Fusing Effectful Comprehensions

Olli Saarikivi
Aalto University and
Helsinki Institute for Information Technology HIIT
Finland
olli.saarikivi@aalto.fi

Margus Veanes       Todd Mytkowicz       Madan Musuvathi
Microsoft Research
Redmond, USA
{margus,toddm,madanm}@microsoft.com

Abstract
List comprehensions provide a powerful abstraction mechanism for expressing computations over ordered collections of data declaratively without having to use explicit iteration constructs. This paper puts forth effectful comprehensions as an elegant way to describe list comprehensions that incorporate loop-carried state. This is motivated by operations such as compression/decompression and serialization/deserialization that are common in log/data processing pipelines and require loop-carried state when processing an input stream of data.

We build on the underlying theory of symbolic transducers to fuse pipelines of effectful comprehensions into a single representation, from which efficient code can be generated. Using background theory reasoning with an SMT solver, our fusion and subsequent reachability based branch elimination algorithms can significantly reduce the complexity of the fused pipelines. Our implementation shows significant speedups over reasonable hand-written code (3.4×, on average) and traditionally fused version of the pipeline (2.6×, on average) for a variety of examples, including scenarios for extracting fields with regular expressions, processing XML with XPath, and running queries over encoded data.

1. Introduction
List comprehensions provide a powerful mechanism for declaratively specifying a pipeline of computations on collections of data. Programmers specify the various stages of the pipeline concisely and modularly without using explicit iteration constructs, while the runtime ameliorates the cost of the abstraction by performing various optimizations such as fusion/deforestation [36, 45].

This paper extends this idea to effectful comprehensions, an elegant way to describe list comprehensions that incorporate loop-carried state. As a motivation, consider the problem of analyzing logs as shown in Figure 1. The log on the disk (or coming across the network from a file server) is compressed, and thus the user has to first decompress the input stream of bits into bytes which are then deserialized into objects in a higher-level language, such as Java. In this example, the application selects stock prices from each object and looks for price dips — decreases followed by increases. The output is then serialized and compressed before being written back to disk. Such processing from input stream of bits to output stream of bits is not uncommon today. For instance, the processing in a single node of a data-processing system [4, 9, 19, 47], is similar to the one shown in Figure 1.

Note that the stages in the pipeline include both “functional” computations that operate on each input independently, such as SelectPrice, and “effectful” computations that iterate over the input list while maintaining loop-carried state, such as Decompress, Deserialize, and FindPriceDips. The goal of this paper is to allow such pipelines to be declaratively and modularly specified as shown at the bottom of the figure, then fuse them to a single representation for which efficient code can be generated. We use a variation of symbolic transducers [43] as our program representation.

In order to provide some intuition we consider a concrete but simplified example scenario of such a pipeline, consisting of two symbolic transducers. The situation that we consider is a fairly typical one when the raw input data is unstructured text, for example when parsing CSV files. Raw text is most commonly assumed to be UTF8 encoded. Suppose that the task is to parse and extract a nonnegative integer from
where an input stream of bits goes through various stages
is in the range 2. Figure 2(a) shows a symbolic transducer.
A transition to symbolic transducers.

Motivating example of a log processing pipeline
that are the decoded Unicode character codes. For simplicity
a sequence of bytes and produces a sequence of integers
and then parses an integer (here ASCII digits which in turn have a single byte UTF8 encoding. We will now work through the steps of the fusion, which builds a product of the reachable control states starting from the initial pair state \((q_0, p_0)\). For example, fusion of the transition \(q_0 \xrightarrow{x \in [0-0x7F]/[x:0]} q_1\) with the transition \(p_0 \xrightarrow{x \in [0x30-0x39]/[x:0-x:30]} p_1\) produces the transition \((q_0, p_0) \xrightarrow{x \in [0x30-0x39]/[x:0-x:30]} (q_0, p_1)\) where the fused register is a pair representing the registers of Utf8Decode and ToInt and the output \(x\) from Utf8Decode has been consumed as the input of ToInt. When the producer (here Utf8Decode) outputs nothing, the consumer (here ToInt) remains in the same state. So in the fusion of Utf8Decode and ToInt there is a product transition \((q_0, p_1) \xrightarrow{(x \in [0x30-0x39]/[x:0-x:30], \pi_2(x))} (q_1, p_1)\) where \(\pi_1\) and \(\pi_2\) project the first and second element of a pair, respectively. The only possible fusion of transitions from \((q_1, p_1)\)
is

\[
x \in [0x80 - 0xBF] \land (\pi_1(x) \in [0x30 - 0x39]/[x:0-x:30], (x \& 0x3F) < 6) \land (\pi_2(x) = 0, (x + 10 \& 0x3F) + (x \& 0x3F) < 0x30) \\
\]

\[
(q_1, p_1) \rightarrow (q_0, p_1)\]

However, the state \((q_1, p_1)\) is associated with the register constraint \(3x \in [0x20 - 0xDF] \land \pi_1(x) = (x \& 0x3F) < 6)\) which together with the guard of the transition from \((q_1, p_1)\) becomes unsatisfiable. Thus the transition can be removed from the fused transducer, which in turn implies that the state \((q_1, p_1)\) has become a dead-end and the transitions to it can be eliminated, since any execution ending up in \((q_1, p_1)\) is guaranteed to finally reject. Similar reasoning allows us to remove \((q_1, p_0)\). The fusion ends up being identical to ToInt.

Observe that the story would be quite different if ToInt accepted non-ASCII digits. Often fusion eliminates a lot of the complexity in the early stages in the pipeline by backpropagating the particular constraints required by the later stages, such as, the only accepted input characters being digits.

As data moves to later stages in the pipeline the data-types tend to become more structured and filtered.
```csharp
IEnumerable<int> Utf8ToInt(IEnumerable<byte> input) {
    int r1 = 0; bool multiByte = false;
    var endState = input.SelectMany(x => { // Utf8Decode
        if (!multiByte) {
            if (0 <= x && x <= 0x7F) yield return x;
            else if (0xC2 <= x && x <= 0xDF) {
                yield return r1 | (x & 0x3F);
                multiByte = true;
            } else throw new Exception();
        } else {
            if (0x80 <= x && x <= 0xBF) {
                yield return r1 | (x & 0x3F);
                multiByte = false;
            } else throw new Exception();
        }
    }).Aggregate(new { r2 = 0, defined = false },
        (s, x) => { // ToInt
            if (0x30 <= x && x <= 0x39) return new { r2 = (10 * s.r2) + x - 0x30,
                defined = true };
            else throw new Exception();
        });
    if (!endState.defined) throw new Exception();
    yield return endState.r2;
}
```

Figure 3. Utf8Decode and ToInt in LINQ.\(^3\)

The scenario that we have just illustrated gives some insight as to what kind of analysis is used in our fusion engine. It uses an SMT solver [18] to decide satisfiability of constraints over the element domains and uses forward and backward reachability techniques to prune unreachable transitions. Such analysis goes far beyond what compilers can do today, techniques that are used in stream fusion [16, 28] or in composition of symbolic finite state transducers [43].

For our techniques to be widely applicable to real-world programs there must be an accessible way to specify effectful comprehensions. One possibility is using existing libraries for writing list comprehensions. Figure 3 presents a function implementing a pipeline of the Utf8Decode and ToInt comprehensions using C#’s LINQ [30] library.\(^4\) Utf8Decode is represented as a SelectMany, which allows producing variable amounts of output. Since SelectMany does not encapsulate state usage, Utf8Decode uses ad-hoc state in the form of local variables, which complicates analyses by potentially allowing different stages in the pipeline to communicate through shared state. Because ToInt’s Update does not produce output it can be represented with Aggregate, which does encapsulate state. However, writing effectful comprehensions that do partial state updates with Aggregate is cumbersome, since returning the new state disallows specifying only the parts that change.

To address these concerns we present a C# interface (Section 5.1) for specifying effectful comprehensions that encapsulates state usage. The interface is similar to ones found in existing streaming libraries (Section 7). We translate programs that implement this interface into symbolic transducers. Additionally, we provide specialized frontends for parsing scenarios based on regex and XPath matching.

We evaluate the efficacy of our approach on a variety of data processing pipelines that decode, parse, compute, and then serialize back to disk. These pipelines exhibit common real-world scenarios of extracting data with regexes, querying XML files with XPath, and working with encoded data. On average, our fused code is 3.4× faster than reasonable handwritten code and 2.6× faster than versions fused with method calls. We further demonstrate that our conservative reachability analysis and subsequent pruning based on background theory reasoning can significantly reduce the complexity of these fused pipelines. The contributions of this paper are:

- A variation of symbolic transducers with branching rules, which simplify analysis and code generation.
- An algorithm for fusing symbolic transducers.
- A branch elimination algorithm based on reachability analysis which complements the satisfiability based branch elimination built into the fusion algorithm.
- A frontend for specifying effectful comprehensions and a strategy for translating these into symbolic transducers. Additionally, we provide frontends for regex and XPath based parsing scenarios.
- A comprehensive evaluation demonstrating the efficacy of our approach.

2. Symbolic Transducers

This section formally introduces branching symbolic transducers or BSTs, as a generalization of deterministic symbolic finite transducers or deterministic SFTs [43] by incorporating registers. At the same time the definition is a specialization of nondeterministic symbolic transducers [43] since nondeterminism is disallowed. The specialization is reflected in the way individual transitions are defined. Rather than using flat transitions from a single source state to a single target state, we use branching transitions called rules that may have multiple target states. The two main reasons for this specialization are: 1) it makes determinism an integral part of the definition rather than a property; 2) it preserves the original program’s structure and supports more efficient serial code generation.

Generating good serial code from flat symbolic transitions would be challenging as a short-circuiting evaluation scheme for shared subformulas would have to be selected from a potentially large search space. Moreover, the choices may be data-dependent, and ultimately depend on the domain knowledge from the user. The following example exhibits an instance of such a choice.

To concretely illustrate branching transitions or rules, consider the example transducer Utf8Decode from Figure 2(a). Instead of two flat transitions from state \(q_0\) (one looping back to state \(q_0\) and one transitioning to state \(q_1\)) the BST has a single rule from each state, as illustrated in Figure 4(a), where \(\bot\) corresponds to an implicit rejecting state that would be added.

---
\(^3\)We ignore C#’s limitation that `yield` is not allowed in lambda functions.
\(^4\)The code for other list comprehension libraries, such as Java 8’s Streams API, is largely similar.
to Figure 2(a) after completion. In Utf8Decode the order of the two input byte conditions from state $q_0$ is important if we suppose that ASCII characters are most frequent. If the two conditions in the branching rule from state $q_0$ were reordered so that the test $x \in [0\times C2 \sim 0\times DF]$ is applied first, then that test would be failing for most of the input characters.

The initial register value is $0$. The basic rules are the leaf transitions of the branches and are labeled $s; g$ where $s$ is the output sequence and $g$ the updated register value. Figure 4(b) illustrates the ToInt transducer with branching transitions. Here the finalizers are represented as rules, since the one from $p_1$ outputs the value stored in the register. In general also finalizers could have branching rules.

Before formally defining rules we introduce some general notations. Given types $\tau$ and $\sigma$, $\tau \times \sigma$ and $\tau \rightarrow \sigma$ stand for the standard Cartesian product and function types, respectively.\(^5\)

The type for Booleans is $\text{bool}$ with truth values $\text{true}$ and $\text{false}$. Let $\mathcal{T}(\tau)$ denote a given predefined set of terms $t$ that denote values $[t]$ of type $\tau$. In our implementation we use Z3\(^1\) expressions for $\mathcal{T}(\tau)$ but the general definition is not restricted to any fixed representation. Further, our implementation constructs no terms of the form $\mathcal{T}(\tau \rightarrow \sigma \rightarrow \rho)$, for which there is no direct representation or decision procedures in Z3. In general our theory and algorithms work with any decidable theory. A term in $\mathcal{T}(\tau \rightarrow \text{bool})$ is a $\tau$-predicate.

Let $[\tau]$ denote the type of finite-length lists of elements of type $\tau$. A list of type $[\tau]$ is denoted by $[t_1, \ldots, t_n]$ or $[t_i]_{i=1}^{n}$ where $n \geq 0$ and each $t_i$ is a term of type $\tau$. We assume that if $\tau$ is a Cartesian product type $\tau_1 \times \tau_2$ then there are projection functions $\pi_1 : \tau \rightarrow \tau_1$ and $\pi_2 : \tau \rightarrow \tau_2$ and a pairing function $\langle, \rangle : \tau_1 \times \tau_2 \rightarrow \tau$ with the intended semantics that $\langle [t_1], [t_2] \rangle = ([t_1], [t_2])$, and $\pi_1([t_1], [t_2]) = [t_1]$, and $\pi_2([t_1], [t_2]) = [t_2]$. Observe that if $t_1 \in \mathcal{T}(\tau_1)$ and $t_2 \in \mathcal{T}(\tau_2)$, then $\langle t_1, t_2 \rangle \in \mathcal{T}(\tau_1 \times \tau_2)$, and if $t \in \mathcal{T}(\tau_1 \times \tau_2)$ then $\pi_1(t) \in \mathcal{T}(\tau_1)$, and $\pi_2(t) \in \mathcal{T}(\tau_2)$.

Every type $\tau$ denotes a nonempty set and has a default element $\_\tau$ (or $\_\tau$). We write $\tau$ both for a type and the denoted set and we write $\pi_1$ for $[\tau_1]$ when this is clear from the context.

A branching symbolic transducer or BST is a tuple $(\iota, o, \rho, Q, q^0, r^0, \delta, \$)$, where $\iota$, $o$ and $\rho$ are the input, output and register types; $Q$ is the finite set of control states; $\$ = $q^0, r^0$ is the initial state of state type $\sigma \equiv Q \times \rho$; $\delta : Q \rightarrow \mathcal{R}(\iota \times \rho, o, Q, \rho)$, $\$ : $Q \rightarrow \mathcal{R}(\rho, o, Q, \rho)$ are, respectively, the transition function and the finalizer, where elements of $\mathcal{R}(\tau, o, Q, \rho)$ are called rules. A rule is, in effect, a tree structure, where every interior node is an $\text{Ite}$ (“if-then-else”) choice and every leaf is either a $\text{Base}$ case that performs a state transition or an Underf case that represents a transition to an implicit rejecting state $\bot$. Such a tree is interpreted as a function of type $\tau \rightarrow (\langle o \times \sigma \rangle \cup \{\bot\})$ or a partial function of type $\tau \rightarrow o \times \sigma$, where $\sigma = Q \times \rho$. A rule for the transition function has $\tau = i \times \rho$ because it makes decisions based on both the input symbol and the register value; while a rule for the finalizer has $\tau = \rho$ because it makes decisions based solely on the register value. Formally, a rule in $\mathcal{R}(\tau, o, Q, \rho)$ has one of the following three forms:

- $\text{Ite}(\varphi, f, t)$, where $\varphi \in \mathcal{T}(\tau \rightarrow \text{bool})$ and $t$ and $f$ are rules.
- $\text{Base}([f_i]_{i=1}^{n}, q, g)$, where $n \geq 0$, $\{f_i\}_{i=1}^{n} \subseteq \mathcal{T}(\tau \rightarrow o)$, $q \in Q$, and $g \in \mathcal{T}(\tau \rightarrow \rho)$.
- Underdef.

A rule $r$ is interpreted as a function $[r]$ in this manner:

$$[\text{Ite}(\varphi, f, t)]v = \begin{cases} [t]v, & \text{if } [\varphi]v = \text{true} \\ [f]v, & \text{otherwise} \end{cases}$$

$$[\text{Base}([f_i]_{i=1}^{n}, q, g)]v = ([f_i]v)_{i=1}^{n}, q, [g]v$$

$$[\text{Underdef}]v = \bot$$

The finalizer is used to produce a final output list upon reaching the end of the input list. It is a generalization of a final state. Intuitively one may think of the finalizer as being a special case of the transition function that is triggered by a unique end-of-input symbol. However, unlike in the classical setting, formally such a symbol cannot in general be treated as an element of type $\iota$. Instead of lifting every input type $\iota$ to a sum type of $\iota$ and an end-of-input symbol, end-of-input is handled separately by the finalizer.

We use the following variable naming conventions of terms occurring in rules. In a term $t$ occurring in a rule, variable $x$ is of type $\iota$ and refers to the input element and variable $r$ is of type $\rho$ and refers to the register. To disambiguate between variables and functions that appear in formulas from those used in our definitions, proofs and algorithms, we use a mono-space font for the former. For example in $x = t$ the $x$ is a literal part of the formula, while $t$ refers to some term.
Simultaneous substitution of variables $y_i$ by terms $u_i$ in $t$ is denoted $t\{u_1/y_1, \ldots, u_n/y_n\}$.

In $\text{Utf8Decode}$, in Figure 4(a), the finalizer is depicted as $q_0$ being accepting and $q_1$ being non-accepting in the classical sense, meaning that the finalizer is the function:

$$\$\text{Utf8Decode} = \{ q_0 \mapsto \text{Base}([], q_0, 0), q_1 \mapsto \text{Undef} \}$$

The finalizer of $\text{TolInt}$, in Figure 4(b), that is shown as the dashed arrow, is the function:

$$\$\text{TolInt} = \{ p_0 \mapsto \text{Undef}, p_1 \mapsto \text{Base}([x], p_1, 0) \},$$

where the final value of the register $x$ is output upon reaching the end of the input list in the control state $p_1$, whereas the initial control state $p_0$ is not valid as a final state and the input would be rejected if the input list terminates in this state.

A BST $A$ denotes a transduction $[A]$ that is a partial function of type $\ [a] \rightarrow [o]$. First, we define the partial functions $\hat{\delta} : \lambda \rightarrow [\sigma] \times \sigma$ and $\hat{\delta} : \lambda \rightarrow [\sigma] \times \sigma$ that enable us to provide a declarative definition of $[A]$:

$$[A] \hat{\delta} a = [\hat{\delta} q](a, b); \quad [A] \hat{\delta} a = [\hat{\delta} q](a, b)$$

Let $\hat{a} = (a_i)_{i=1}^k$ be a given input list. Then

$$[A] \hat{\delta} a = [\pi_1((\hat{\delta} a_1) \oplus \cdots \oplus (\hat{\delta} a_k) \oplus \hat{\delta}s^0)]$$

where $\oplus$ is a left-associative operator that composes single-input transduction steps into multi-input transduction steps:

$$\oplus : [\sigma] \rightarrow [\sigma] \times [\sigma] \rightarrow [\sigma] \rightarrow [\sigma] \rightarrow [\sigma]$$

$$F_1 \oplus F_2 \equiv \lambda s. \text{let } (u_1, s_1) = (F_1 s) \text{ in } (u_1, + s_2)$$

where $+\equiv \text{list concatenation}$, $\oplus$ is called step composition.

If we depict $[\delta A a s_{\text{fin}}] = (u, s_{\text{out}})$ and $[\delta A s_{\text{fin}}] = (u, \_)$ by

$$s_{\text{fin}} \rightarrow A \rightarrow s_{\text{out}} \quad \text{and} \quad s_{\text{fin}} \rightarrow A$$

respectively, then

$$a_1 \rightarrow A \rightarrow A \rightarrow A \rightarrow A \rightarrow A \rightarrow A \rightarrow A \rightarrow A$$

$$u_1 \rightarrow u_2 \rightarrow u_3 \rightarrow u_4 \rightarrow u_5 \rightarrow u_6 \rightarrow u_7 \rightarrow u_8 \rightarrow u_9$$

$$a_2 \rightarrow a_3 \rightarrow a_4 \rightarrow a_5 \rightarrow a_6 \rightarrow a_7 \rightarrow a_8 \rightarrow a_9 \rightarrow a_{10}$$

$$s_0 \rightarrow A \rightarrow A \rightarrow A \rightarrow A \rightarrow A \rightarrow A \rightarrow A \rightarrow A$$

$$u_0 \rightarrow u_1 \rightarrow u_2 \rightarrow u_3 \rightarrow u_4 \rightarrow u_5 \rightarrow u_6 \rightarrow u_7 \rightarrow u_8$$

depicts Equation (1), where ‘↓’ shows list comprehension, similar to for example $\text{SelectMany}$ list comprehension in LINQ, and ‘→’ shows state evolution. For example let $\hat{a} = [a_1, a_2] = [0x05, 0x93]$ and $A = \text{Utf8Decode}$. Then

$$\begin{aligned}
A &\rightarrow (q_0, 0) \\
A &\rightarrow (q_1, (a_1 \& 0x3F)) \rightarrow A \\
A &\rightarrow (q_0, 0) \\
A &\rightarrow \text{Undef}
\end{aligned}$$

$$\begin{aligned}
A &\rightarrow (q_1, (a_2 \& 0x3F)) \rightarrow A \\
A &\rightarrow (q_0, 0) \\
A &\rightarrow \text{Undef}
\end{aligned}$$

The result is $\[] + \ [(a_1 \& 0x3F) \& 0] + (a_2 \& 0x3F) + [\[]$ that equals $\{0x153\}$ and represents the string "aw".

When the transition function or the finalizer maps to $\perp$ then the transduction is considered to be undefined for the corresponding input. Alternatively, one may choose to work with total functions and use a designated rejecting control state $q_\perp$ such that $\hat{\delta} q_\perp = \text{Undef}$ and, for all $\nu$, $[\text{Undef}] \nu = (\{\}, q_\perp, \_)$.

3. Fusion of BSTs

Consider two BSTs $A$ and $B$ such that $o_A = \nu_B$. We want to fuse $A$ and $B$ into a single BST $A \otimes B$ such that $[A \otimes B]$ is equivalent to $[A] \circ [B]$, i.e., $\lambda x.[B][[A](x)]$. We first main the idea behind the construction. We then explain the incremental algorithm that makes the composition scale in practice. The control-state complexity of the algorithm is $|Q|^2$. The worst-case complexity with respect to the size of the rules is also quadratic, even when the number of control states is small. It is therefore instrumental to prune unreachable states early and to develop incremental algorithms.

3.1 Main Idea

At a high level, the fusion algorithm of $A \otimes B$ can be described as follows. $A \otimes B$ has the following components: $\nu = \nu_A \otimes \nu_B$, $\rho = \rho_A \times \rho_B$, $Q \subseteq Q_A \times Q_B$, $v^0 = (v_{A}^0, v_{B}^0)$, $q^0 = (q_{A}^0, q_{B}^0)$. The goal of the fusion algorithm is to construct $\delta_{A \otimes B}$ and $\$\$A \otimes B$. See Figure 5.

For each pair $(\nu, q)$ of control states in $Q_A \times Q_B$ build a fused rule that, given the rule $\delta_A \rho$, symbolically runs $\delta_B$ repeatedly, starting from $q$, over each of the output lists $[v_i]_{i=1}^n$ that occur in the $\text{Base}$-subrules of $\delta_A \rho$ as symbolic values. The symbolic values are substituted into the register update and output functions of $[\delta_B v_1] \oplus \cdots \oplus [\delta_B v_n]$, that is partially evaluated with respect to the control state $q$, and finally normalized into a rule in $R(\rho, o, Q, \rho)$. The finalizer is constructed similarly.

While such brute force approach will terminate in theory, because the output lists have a fixed length that is independent of the input element, it is highly impractical for several reasons. One problem is control state space size, because $|Q| = |Q_A||Q_B|$. Another problem is output-branch explo-
There are several key optimizations used in the construction of composed rules, powered by the use of an SMT solver for satisfiability checks and model generation. One technique is to incrementally check for unsatisfiability and validity of guards of newly formed `Ite`-rules and to remove branches that are inaccessible and consequently also eliminate control states that become inaccessible. The distinction between control states and registers is instrumental because finiteness of control states guarantees termination and enables techniques not directly available over infinite state spaces.

We provide a top-down view of the fusion algorithm in Figure 6 with further helper procedures in Figure 7. Fusion is implemented using depth first search starting from \((p, q) = (q_A^0, q_B^0)\). Only satisfiable parts of composite rules are ever explored. The procedure `Fuse\(_\delta\)(\(\gamma, R, q\))` in Figure 6 uses an accumulating context condition \(\gamma\) for a branch of an `Ite`-rule of \(A\) with \(R\) as the unexplored subrule in that context, and \(q\) is a control state of \(B\). If the condition `SAT\((\gamma \land R_1^0 \neq R_2^0)\)` is false then for all \((x, r) \in [\gamma]\), \([R_1^0](x, r) = [R_2^0](x, r)\), so the branching condition is redundant. The condition \(R_1^0 \neq R_2^0\) is itself, w.l.o.g., expressible as a \(\times \pi\)-predicate. The newly discovered states in the depth first search are added to the `Frontier` in line 8 of the definition of `Prod` in Figure 6. Elements of \(Q_A \times Q_B\) that are never added to `Frontier` are unreachable and thus irrelevant.

To construct a rule, the mutually recursive `Run\((\gamma, \bar{v}, q, s)\)` and `Step\((\gamma, \bar{v}, \text{rest}, R, s)\)` procedures shown in Figure 7 symbolically execute the step composition operator \(\oplus\) for \(B\) over the symbolic value list \(\bar{v}\) starting from the state \((q, s)\) of \(B\). The satisfiability checks in `Step` on lines 5 and 8 maintain that the constructed rules only have branches that are feasible and non-redundant. A trivial case of redundancy is when both \(R_1^0\) and \(R_2^0\) are `Undef`, but more complicated conditional cases may arise when \(R_1^0\) and \(R_2^0\) are syntactically different but semantically equivalent in the given context \(\gamma\).

### 3.2 Incremental Fusion Algorithm

There are several key optimizations used in the construction of composed rules, powered by the use of an SMT solver for satisfiability checks and model generation. One technique is to incrementally check for unsatisfiability and validity of guards of newly formed `Ite`-rules and to remove branches that are inaccessible and consequently also eliminate control states that become inaccessible. The distinction between control states and registers is instrumental because finiteness of control states guarantees termination and enables techniques not directly available over infinite state spaces.

We provide a top-down view of the fusion algorithm in Figure 6 with further helper procedures in Figure 7. Fusion is implemented using depth first search starting from \((p, q) = (q_A^0, q_B^0)\). Only satisfiable parts of composite rules are ever explored. The procedure `Fuse\(_\delta\)(\(\gamma, R, q\))` in Figure 6 uses an accumulating context condition \(\gamma\) for a branch of an `Ite`-rule of \(A\) with \(R\) as the unexplored subrule in that context, and \(q\) is a control state of \(B\). If the condition `SAT\((\gamma \land R_1^0 \neq R_2^0)\)` is false then for all \((x, r) \in [\gamma]\), \([R_1^0](x, r) = [R_2^0](x, r)\), so the branching condition is redundant. The condition \(R_1^0 \neq R_2^0\) is itself, w.l.o.g., expressible as a \(\times \pi\)-predicate. The newly discovered states in the depth first search are added to the `Frontier` in line 8 of the definition of `Prod` in Figure 6. Elements of \(Q_A \times Q_B\) that are never added to `Frontier` are unreachable and thus irrelevant.

To construct a rule, the mutually recursive `Run\((\gamma, \bar{v}, q, s)\)` and `Step\((\gamma, \bar{v}, \text{rest}, R, s)\)` procedures shown in Figure 7 symbolically execute the step composition operator \(\oplus\) for \(B\) over the symbolic value list \(\bar{v}\) starting from the state \((q, s)\) of \(B\). The satisfiability checks in `Step` on lines 5 and 8 maintain that the constructed rules only have branches that are feasible and non-redundant. A trivial case of redundancy is when both \(R_1^0\) and \(R_2^0\) are `Undef`, but more complicated conditional cases may arise when \(R_1^0\) and \(R_2^0\) are syntactically different but semantically equivalent in the given context \(\gamma\).

### Figure 6. Fusion \(A \oplus B\) of BSTs \(A\) and \(B\) with \(o_A = \iota_B\).

Definition of `Run` is given in Figure 7.

Observe how the procedure `Fuse\(_\delta\)` uses \(\gamma\) on lines 5–7: \(\gamma\) is included as a conjunct in every solver call to `SAT` and every recursive call to `Fuse\(_\delta\)`. This pattern of use allows incremental SMT solving, where the solver is used in such a way that subsequent solver calls can reuse previously learned clauses. For example, on line 5 in `Fuse\(_\delta\)` this would be implemented by pushing \((\varphi \theta)\) into the solver context before the recursive call and popping the context afterwards. In fact, both procedures `Fuse\(_\delta\)` and `Step` use the parameter \(\gamma\) in a way such that \(\gamma\) is included as a conjunct in (i) each call to `SAT`, and (ii) each \(\gamma\) argument in recursive calls. Furthermore when `Fuse\(_\delta\)` calls `Step` on line 11 it passes its \(\gamma\) as an argument. Therefore, each call to `Fuse\(_\delta\)` can use a single solver context incrementally for all satisfiability checks. The structure of pushing and popping follows the structure of the `Ite`-rules. From our experience using the solver incrementally may decrease the fusion time by an order of magnitude.
Theorem 3.1. ⊕ version of a run (of multiple steps) of T

Reachability Based Branch Elimination

The main intuition for the proof is that \( S \) implements a flat description of a symbolic forward reachability and backward reachability algorithms adapted to BSTs.

The reachability algorithm reasons about transition rules as a flattened set of \( Base \)-rules with their associated combined branch constraints. Given a rule \( r \in \mathcal{R}(\tau, o, Q, \rho) \) let \( Paths(r) \) be defined as follows:

\[
\begin{align*}
\text{Paths}(\text{Undef}) & \equiv \emptyset \\
\text{Paths}(\text{Base}([\cdot])_i^n, g, q) & \equiv \{(\text{true}, g, q)\} \\
\text{Paths}(\text{Ite}(\varphi, u, v)) & \equiv \bigcup_{(\psi, g, q) \in \text{Paths}(u)} \{(\varphi \land \psi, g, q)\} \\
& \quad \cup \bigcup_{(\psi, g, q) \in \text{Paths}(u)} \{\neg \varphi \land \psi, g, q\}
\end{align*}
\]

Since outputs do not affect reachability they are dropped from the flattened representation. Given a BST \( A \) let there be the following:

\[
\begin{align*}
\text{Moves}^\delta(A) & \equiv \bigcup_{p \in Q_A} \text{Paths}(\varphi, g, q) \\
\text{Moves}^\delta(A) & \equiv \bigcup_{p \in Q_A} \text{Paths}(\varphi, g, q)
\end{align*}
\]

These give a flat representation of all transitions and finalizers (respectively) by source and target control state. We call elements of these sets \textit{moves} and \textit{final moves} respectively.

The \textit{Eliminate} procedure in Figure 8 implements the top-level reachability algorithm. The variable \( w : [\cdot]_A \) is used to represent a list of inputs. To check the reachability of a (final) move it calls \textit{IsReachable} with a \( (\cdot)_A \times \rho_A \)-predicate such that the (final) move is reachable if and only if the source control state can be reached such that the predicate holds (lines 5 and 9). If \textit{IsReachable} returns \textit{false} then the branch is eliminated by simplifying the corresponding \textit{Ite}(\varphi, u, v), where \( u \) (or \( v \)) is the unreachable base rule, into \( v \) (or \( u \)). Note that if \textit{IsReachable} hits the bound \( k \) then it returns \( \perp \) and the branch can not be safely removed.

To minimize calls to \textit{IsReachable}, \textit{Eliminate} uses a more efficient \textit{ComputeUnderApproximation} procedure. It performs a breadth-first forward-reachability analysis from the initial state and tags moves whose path conditions from the initial state are satisfiable as reachable. Breadth-first search increases coverage and ensures that there are potentially several states in a breadth-first frontier for the same control state, hopefully capturing different ways of entering the control state. While more sophisticated under approximations are possible, this approach was adequate for our experiments.

The \textit{IsReachable} procedure in Figure 8 performs a backward breadth-first traversal on \( A \), exploring the states one layer at a time. Each layer is associated with the map \( \Psi \) from control states to reachability conditions yet to be explored. Initially the control state \( q_{\text{tgt}} \) is mapped to the predicate \( \varphi_{\text{tgt}} \). \( \Sigma \) maps control states to the predicates that summarize the arguments for which exploration has already been performed or is about to be performed.

4. Reachability Based Branch Elimination

Fusing already removes many unsatisfiable branches. Still, the resulting BSTs may have a large number of control states and/or rules with redundant conditions. In particular, some branches may be unreachable due to state carried constraints, i.e., even though the branch itself is satisfiable, the conjunction of reachable register values in the source states together with the branch is unsatisfiable. In this section we present a reachability based branch elimination (RBFE) algorithm, that proves the unreachability of and removes such branches in the target BST. The algorithm is a combination of symbolic forward reachability and backward reachability algorithms adapted to BSTs.
ELIMINATE($A$)
1 let $U = \text{ComputeUnderApproximation}(A)$
2 let $M = \text{Moves}^4(A) \cup \text{Moves}^3(A) \setminus U$
3 let $k = |Q_A|
4 foreach move $(p, \varphi, g, q)$ in $M$
5 let $\varphi' = [w \neq \emptyset] \land \varphi \land \varphi(\text{Heap}(w)/x)$
6 if ISREACHABLE($A, p, \varphi', k$) = false
7 eliminate the corresponding branch in $\delta_A$
8 foreach final move $(p, \varphi)$ in $M$
9 let $\varphi' = [w = \emptyset] \land \varphi$
10 if ISREACHABLE($A, p, \varphi', k$) = false
11 eliminate the corresponding branch in $\delta_A$
12 remove control states with no path from $q_0$

ISREACHABLE($A, q_{tgt}, \varphi_{tgt}, k$) : $(ST \times Q_A \times F \times \text{int}) \rightarrow \text{bool}$
1 let layer = \{$q_{tgt}$\}
2 let layer' = \{
3 let $\Psi' = \text{empty} \cup \{q_{tgt} \mapsto \varphi_{tgt}\}$
4 let $\Sigma = \Psi \cup \{q_{tgt} \mapsto \varphi_{tgt}\}$
5 while layer' \neq \{
6 while layer \neq \{
7 pop $q$ from layer
8 let $\psi = \Psi[q]$
9 if $q = q^0_A \land SAT(\psi(r^0_A/x))$
10 return true
11 foreach $(p, \varphi, g, q)$ in $\text{Moves}^4(A)$
12 if $\varphi$ depends on $x$ or $g$ depends on $x$
13 let update = $g(\text{Heap}(w)/x)$
14 let $\gamma = [w \neq \emptyset] \land \varphi(\text{Heap}(w)/x) \land$
15 $\varphi(\text{Tail}(w)/w, \text{update}/x)$
16 else
17 let $\gamma = \psi(g(\cdot,—_A/x)/x)$
18 if SAT($\gamma \land \neg \Sigma[p]$)
19 let $\Sigma[p] = \Sigma[p] \lor \gamma$
20 let $\Psi'[p] = \Psi'[p] \lor \gamma$
21 add $p$ to layer'
22 if $k = 0 \land \text{layer}' \neq \{
23 return \perp
24 let layer = layer'
25 let layer' = \{
26 let $\Psi' = \Psi$
27 let $\Psi' = \text{empty}$
28 return false

Figure 8. Reachability based branch elimination (RBBE).

Let $\Delta_A$ denote the following partial function that extends the transition function $\delta_A$ to input lists and omits the output:

$\Delta_A : (\Gamma \times \text{int}) \times 2^A \rightarrow 2^A$

$\Delta_A([i,w], s) \equiv \Delta_A(w, \pi_2(\delta_A i s))$

A state $s$ is $k$-reachable if there exists $w \in \bigcup_{n \in [0,k]} (\Gamma)^n$ such that $\Delta_A(w, s^0_A) = s$. For example $s^0_A$ is 0-reaching.

A state $s$ is reachable if it is $k$-reachable for some $k \geq 0$. Given $q \in Q_A$ and an $\rho_A$-predicate $\varphi$, we say that $(q, \varphi)$ is $(k)$-reachable if there exists a $(k)$-reachable state $(q, r)$ such that $r \in [\varphi]$

Theorem 4.1. If ISREACHABLE($A, q_{tgt}, \varphi_{tgt}, k$) equals (a) true then $(q_{tgt}, \varphi_{tgt})$ is reachable; (b) false then $(q_{tgt}, \varphi_{tgt})$ is not reachable; (c) $\perp$ then $(q_{tgt}, \varphi_{tgt})$ is not $k$-reachable.

In this algorithm, $\Sigma$ enables a crucial subsumption checking for predicates (line 17) — if a reachability condition $\varphi$ for a control state $p$ is subsumed by $\Sigma[p]$, then any search from $\varphi$ is already covered, so adding $\varphi$ to the next layer would be redundant. A subtlety is to avoid the possible quantifier alternation that would arise if we treat $\Sigma[p]$ as the predicate $\exists w (\Sigma[p])$ (i.e. characterize the reachable set of registers independent of inputs used to reach them). This could potentially introduce undecidability. However, the test in line 17 works because it is sufficient in the the else case (when we omit $\varphi$). When the else case is taken, it means that $\forall w, x (\varphi \Rightarrow \Sigma[p])$ holds, which implies that $\forall r (\exists \varphi \Rightarrow \exists w \Sigma[p])$ holds. The latter condition is the necessary condition needed to preserve all register values.

5. Specifying Effectful Comprehensions

We have explored several frontends for specifying effectful comprehensions. In Section 5.1 we present a frontend from imperative C# code to BSTs. This pattern matches interfaces present in existing streaming frameworks (Section 7).

Some comprehensions can be more efficiently specified with a specialized frontend. In Section 5.2 we translate regexes with named captures into BSTs, while Section 5.3 presents a similar approach for XPath queries.

5.1 Effectful Comprehensions as C#

We have implemented a translation from a subset of C# to BSTs. Users extend an abstract class Transducer<Char, Int> and override methods named Update and Finish to define $\delta$ and $\delta$, respectively. Users may opt to not override Finish, in which case a trivial no-op finalizer is used.

Example 5.1. The following code implements the ToInt transducer from Figure 4(b):

```csharp
partial class ToInt : Transducer<Char, Int> {
    int i = 0; bool defined = false;
    override IEnumerable<Int> Update(Char d) {
        if (0x30 <= d && d <= 0x39) {
            i = (10 * i) + (d - 0x30);
        } else throw new Exception();
        defined = true;
        yield break;
    }
    override IEnumerable<Int> Finish() {
        if (!defined) throw new Exception();
        yield return i;
    }
}
```
We use regular expressions with captures to enable scenarios where each type is split into \( \rho \) enum some output of type \( o \), which captures the state update and outputs represented.

5.2 Effectful Regex Comprehensions

We use regular expressions with captures to enable scenarios that require custom pattern matching. A typical example is to extract some information stored in a text file using a custom parser. Consider a regex pattern \( P \) of the form

\[
(S_1 (\langle cap_1 \rangle P_1) S_2 \cdots S_n (\langle cap_n \rangle P_n) S_{n+1})^*
\]

where \( S_i \) and \( P_i \) are regular expressions such that no \( P_i \) accepts the empty string and there is no ambiguity about where each \( S_i \) ends or where each \( P_i \) starts. In particular, if one pattern accepts a string ending with some character then the following pattern must reject any string starting with the same character.

The intent is that each \( S_i \) is a skip pattern and each \( P_i \) is a parse pattern. The capture names \( cap_i \) are mapped to transducers \( A_i \) that map strings matching pattern \( P_i \) to some output of type \( o_i \). We developed an algorithm that given \( P \) and the transducers \( \{ cap_i \mapsto A_i \}_{i=1}^n \) constructs a fused transducer that parses strings matching \( P \) into \( n \)-tuples \((o_1, \ldots, o_n)\). The algorithm works as follows:

1. Parse and translate the regex into an SFA.
2. Keep track of which parts \( B_i \) of the SFA accept the patterns \( P_i \). The input values accepted inside \( B_i \) represents a match of the capture group (with no ambiguity due to our assumptions).
3. Fuse each \( B_i \) with the appropriate \( A_i \). The start and end of a capture group match respectively trigger initialization and finalization of the BST.

The fusion performed in step 3 differs from that in Section 3 in that the BSTs are composed in a hierarchical manner, i.e., instead of all output being directed through another BST, a part of the transduction is delegated to another BST. This model allows subsequences of an effectful comprehension to be specified modularly.

Example 5.2. The following regex illustrates a case that parses each line of a csv file in such a way that the substring in the third column (between the second and third commas) is parsed as a non-negative integer in decimal notation and the substring in the fourth column is parsed as a Boolean:

\[
((,"\),\{2\}(\langle int\rangle\d+),\langle bool\rangle\w+,\{\[n\]*\n)*
\]

Here \( S_1 \) is "," (skip to the third column), \( S_2 \) is "," (skip to the next column), and \( S_3 \) is ",[\[n\]*\n" (skip remaining columns until EOL). The capture int is mapped to the transducer ToInt from Figure 4(b) and the capture bool is mapped to a transducer ToBool, which maps the strings "true" and "false" respectively to true and false.

5.3 Effectful XPath Comprehensions

For extracting information from XML formatted data we use transducers constructed from XPath query expressions. Consider an expression \( X \) of the form

\[
\text{st:trans}(/\text{tag}_1/\text{tag}_2/\text{tag}_3/\cdots/\text{tag}_n)
\]

The tag names \( \text{tag}_n \) specify a path to match in an XML file. \( \text{trans} \) is a name that maps to a transducer \( A \) that maps the contents of any matching elements to output of type \( o \). Given \( X \) and the transducer \( A \), a fused transducer that parses matches of \( X \) into values of \( o \) is constructed. The match for the query uses counting with an integer register to ignore arbitrarily deep nestings of non-matching elements. Otherwise the algorithm is similar to the one for regular expressions in Section 5.2 (i.e. for steps 2 and 3).

Example 5.3. Consider the following XML:

\[
<\text{cities}>
<\text{city name='Roslyn'>}
<\text{timezone}=\text{PST}</\text{timezone}>
<\text{population}=893</\text{population}>
</\text{city}>
<\text{city name='Santa Barbara'>}
<\text{population}=88410</\text{population}>
</\text{city}>
</\text{cities}>
\]
We have implemented the techniques described above in BSTs, fuses them and finally generates efficient C# code. A tool that translates C# (and our other frontends) into parse the populations in the dataset as non-negative integers in samples to obtain a confidence interval smaller than throughputs are means of a sufficient number of MB/s at a 95% confidence level. All pipelines were run to the code block of the target control state. For each control state a labeled code block that implements the transition rule is generated. Given a rule, a tree of if else statements is generated, where each leaf consist of an appropriate sequence of outputs, state updates and finally a goto to the code block of the target control state.

We evaluate the viability of our approach with a set of benchmark pipelines. The experiments were run on an Intel Core i5-3570K CPU @ 3.4 GHz with 8 GB of RAM. All reported throughputs are means of a sufficient number of samples to obtain a confidence interval smaller than ±0.5 MB/s at a 95% confidence level. All pipelines were run through C#’s NGen tool, which produces native code for C# assemblies ahead-of-time.

Figure 9 presents throughputs for four variations of each pipeline. For LINQ the pipeline stages produce output with yield return and read input from an IEnumerable<T> of the previous stage. For Method call the pipeline stages receive input through method parameters and produce output by directly calling the next stage. The Hand-written pipelines are straightforward implementations using arrays as buffers between phases. The fused and optimized pipelines are labeled Fused. The pipeline stages in the LINQ and Fused pipelines use code generated from BSTs by our implementation, while the Hand-written pipelines use Hand-written C# and .NET system libraries where available. For the Hand-written pipelines we did not perform any manual fusion, since the aim of this paper is to allow pipeline stages to be specified modularly with the fusion being handled by the compiler. The Method call pipelines use a variation of our code generator, which stores the control state as an int and uses a switch to execute the appropriate rule.

The first four pipelines implement various computations: Base64-avg calculates a running average (window of 10) for Base64 encoded ints and re-encodes the results in Base64; CSV-max decodes an UTF-8 encoded CSV file to UTF-16, extracts the third column with a regular expression, finds the maximum length of these strings and outputs it as a single UTF-8 encoded decimal formatted integer; Base64-delta reads Base64 encoded ints and outputs deltas of successive inputs as UTF-8 encoded decimal integers on separate lines; and UTF8-lines decodes an UTF-8 encoded file to UTF-16, counts the number of newline characters and outputs the count as a single UTF-8 encoded decimal formatted number.

For these pipelines we measured the throughput with 100 MB of data. For the UTF8-lines pipeline we used Herman Melville’s “Moby Dick” repeated a sufficient number of times, while for the others we used randomly generated data. For all pipelines except CSV-max the LINQ version has the highest throughput. We believe this is due to the overhead associated with passing values through IEnumerable<T>.

The rest of the pipelines in Figure 9 present a more detailed comparison of CSV parsing scenarios. Pipelines for three different datasets are compared: CHSI benchmarks use a dataset on health indicators from the U.S. Department of Health & Human Services, for which the three pipelines produce the average lung cancer deaths, minimum births and maximum total deaths for counties in the dataset; SBO benchmarks use a dataset on business owners from the U.S. Census Bureau, for which the three pipelines find the maximum employees, minimum gross receipts and average payroll for businesses in the dataset; and CC uses a dataset of consumer complaints received by the U.S. Consumer Financial Protection Bureau, for which the pipeline produces the maximum value for the ID column.

Each of these pipelines apply four effectful comprehensions: (i) decode UTF-8 to UTF-16, (ii) parse a column as an int using a regular expression based parser, (iii) run a query (maximum, minimum or average), and (iv) output the result.
as a sequence of bytes. The pipelines differ only in the regular expression and query used.

Each version of the pipelines uses the same regular expression for parsing the CSV file. For example, the expression 

```
([".*"])*\{\$\{value\}\d+\},[^\n]+\n*\)
```

is used in the maximum employees pipeline for matching the sixth column on each line. In the Hand-written tests the .NET framework’s RegexOptions.Compiled option was used, which generates a .NET assembly for doing the matching. This extra work is not counted against the reported throughputs. Another optimization we implemented for the Hand-written pipelines is that the regular expression is matched for the whole dataset and the values captured are then iterated. This proved to be significantly faster than splitting the dataset into lines and running the regular expression on each line separately.

The original SBO dataset is 744 MB, which caused the .NET regular expression library to run out of memory. To work around this we cut the dataset down to a 83 MB prefix. All of the Fused pipelines in Figure 10 use an XPath based transducer for extracting the relevant data. The XmlDocument pipelines use the the XPath matching implemented in C#’s standard libraries. The throughput for the XmlDocument version of the PIR-proteins pipeline is not reported because the library runs out of memory with the 700 MB dataset. The XpathReader pipelines use Microsoft’s XpathReader library, which allows evaluating a subset of XPath in a streaming manner and is able to process the PIR-proteins dataset.

The Fused versions have the highest throughput on all of the XPath benchmarks, with an average speedup of 9× over the streaming XpathReader library. The fact that in the Fused pipelines the XPath matching code is specialized to the query is likely to give it a significant advantage over the XmlDocument and XPathReader versions, which do not perform any code generation. This also holds for the Method call pipelines, which were second on all XML benchmarks. For queries over large XML datasets using our approach over a general purpose XPath library makes sense, because the speedup will make up for the compilation time.

Figure 11 presents the number of branches in rules removed by RBBE (Section 4) for each pipeline. The numbers are sums of removals after all fusions that contribute to the complete pipeline. We can see that for most pipelines applying RBBE resulted in branches being removed. Thus RBBE is helpful for allowing bigger pipelines to be practically fused.

The figure also presents the total running time for the transformation from user code to fused C#. We can see that none of the running times are excessive for retail builds. The LINQ versions of the pipelines can be used with equivalent semantics and negligible compilation overhead.

### 6.1 Comparison with Hand-Coded Fusion

This section provides a performance comparison of fused code produced by our tool against a real-world hand-fused version providing the same functionality, the concrete example we look at being HTML encoding. In modern implementations of HTML encoders (as well as other anti-XSS encoders), for robustness reasons, the input string being en-
encoded is also repaired by replacing any misplaced surrogates by the Unicode replacement character \( \phi \) or 0xFFFF (65533 in decimal). For comparison we use here the built-in hand-coded anti-XSS encoder AntiXssEncoder.HtmlEncode from .NET 4.5, which is equivalent to the fused symbolic transducer Rep \( \otimes \) HtmlEncode, with Rep and HtmlEncode illustrated in Figure 12.\(^7\) Rep replaces any misplaced surrogates by \( \phi \) and HtmlEncode assumes that the input is valid Unicode and encodes non-HTML safe characters using the appropriate escape sequences. The input and output types of both Rep and HtmlEncode is char.

The predicate \( \varphi_h(x) \) is the high surrogate predicate defined by the character range [0xD800 – 0xDBFF] and \( \varphi_l(x) \) is the low surrogate predicate defined by the character range [0xDC00 – 0xDFFF]. In any correctly formatted UTF16 encoded string, surrogates may only occur in pairs, where a high surrogate is immediately followed by a low surrogate. The predicate \( \varphi(x) \) is defined as \( \varphi_h(x) \lor \varphi_l(x) \) and corresponds to the character range [0xD800 – 0xDFFF]. The predicate \( \varphi_{safe} \) is defined as the set:

\[
\{\text{0x20, 0x21, 0x3D}\} \cup \{\text{0x23 – 0x25}\} \cup \{\text{0x28 – 0x3B}\} \cup \{\text{0x3F – 0x7E}\} \cup \{\text{0xA1 – 0xAC}\} \cup \{\text{0xAE – 0x36F}\}
\]

that are considered to be safe or whitelisted and are not encoded. Observe that \( \phi \) is not whitelisted here. Encode\( (c) \) is a pattern for a rule defined as:

\[
\text{Encode}(c) = \\
\text{It}c = 0\text{x22. }\text{Base}("\text{quot;}", q_0, 0), \\
\text{It}c = 0\text{x26. }\text{Base}("\&","q_0, 0), \\
\text{It}c = 0\text{x3C. }\text{Base}("\&lt;","q_0, 0), \\
\text{It}c = 0\text{x3E. }\text{Base}("\&gt;","q_0, 0), \\
\text{It}c < 10. \text{Base}("\#" + \text{Digits}(c, 1) + ";", q_0, 0), \\
\text{It}c < 100. \text{Base}("\#" + \text{Digits}(c, 2) + ";", q_0, 0), \\
\text{It}c < 1000. \text{Base}("\#" + \text{Digits}(c, 3) + ";", q_0, 0), \\
\text{It}c < 10000. \text{Base}("\#" + \text{Digits}(c, 4) + ";", q_0, 0), \\
\text{It}c < 1000000. \text{Base}("\#" + \text{Digits}(c, 5) + ";", q_0, 0), \\
\text{It}c < 10000000. \text{Base}("\#" + \text{Digits}(c, 6) + ";", q_0, 0), \\
\text{Base}("\#1" + \text{Digits}(c, 6) + ";", q_0, 0))\}
\]

\(^7\)Equivalence holds provided that AntiXssEncoder.HtmlEncode is called with the second parameter being false to specify the decimal encoding style. The AntiXssEncoder class is implemented in the System.Web.Security.AntiXss namespace.

![Figure 12. Symbolic transducers for HTML encoding](image)

Figure 12. Symbolic transducers for HTML encoding

where \( \text{Digits}(c, n) \) is shorthand for a list of expressions that give the \( n \) least significant decimal digits as characters. For example \( \text{Digits}(c, 2) = \lfloor (c/10) \mod 10 \rfloor + 0x30, (c \mod 10) + 0x30 \). HtmlEncode instantiates two versions of the pattern as \( \text{Encode}(x) \) and \( \text{Encode}(CP(r, x)) \), where the function \( CP \) computes a Unicode code point from a high and a low surrogate and is defined as \( \text{CP}(h, l) = (((h \& 0x3FF) + 0x40) \& 0x10) || (l \& 0x3FF) \). Note that while both instantiations of \( \text{Encode} \) include some unreachable branches these are removed either by pruning in the fusion or during RBBE. For example, in \( \text{Encode}(CP(r, x)) \) RBBE eliminates the first eight true branches using the state carried constraint that \( CP(r, x) \geq 0x0D0000 \).

In Figure 13 we compare the throughputs of Rep \( \otimes \) HtmlEncode with the transducers implemented in C# and fused with our tool, and AntiXssEncoder.HtmlEncode on three datasets: Random is uniformly random characters, English is Herman Melville’s “Moby Dick”, Chinese is Guanzhong Luo’s “Romance of the Three Kingdoms”. The throughputs are reported for the size of the input in UTF16, where each character takes two bytes.

The throughput of the fused code generated for Rep \( \otimes \) HtmlEncode by our tool is comparable to (and sometimes greater than) that of AntiXssEncoder.HtmlEncode. This allows Rep and HtmlEncode to be implemented modularly without losing in performance to hand-fused code.

7. Related Work

Symbolic transducers were introduced in flat form in [43] for analysis of string sanitizers with the main focus on symbolic finite transducers or SFTs. The core difference between BSTs and STs in [43] is branching structure in rules. This causes the fusion algorithm here to be fundamentally different and much more intricate from composition with flat rules that [43, Proposition 1] refers to but does not define. Another difference to [43] is explicit handling of registers. This difference is fundamental, because SFTs are composed for analysis such as commutativity and idempotence, which become undecidable when registers are allowed.

Composition of SFTs in [43] is agnostic regarding determinism, i.e., whether guards overlap. Rather, what matters is single-valuedness for decidability of equivalence. Initially, we tried to use flat rules but this attempt failed. Branches of if-then-else programs, when represented as separate Z3 formulas, get, after simplification, internalized representations whose semantics is very difficult to recover and often depend
on context conditions. For example, bit-vector expressions often end up using 2-complement arithmetic applied to 2-bit or 3-bit bit-vectors in subexpressions, combined with 0-padding to adjust bit-vector widths. Rediscovering the original intent (branching structure and appropriate arithmetic operations) becomes hopelessly error-prone, resulting in very inefficient code, even when feasible. Code generation after composition was not addressed in [43].

As an orthogonal approach to fusion, method-call composition \( B(A(x)) \), called lazy composition in [43], is deemed the more efficient way to handle semi-decision problems for symbolic transducers with registers, by using well-founded recursive axioms over algebraic datatypes, \( Th(A) \cup Th(B) \), asserted as a sub-theory to Z3.

Prior work on STs has focused on register exploration and input grouping that are orthogonal problems [17, 44]. Register exploration attempts to project the register type \( \rho \) into a Cartesian product type \( \rho_1 \times \rho_2 \) where \( \rho_1 \) is a finite type, the primary goal is to reduce register dependency by migrating \( \rho_1 \) into the set of control states. Input grouping tries to take advantage of grouping characters into larger tokens in order to avoid intermediate register usage, that has applications in decoder analysis [17] and parallelization [44].

**Streaming string transducers** or SSTs [14] correspond to 2-way deterministic finite-state transducers [21] that can be specified through regular combinators [12] using an associated programming language called DReX [13]. Kleenex [24] is an elegant programming language that uses nondeterministic finite state transducers [10] with embedded action semantics for side effects. Kleenex programs are greedily disambiguated to resolve nondeterminism and compiled into SSTs.

SSTs with data values or symbolic alphabets are unfortunately not closed under functional composition [11, Proposition 4] and cannot therefore be fused in general.

The stream processing area has a large body of work [20, 29, 31, 33, 41]. Stream computations with internal state have been studied before. The work in [16] defines a Stream data-type with internal state that yields elements and allows operations such as map, fold, and zip. These operations are functional and operate on one element at a time with no operation-state carried across elements. The state in the Stream allows one to represent the current position, and bundling in the case of generalized stream fusion [28], in the stream. In contrast, our focus is on applying transformations that have operation-state carried across elements (as opposed to streams having state). This allows us to represent effectful functions such as UTF decoding/encoding.

Some libraries for streams provide APIs for expressing stateful operations. The Apache Flink [3] and Spark Streaming [7] distributed streaming engines both provide support for using state in stream operations and an associated framework for implementing fault tolerance in the presence of state. The Highland.js [5] and Conduit [1] are traditional stream libraries, which both provide a way to express stateful operations. However, in these libraries the stateful operations are treated as black boxes, as opposed to our approach that fuses operations in compositions of BSTs. Implementing frontends similar to the C# one (Section 5.1) for these libraries would allow code written for them to use our backend.

**Fusion** is one of many optimization techniques that are used in a variety of streaming applications, as discussed in the survey on stream processing optimizations [25]. Fusion is typically implemented through method call composition. Our experiments indicate that, for BSTs, fusion provides on average \( 2.6 \times \) speedup over fusion by method call composition alone.

StreamIt [42] is a language and compiler that provides a high-level stream abstraction view designed for signal processing applications. The two primary transformations of the compiler are fission and fusion of filters. Fission is used for splitting filters (and streams) to expose parallelism. Fusion is used for merging filters (and streams) for load balancing and synchronization removal. Typically, fusion means pipeline fusion, where two or more filters connected in a pipeline are fused into a single filter. The paper [8] studies fusion with a linear state space representation, i.e., where the outputs and the next state values are computed as linear combinations of the inputs and the previous states. The composition retains the linear state space representation with a linear increase in size. In contrast, we can compose any filters with operation-state where the state update is over any decidable (quantifier-free) theory. To this end we use state-of-the-art SMT technology in our compiler. The work we propose here is complimentary to current techniques used in StreamIt: the composition and optimization techniques for BSTs could be used as an additional backend module in the StreamIt compiler for filters with operation-state which are not amenable to a linear state space representation. Other related work on StreamIt discusses fusion followed by optimizations like constant propagation and scalar replacement [23], and loop unrolling [39].

Fusion trades communication cost against pipeline parallelism [25]. Sometimes fission can be applied to expose parallelism, as studied for example in [38]. It is an open question as to what fission would mean in the context of BSTs and if it would be beneficial. Keep in mind that fusion of BSTs is achieved through complex symbolic algebraic manipulation of expressions, the inverse of which may not be possible or the search space may be astronomical.

Fusion of effectful comprehensions is also related to classical work on filter fusion [36] and deforestation [45]. Fusion of symbolic transducers can be viewed as an extended form of filter fusion that incorporates loop carried state and SMT based constraint satisfaction techniques.

The Steno library in [34] implements deforestation for LINQ queries and achieves speedups from removing the \( T\text{Enumerable} \) abstraction similar to what we report in Section 6. In contrast with our work, Steno treats filters as black
boxes, although the deforestation can expose some optimization opportunities to the compiler. Additionally, some of Steno’s optimizations assume that filters are stateless.

Filter fusion has also been extended to network fusion [22] that uses the product of labeled transition systems, to merge a network of interconnecting components. Synchronous product of automata and fusion of symbolic transducers have different semantics and properties.

The work in [40] is related to our work regarding motivation. The difference is that we use an automata based definition of transducers with an explicit control flow graph and use an SMT solver as an oracle in our algorithms. This leads to a different set of algorithms and opens up a different set of optimization techniques.

LINQ [30] uses the list monad (or list comprehension [46]) as its primary construct for query processing and (unlike SQL) also supports nested lists. The list comprehension construct is in LINQ expressed with the Select or, more generally, SelectMany extension method of the IEnumerable<

...
References

[1] Conduit (Haskell library).  


https://flink.apache.org/.


http://highlandjs.org/.


http://spark.apache.org/streaming/.


