma$, when one of the following conditions holds:

- r is the rule $\bot$.
- r is a rule $br(f:\sigma \to \gamma)$.
- r is a rule $ite(\varphi:\sigma \to \mathsf{bool}, r_1:\sigma \to \gamma, r_2:\sigma \to \gamma)$.

A rule $r:\sigma \to \gamma$ represents a function $[\![r]\!]$ from $\mathcal{U}^\sigma$ to $\mathscr{P}(\mathcal{U}^\gamma)$ For all $a \in \mathcal{U}^\sigma$:

$$\begin{array}{rcl} [\![\bot]\!](a) & \stackrel{\rm def}{=} & \emptyset \\ [\![br(f)]\!](a) & \stackrel{\rm def}{=} & \{[\![f]\!](a)\} \\ [\![ite(\varphi, r_1, r_2)]\!](a) & \stackrel{\rm def}{=} & \begin{cases} [\![r_1]\!](a), & \text{if } a \in [\![\varphi]\!]; \\ [\![r_2]\!](a), & \text{otherwise.} \end{cases} \end{array}$$

We now introduce the central definition of a symbolic branching transducer that uses the definition of rules.

Definition 1: A *Symbolic Branching Transducer* or ST$^{\rm b}$ A with *input* type σ, *output* type γ and *state* type τ is a tuple (q^0, R, F), where

- $q^0 \in \mathcal{U}^\tau$ is the *initial state*;
- R is a finite set of rules of type $(\sigma \times \tau) \to (\gamma^k \times \tau)$, for some $k \geq 0$, rules in R are called the *input rules* of A;
- F is a finite set of rules of type $\tau \to \gamma^k$, for some $k \geq 0$, rules in F are called the *final rules* of A.

For a basic subrule $r = br(\lambda(x, y).\langle f(x, y), g(x, y)\rangle)$ of an input rule, f is called the *yield* and g the *update* of r. A basic subrule of a final rule is called a *final yield*. ⊠

We write $p \xrightarrow{a/\boldsymbol{b}}_A q$ for a *concrete transition* of A: it means that there exists $r \in R_A$ such that $[\![r]\!](a, p) = \{(\boldsymbol{b}, q)\}$. Similarly, we write $q \xrightarrow{/\boldsymbol{b}}_A$ for a *final output* of A: it means that there exist $r \in F_A$ such that $[\![r]\!](q) = \{\boldsymbol{b}\}$. Intuitively, a final output is a special case of an input-epsilon move of a classical finite state transducer into a final state, but it is algorithmically useful to keep final rules separate from general input-epsilon moves. Unlike input-epsilon moves in general, final rules do not affect the core algorithms, while providing a very convenient mechanism to yield additional outputs upon reaching the end of the input tape.

We write $A^{\sigma/\gamma;\tau}$ to indicate the input/output types σ/γ and the state type τ of an $\mathrm{ST^b}$ A.

The *reachability relation* $p \xrightarrow{\boldsymbol{a}/\boldsymbol{b}}\!\!\!\!\rightarrow_A q$ for $\boldsymbol{a} \in \Sigma^*$, $\boldsymbol{b} \in \Gamma^*$, and $p, q \in Q$ is defined through the closure under the following conditions, where '$\cdot$' is concatenation of sequences:

- For all $q \in Q$, $q \xrightarrow{\epsilon/\epsilon}\!\!\!\!\rightarrow_A q$.
- If $p \xrightarrow{\boldsymbol{a}/\boldsymbol{b}}\!\!\!\!\rightarrow_A p'$ $\xrightarrow{a/\boldsymbol{c}}_A q$ then $p \xrightarrow{\boldsymbol{a}\cdot a/\boldsymbol{b}\cdot\boldsymbol{c}}\!\!\!\!\rightarrow_A q$.

Definition 2: The *transduction* of an $\mathrm{ST^b}$ A, $\mathscr{T}_A$, is the following function from Σ^* to $\mathscr{P}(\Gamma^*)$.

$$\mathscr{T}_A(\boldsymbol{a}) \stackrel{\mathrm{def}}{=} \{\boldsymbol{b}\cdot\boldsymbol{c} \mid \exists\, q \in Q\,(q_A^0 \xrightarrow{\boldsymbol{a}/\boldsymbol{b}}\!\!\!\!\rightarrow_A q \xrightarrow{/\boldsymbol{c}}_A)\}$$

⊠

We say that A is *single-valued* when $|\mathscr{T}_A(\boldsymbol{a})| \leq 1$ for all $\boldsymbol{a} \in \Sigma^*$. A is *deterministic* when $|R_A| = 1$ and $|F_A| = 1$. Note that determinism immediately implies single-valuedness. The definition of $\mathrm{ST^b}$s is consistent with the definition of STs in [21].

In the rest of the paper we will only consider deterministic $\mathrm{ST^b}$s and we will identify R_A (resp. F_A) with the rule it contains. The exploration algorithm can be extended to nondeterministic $\mathrm{ST^b}$s, but for most analysis tasks the $\mathrm{ST^b}$s are required to be at least single-valued. For the data-parallel translation explained in Section 4 the $\mathrm{ST^b}$s are required to be deterministic, that is naturally the case for the kinds of string transformations we have in mind with this approach.

The following example illustrates the use of $\mathrm{ST^b}$s on a typical string transformation scenario and introduces the concrete language BEK that we use for describing $\mathrm{ST^b}$s in this paper.

Example 1 Let the input type, output type, and state type be `bv7`. (Intuitively $\mathcal{U}^{\mathtt{bv7}}$ corresponds to the set of ASCII characters). The BEK program in Figure 1 corresponds to an $\mathrm{ST^b}$ that decodes certain occurrences of pairs of digits by their corresponding ASCII letters. For example `Decode67("a77")` is `"aM"`. The initial state is 0 that corresponds to the initial value of register r. The case statements map to the following input rule where we lift the λ-prefix to be in the front:

$$\begin{array}{ll}\lambda(c,r). & ite(r = 0 \wedge \texttt{'6'} \leq c \leq \texttt{'7'},\\ & \quad br([], c),\\ & \quad ite(r \neq 0 \wedge \texttt{'6'} \leq c \leq \texttt{'7'},\\ & \quad\quad br([(10 * (r-48)) + (c-48)], 0),\\ & \quad\quad ite(r = 0 \wedge \neg(\texttt{'6'} \leq c \leq \texttt{'7'}),\\ & \quad\quad\quad br([c], r),\\ & \quad\quad\quad br([r, c], 0))))\end{array}$$

The final rule is $\lambda r.[]$ (produces the empty sequence). There are no exception rules, i.e., the $\mathrm{ST^b}$ is a total function over strings of ASCII characters. The graphical illustration of the $\mathrm{ST^b}$ for `Decode67` is shown in Figure 2. All graphs in the paper are produced automatically from our analysis framework. This example is also available in the online version of BEK[1]. ■

3. Exploration of $\mathrm{ST^b}$s

In this section we develop an algorithm that allows us to eliminate either all or some of the state registers used in a deterministic $\mathrm{ST^b}$ A. In particular, we focus on two cases: *Full* exploration and *Boolean* exploration.

[1] http://www.rise4fun.com/Bek/Nv4

$\boldsymbol{Explore}(A^{\sigma/\gamma;\tau_1\times\tau_2}) \stackrel{\mathrm{def}}{=}$
$\quad p^0 := \mathit{first}(q_A^0);$
$\quad q^0 := \mathit{second}(q_A^0);$
$\quad S := \mathit{stack}(p^0);$
$\quad P := \{p^0\};$
$\quad \boldsymbol{Add}(p) \stackrel{\mathrm{def}}{=}$ **if** $p \notin P$ **then** $P := P \cup \{p\};\ \mathit{Push}(S, p);$
$\quad R := \{\mapsto\};$
$\quad F := \{\mapsto\};$
$\quad$**while** $S \neq \emptyset$
$\qquad p := \mathit{Pop}(S);$
$\qquad R(p) := \mathit{Expl}(\lambda y : \tau_2.\mathrm{t}, \mathit{Inst}(\lambda y : \tau_2.\mathrm{t}, R_A, p), \boldsymbol{Add});$
$\qquad F(p) := \mathit{Expl}(\lambda y : \tau_2.\mathrm{t}, \mathit{Inst}(\lambda y : \tau_2.\mathrm{t}, F_A, p), \boldsymbol{Add});$
$\quad$**return** $(p^0, q^0, R, F);$

$\boldsymbol{Inst}(\varphi, \bot, p) \stackrel{\mathrm{def}}{=}$ **return** $\bot;$
$\boldsymbol{Inst}(\varphi, br(f, g), p) \stackrel{\mathrm{def}}{=}$ **return** $br(\lambda y.f(p, y), \lambda y.g(p, y));$
$\boldsymbol{Inst}(\varphi, ite(\psi, t, f), p) \stackrel{\mathrm{def}}{=}$
$\quad \varphi_t := \lambda y.\varphi(y) \wedge \psi(p, y);$
$\quad \varphi_f := \lambda y.\varphi(y) \wedge \neg\psi(p, y);$
$\quad$**if** $\mathit{IsUnsatisfiable}(\varphi_t)$ **return** $\mathit{Inst}(\varphi_f, f, p);$
$\quad$**else if** $\mathit{IsUnsatisfiable}(\varphi_f)$ **return** $\mathit{Inst}(\varphi_t, t, p);$
$\quad$**else return** $ite(\lambda y.\psi(p, y), \mathit{Inst}(\varphi_t, t, p), \mathit{Inst}(\varphi_f, f, p));$

$\boldsymbol{Expl}(\varphi, \bot, _) \stackrel{\mathrm{def}}{=}$ **return** $\bot;$
$\boldsymbol{Expl}(\varphi, ite(\psi, t, f), \boldsymbol{Add}) \stackrel{\mathrm{def}}{=}$
$\quad$**return** $ite(\varphi, \mathit{Expl}(\varphi \wedge \psi, t, \boldsymbol{Add}), \mathit{Expl}(\varphi \wedge \neg\psi, f, \boldsymbol{Add});$
$\boldsymbol{Expl}(\varphi, br(f, g), \boldsymbol{Add}) \stackrel{\mathrm{def}}{=}$
$\quad \psi := \lambda y\, z\,.\varphi(y) \wedge (z = \mathit{first}(g(y)));$
$\quad r := \bot;$
$\quad$**while** $\exists M \models \psi$
$\qquad$let $p = z^M;$
$\qquad r := ite(\lambda y.p = \mathit{first}(g(y)), br(f, \lambda y.\mathit{second}(g(y)), p), r);$
$\qquad \psi := \lambda y\, z\,.\psi(y, z) \wedge z \neq p;$
$\qquad \boldsymbol{Add}(p);$
$\quad$**return** $r;$

Figure 3. Exploration algorithm of $\mathrm{ST^b}$s.

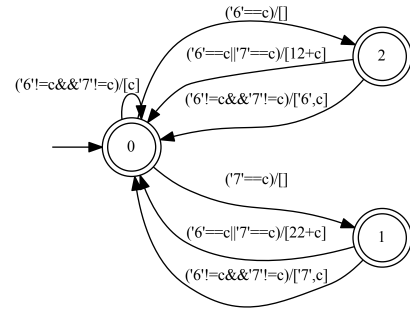


Figure 4. Full exploration of `Decode67` in Figure 1.

For the purpose of explaining the exploration algorithm, we extend $A = (q^0, R, F)$ with a component P that is a finite set of *control states* and an initial control state $p^0 \in P$. The sets R and F are extended to be maps from P to rules, and each basic subrule of an input rule in R has an additional control state component $p \in P$. With this extension in mind, we write a basic rule as $br(\mathit{yield}, \mathit{update}, p)$. We say that A is *stateless* when the register type τ is the unit type $\mathsf{T0}$ ($\mathcal{U}^{\mathsf{T0}} = \{\langle\rangle\}$), i.e., registers are not used in a stateless $\mathrm{ST^b}$, and thus R has the equivalent form

$$\{p_1 \mapsto r_1, p_2 \mapsto r_2, \ldots, p_{|R|} \mapsto r_{|R|}\}$$

where each rule r_i corresponds to a conditional statement that may yield outputs and transition to new control states but does not make use of registers by storing intermediate results in registers. This extension is useful for separation of concerns, it helps to keep the control state separate from the data state.

For example, the $\mathrm{ST^b}$ is Example 1 is not stateless because the rules depend on the register r.

By *full exploration* of A, we mean a construction of a stateless $\mathrm{ST^b}$ $A^{\mathbf{f}}$ such that $\mathscr{T}_A = \mathscr{T}_{A^{\mathbf{f}}}$, i.e., A and $A^{\mathbf{f}}$ are equivalent. Full exploration is not always possible, because equivalence of stateless $\mathrm{ST^b}$s reduces to equivalence of symbolic finite transducers (SFTs), and equivalence of SFTs is decidable [21] modulo a decidable label theory, while equivalence of $\mathrm{ST^b}$s is undecidable already for very restricted decidable label theories. Even when full exploration is possible, $A^{\mathbf{f}}$ may still be exponentially larger than A.

By *Boolean exploration* of A, we mean a construction of an $\mathrm{ST^b}$ $A^{\mathbf{b}}$ such that $\mathscr{T}_A = \mathscr{T}_{A^{\mathbf{b}}}$ where all Boolean registers of A have been eliminated. For example, if the state type of A is $(\mathsf{bool} \times \mathsf{bool}) \times \mathsf{int}$ then the the state type of $A^{\mathbf{b}}$ is int, i.e., the two Boolean registers have been eliminated by adding new control states.

Note that, in order to completely eliminate the symbolic update of a rule $br([], \lambda(x,y).\varphi(x))$, where φ is a σ-predicate, i.e., to replace φ by $\lambda x.\mathsf{t}$ (resp. $\lambda x.\mathsf{f}$) we would need to decide if $\forall x\, \varphi(x)$ holds, i.e., $\neg\varphi$ is unsatisfiable, (resp. if $\forall x\, \neg\varphi(x)$ holds, i.e., φ is unsatisfiable).

3.1 Algorithm *Explore*.

The generic exploration algorithm of $\mathrm{ST^b}$s is described in Figure 3. The algorithm takes as its input an $\mathrm{ST^b}$ A, and assumes a projection of the state type τ of A into two parts τ_1 and τ_2. We assume, without loss of generality, that $\tau = \tau_1 \times \tau_2$. The algorithm uses an SMT solver, as a black box, to decide satisfiability and to generate models for formulas. No assumptions are made about the particular types. In the concrete case studies the types are typically numeric (bitvector, integer or real), but could also be algebraic datatypes or array types.

The algorithm generates a new $\mathrm{ST^b}$ by exploring the rules with respect to τ_1, effectively eliminating τ_1, i.e. turning the first state component into an explicit control state. At the topl level, the algorithm is a depth first search algorithm, starting from the initial state q_0

In order to avoid special cases, we may always assume that either τ_1 or τ_2 can be unit types $\mathsf{T0}$ ($\mathcal{U}^{\mathsf{T0}} = \{\langle\rangle\}$). Now, full exploration of A corresponds to the case when τ_2 is the unit type, and Boolean exploration corresponds to the case when τ_1 is a Cartesian combination of Boolean registers and τ_2 is a Cartesian combination of all the non Boolean registers.

Inst *Inst*(φ,r,p) creates an *instance* of the rule r with the path condition φ with respect to the fixed register values given by p. For the exception rule this is a noop. For a basic rule this is a partial instantiation of the yield and update with respect to p, where $\lambda y.f(p,y)$ instantiates the first projection of the state register with the value p. An important point for the rules is that unreachable rule instances are eliminated by deciding satisfiability of corresponding accumulated path conditions. For best utilization of the architecture of modern SMT solvers, the path conditions are accumulated implicitly by pushing and popping logical contexts of the SMT solver, e.g., *Inst*(φ_f, f, p) is evaluated in a newly pushed context where $\psi(p,y)$ has been asserted, provided that the context is still satisfiable. This step is similar to how path conditions are incrementally constructed during symbolic execution for example in the context of parameterized unit testing. Here the use of branching rules is needed for exploiting logical contexts.

Expl *Expl*(φ,r,*Add*) performs partial exploration of r with respect to τ_1 (or the projection function *first*). For the exception rule the operation is a noop. For an ite rule, the step is a direct propagation of the concretizations of the branches. The core of the computation takes place during concretization of basic rules. Suppose for simplicity that τ_2 is empty, or equivalently that $\tau_1 = \tau$, i.e., cosider the case of full exploration. Notice that f (yield) and g (update) do not depend on the register variable that has been eliminated at this point due to *Inst*. The variable y here is the input parameter, that should, if possible, remain symbolic. The while loop searches for all possible *distinct* interpretations of $g(y)$ such that $\varphi(y)$ holds. Suppose that $g(y)$ is the arithmetic expression $(y \mod 3)$ and that φ is the condition $(y \mod 3) > 0$. Then ψ is initially the fromula $(y \mod 3) > 0 \wedge z = (y \mod 3)$. A model $M \models \psi$ exists, e.g., $z^M = 1$. Now 1 is pushed to the search stack and ψ is strengthened to be $(y \mod 3) > 0 \wedge z = (y \mod 3) \wedge z \neq 1$. The rule r is case-split using the new value of $z = 1$. The step is repeated yielding another solution $z = 2$. A third solution does not exist and the while loop terminates returning the rule

$$\begin{aligned} ite(2 = (y \mod 3), \quad & br(f, \langle\rangle, 2), \\ & ite(1 = (y \mod 3), br(f, \langle\rangle, 1), \bot)) \end{aligned}$$

that is now case-split according to all the possible feasible values of the original register.

Theorem 1: *Let A be a deterministic* $\mathrm{ST^b}$ *with state type $\tau_1 \times \tau_2$. If Explore(A) terminates then the result is an* $\mathrm{ST^b}$ *that is equivalent to A and whose state type is τ_2.*

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 of $\mathcal{U}^{\tau_1}$. The first point is already needed in the *Inst* procedure that eliminates unsatisfiable branches. The second point is needed both, for termination of construction of r in *Expl*, as well as for guaranteeing that the search stack S is bounded in the top level depth first search loop of the algorithm. A sufficient condition for the second point is when the functions used for computing the first state projection have the *finite-range* property, i.e., when $\mathcal{U}^{\tau_1}$ can be assumed to be finite.

Example 2 The $\mathrm{ST^b}$ after full exploration of `Decode67` from Figure 1, is illustrated in Figure 4. The unexplored $\mathrm{ST^b}$ (in Figure 2) has a single control state 0, while the fully explored $\mathrm{ST^b}$ has 3 control states. ■

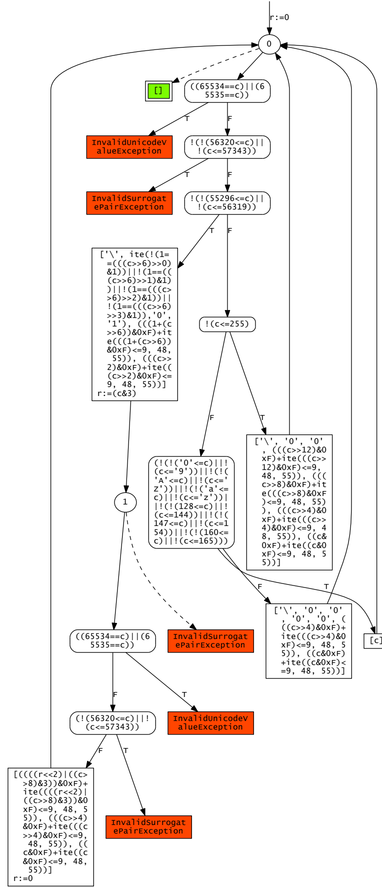


Figure 5. ST^{b} corresponding to the Web sanitizer `CssEncode` after Boolean exploration. The original ST^{b}contained a register which has been eliminated by exploration. Full exploration introduces three more control states and enables us to apply the matrix data-parallel approach.

4. Data-Parallel Translation

In this section we describe how to compile a symbolic ST^{b} so it can exploit data-parallel hardware. In particular, we demonstrate an end to end compilation of ST^{b} to a large cluster running LINQ to HPC [5].

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 finite automata be small in order to be efficient.

Our key insight that allows us to combine ST^{b} and the approach of Mytkowicz and Schulte is that ST^{b} exploration reduces the number of states in the ST^{b} and pushes the complexity of the ST^{b} into the edges, which in turn allows us to efficiently target data-parallel hardware.

4.1 Data-Parallel Operators

Before we get into the details of our translation, we introduce two higher-order data-parallel primitives. These primitives are well-known data-parallel operators and are easy to implement on a variety of hardware platforms.

`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 ST^{b} in terms of these primitives.

4.2 Describing ST^{b} With Higher-Order Functions

Recall that a ST^{b} is a tuple (q^0, R, F) where q^0 is the initial state, R is a finite set of *input rules*, and F is a finite set of *final rules*. Each rule defines a transition from a pair of input symbol and state to an output symbol and a new state. Let $\phi(q, s)$ 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 trigered by a special “end-of-input” symbol and leads to a unique final state.)

To transduce a string s by a ST^{b}, the ST^{b} starts in state q^0 and sequentially reads the symbols of s. When the ST^{b} 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 symbols in the output alphabet.

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

$$\texttt{Transduce}(\mathrm{ST}^{\mathrm{b}}, s) = \texttt{zipwith}(\phi, \texttt{scan}(\delta, q_0, s), s)$$

4.3 Translating ST^{b} 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 a ST^{b}. Finally, we compile from the ST^{b} 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 that corresponds to the full exploration of the Bek program `Decode67` in Figure 1, the fully explored ST^{b} is also depicted in Figure 4.

```
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 $\mathrm{ST^b}$ 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;
      }
    }
  }
}
```

To build a data-parallelversion of this $\mathrm{ST^b}$ we need two functions δ and ϕ. We alter the syntax directed translation of a $\mathrm{ST^b}$ 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 $\mathrm{ST^b}$

The prior section formalized $\mathrm{ST^b}$ 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 $\mathrm{ST^b}$ with an associative δ and ϕ. Fortunately, as we saw in the previous section, the exploration algorithm can remove registers in many cases.

We now demonstrate how to turn δ and ϕ into associative operations on vectors and matrices. Our insight is that matrix multiplication is an associative operation that encodes graph traversals.

Graph Traversals with Matrix Multiplication: A convenient way to view $\mathrm{ST^b}$ is as a graph where nodes in the graph are states in the set of states Q 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 $s \in \Sigma$ can be described by M_s, a $n \times n$ adjacency matrix, where $n = |Q|$, 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, a adjacency matrix is a symbolic representation of how a symbol from Σ transitions *every* state in a $\mathrm{ST^b}$.

Given this formulation, we use matrix multiplication as a mechanism for graph traversal; if the adjacency matrix that encodes the state of the $\mathrm{ST^b}$ after reading the empty string, ϵ, is M_I, then the adjacency matrix that encodes the state $\mathrm{ST^b}$ 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 $\mathrm{ST^b}$ 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 $\mathrm{ST^b}$ to Matrices: To transform a $\mathrm{ST^b}$ to operations on vectors and matrices, we define the following two functions, `inflate` and `project`. [`inflate`] generates a matrix from each input symbol $s \in \Sigma$. 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 $\mathrm{ST^b}$ 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

$$V_F^T = \begin{pmatrix} 0, & 1, & \ldots, & \mathrm{n} \end{pmatrix}$$

Given this formulation, we can now 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 $\mathrm{ST^b}$, and s_i is the ith symbol in string s.

With an associative version of δ, we can now implement a data parallel version of `Transduce` as:

$$\texttt{Transduce}(\mathrm{ST^b}, s) = \\ \texttt{zipwith}(\phi, \texttt{map}(\texttt{project}, \texttt{scan}(\hat{\delta}, M_I, s)), s)$$

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:

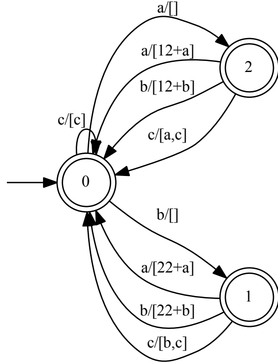
$$\texttt{Transduce}(\mathrm{ST^b}, s) = \texttt{zipwith}(\phi \cdot \texttt{project}, \texttt{scan}(\hat{\delta}, M_I, s), s)$$

In other words, given a $\mathrm{ST^b}$, this section describes an automatic method to compile a $\mathrm{ST^b}$ into an implementation that is suitable to data-parallel hardware. The end-to-end pipeline is shown in Figure 6. 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 alpbabet $\mathcal{U}^{\mathrm{bv7}}$, but it is convenient for the exposition to view them as three distinct characters.

In this view, the $\mathrm{ST^b}$ has control states $\{0, 1, 2\}$, input alphabet $\Sigma = \{a, b, c\}$, initial state 0, and whose δ and ϕ can be depicted as:



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 $\mathrm{ST^b}$ 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 $\texttt{Transduce}(s)$, 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 $\mathrm{ST^b}$ on Parallel Hardware

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 [5]. 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[17, 20]. 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.

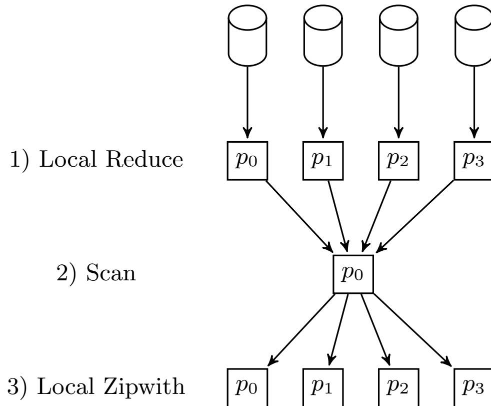
Our implementation is a few hundred lines of C#. The intuition behind our approach is that we can break the input up into sections, so each processor in a cluster works on an isolated contiguous section of the input. *If* each processor knew the starting state of the $\mathrm{ST^b}$ for its section of 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 $\mathrm{ST^b}$ 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 $\mathrm{ST^b}$ for processor p.
3. `Local Zipwith:` The problem has now become embarrassingly parallel: the prior two steps calculated the start-

Figure 6. The pipeline for compiling from Bek to a data-parallel implementation. We start with a Bek program, then compile the program to a symbolic branching transducer. From the transducer we derive the associative operators ϕ and $\hat{\delta}$. We plug these operators into the data-parallel primitives `scan` and `project` along with the input string s and a matrix M that encodes the state of the transducer. Finally we feed these to the `zipwith` primitive which in turn yields the output `out`.

ing state of the ST^b 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 7 shows the number of states in four representative sanitizers before exploration, compared with Boolean 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, this is small due to restricted range of the values stored in the registers. The time to do full exploration is less than .1 second for the encoders in Figure 7.

	States	
Sanitizer	Original	Explored
`UTF8Encode`	1	5
`CssEncode`	1	5
`HtmlDecode`	1	113
`HtmlEncode`	1	113

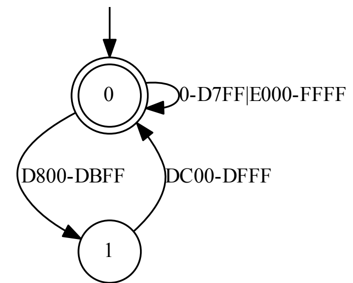
Figure 7. Input statistics and number of control states in unexplored and fully-explored transducers.

First, we discuss the cost of our exploration algorithm in terms of the speed to perform exploration and the number of states added.

In terms of states, ST^bs 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. This enables us to perform additional analysis on ST^bs, by reduction to STs, that would not scale on the corresponding classical transducers. In this section, we use suffix $_F$ refers to *full exploration.*

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.



We used independently-produced implementations in C# listed in Figure 8 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
`HTMLDecode`	AntiXSS	2.4	110
`HTMLEncode`	AntiXSS	2.4	110

Figure 8. Pre-existing sanitizers used for comparison.

5.3 Serial Execution

We discuss client-side complication to JavaScript and server-side compilation to C and C# in turn. We evaluated

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 9. Running times for inputs of size 50 and 500.

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 forJavaScript compilation are shown in Figure 10. Overall, the running times for the Bek-provided implementatatin 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 10. 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 9 shows a speed comparison.

The C# running times are competitive with other library implementatoins, 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.

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 11 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 `&`). Furthermore, `CSSEncode` does the most encoding and thus has the lowest throughput.

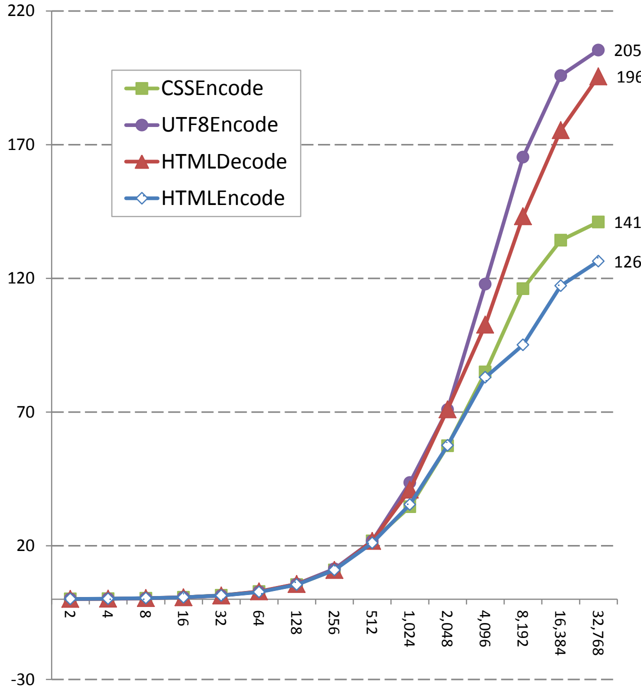


Figure 11. The throughput of various BEK encoders on large data, shown in GB/s on the y axis, as function on input size on the x axis.

- We see nice scaling as we increase the size of the input from 1 megabyte up to 5,000 megabytes (e.g. 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 key properties that are studied in [9] from a practical point of view are idempotence, commutativity and equivalence checking of sanitizers. The formal foundations and the theoretical analysis of the underlying SFT algorithms, in particular, an algorithm for deciding equivalence of SFTs, modulo a decidable background theory is studied in [21], including a more general *1-equality* algorithm that factors out the decision problem for single-valuedness, and allows non-determinism without violating single-valuedness. The formalism of SFTs is also extended in [21] to Symbolic Transducers (STs) that allow the use of registers. The focus of the current paper and the motivation is *code generation*. The two extensions of STs that we introduce in the current paper are *conditional branching* and *exception handling*. From the point of view of analysis, such as equivalence checking, the branching rules do not offer more expressiveness, because the branching rules can be flattened, however the branching rules maintain the evaluation order of conditions, and, more importantly, maintain the semantics of *exception handling* that is essential for correct code generation. Exploration algorithms for STs are not studied or analyzed in [9, 21], nor is efficient compilation.

In recent years there has been considerable interest in automata over infinite languages [19], starting with the work on *finite memory automata* [11], 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. 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.

Streaming transducers [1] provide another recent symbolic extension of finite transducers where the label theories are restricted to be total orders, in order to maintain decidability of equivalence. Streaming transducers are largely orthogonal to SFTs or the extension of STs with branching rules, as presented in the current paper. For example, streaming transducers allow reversing the input, which is not possible with $\mathrm{ST^b}$s, while arithmetic is not allowed in streaming transducers but plays a central role in our applications of $\mathrm{ST^b}$s to string encoders. The work in [16] introduces a different symbolic extension to finite state transducers called *predicate-augmented finite state transducers*. Besides identities, it is not possible to establish functional dependencies from input to output that are needed for example to encode transformations such as `UTF8Encode`.

We use the SMT solver Z3 [6] for incrementally solving label constraints that arise during the exploration algorithm. Similar applications of SMT techniques have been introduced in the context of symbolic execution of programs by using path conditions to represent under and over approximations of reachable states [8]. The distinguishing feature of our exploration algorithm is that it computes a precise transformation that is symbolic with respect to input labels, while allowing different levels of concretization with respect to the state variables. The resulting $\mathrm{ST^b}$ is not an under or over approximation, but functionally *equivalent* to the original $\mathrm{ST^b}$. This is important for correct code generation, as opposed to other applications such as test case generation, where under approximations are used, or verification of safety properties, where over approximations are used.

Finite state transducers have been used 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, 22]. Yu et.al. show how multiple automata can be composed to model looping code [23]. Our work is complementary to previous efforts in using SMT solvers to solve problems related to list transforma-

tions. HAMPI [12] and Kaluza [18] extend the STP solver to handle equations over strings and equations with multiple variables. The work in [10] shows how to solve subset constraints on regular languages. We are not aware of previous work investigating the use of finite transducers for efficient code generation. One obvious explanation for this is that classical finite transducers are not directly suited for this purpose; as we have demonstrated, finite state ST^{b}s can be exponentially more succinct than classical finite transducers with respect to alphabet size.

7. Conclusions

We showed 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. At the core of Bek is a novel representation called symbolic branching transducers. We introduced a new *exploration* algorithm on symbolic branching transducers that performs partial symbolic evaluation and can in some cases remove register state that impedes parallelism. We showed how to compile the resulting finite state transducers to 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 acheives more impressive results of up to 13.7 times speedup on a 32-node cluster, compared to a sequential implementation.

8. Acknowledgments

We would like to thank Lubomir Litchev, Barry Dorrans, Wolfram Schulte, Ben Zorn, Bryan Sullivan, Jim Larus, Tom Ball, and Bryan Parno for their valuable comments, insights, and practical help with this work.

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] M. Corporation, 2011. `http://msdn.microsoft.com/en-us/library/hh378101.aspx`.

[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] P. Hooimeijer and W. Weimer. A decision procedure for subset constraints over regular languages. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 188–198, New York, NY, USA, 2009. ACM.

[11] 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.

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

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

[14] 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] G. V. Noord and D. Gerdemann. Finite state transducers with predicates and identities. *Grammars*, 4:263–286, 2001.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

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

[23] 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.