Fusing Effectful Comprehensions

Olli Saarikivi
Aalto University
olli.saarikivi@aalto.fi

Margus Veanes Todd Mytkowicz
Madan Musuvathi
Microsoft Research
{margus,toddm,madann}@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×, on average) and a LINQ implementation of the pipeline (5×, 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.

Finally, we formalize the semantics of symbolic transducers and their compositions as a transduction monad, which provides a link between the automata-theoretic view and a monadic view of symbolic transducers.

Categories and Subject Descriptors F.1.1 [Computation by Abstract Devices]; Models of Computation—Automata; D.3.4 [Programming Languages]; Processors—Code generation; F.3.1 [Logics and Meanings of Programs]; Specifying and Verifying and Reasoning about Programs—Mechanical verification; D.3.2 [Programming Languages]; Language Classifications—Specialized application languages

General Terms Algorithms, Languages, Performance, Verification

Keywords transducers, comprehensions, fusion, deforestation, reachability analysis, applicative functors, monads

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 explicit iteration constructs, while the runtime ameliorates the cost of the abstraction by performing various optimization such as fusion/deforestation [27, 35].

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 is then deserialized into objects in a higher-level language, such as Java. In this example, the application selects stock prices from each log, then deserialized into objects in a higher-level language, such as Java. 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, such as a mapper or a reducer, of data-processing systems [2, 7, 13, 38], 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 [33] 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 the text, assuming a decimal encoding with ASCII digits, i.e., matching the regex \[0-9]+\$. Suppose our sample pipeline is as follows: it first UTF8 decodes (Utf8Decode) and then parses an integer (ToInt). Utf8Decode takes as input a sequence of bytes and produces a sequence of integers that are the decoded Unicode character codes. For simplicity assume that only up to 2 byte encodings are allowed.\(^1\) Utf8Decode can be illustrated graphically as follows:\(^2\)

\[
\begin{align*}
\text{c} \in [0-0x7F] &/ [\text{c}] ; r := 0 \\
\text{r} := 0 &\quad \text{q} \quad \downarrow & \text{true} / [1] \\
\text{c} \in [0x80-0x8F] &/ [\text{r} | (\text{c} \& 0x3F)] ; r := 0 \\
\text{c} \in [0xC2-0xDF] &/ [\text{r} | (\text{c} \& 0x3F)] ; r := 0 \\
\text{r} := ((\text{c} \& 0x3F) \ll 6) &; (\text{r}, \text{i}) := ((\text{c} \& 0x3F), \text{i}) \\
\text{c} \in [0x80-0x8F] &/ [\text{r} | (\text{c} \& 0x3F)] ; r := 0 \\
\text{r} := 0 &\quad \text{q} \quad \downarrow & \text{true} / [1] \\
\end{align*}
\]

**Figure 2.** Utf8Decode as a flat symbolic transducer.

The following paragraphs serve also as an informal introduction to symbolic transducers. Utf8Decode uses two control states \(q_0\) and \(q_1\), where \(q_0\) is both the initial and the final state. A transition \(p \in \Delta(q_0; r; x) \to q\) has the following meaning: if the current state is \(p\) and the current byte \(c\) is in the range \(\alpha\) then enter state \(q\), yield the elements in the sequence \(s\) and update the register \(r\) to the value \(g\). Initially \(r\) has the value 0. For example, if the input sequence of bytes is \([0x61, 0xC5, 0x93]\) then the output sequence of character codes is \([0x61, (0x0C5803F) \ll 6] | (0x93803F)\] that is equal to \([0x61, 0x153]\) or the string "aa".

ToInt can be illustrated as follows:

\[
\begin{align*}
d &\in [0x30-0x39] / [\text{i}] ; i := (d - 0x30) \\
\text{i} := 0 &\quad \text{p} \quad \downarrow & \text{true} / [1] \\
d &\in [0x30-0x39] / [\text{i}] ; i := (10 \times i) + d - 0x30 \\
\end{align*}
\]

**Figure 3.** ToInt as a flat symbolic transducer.

In addition to normal transitions, ToInt also uses a finalizer (drawn as a dashed arrow), that upon reaching the end of the input outputs the value of its register in the singleton sequence \([1]\). In a finalizer, the elements in the output sequence may only depend on the register value and there is no register update.

Symbolic transducers can be fused into a single symbolic transducer that preserves the semantics of function composition. Consider the fusion of Utf8Decode with ToInt, which ends up being identical to ToInt due to ToInt only accepting ASCII digits which in turn has become a dashed arrow), that upon reaching the end of the input outputs the value of its register in the singleton sequence \([1]\). In a finalizer, the elements in the output sequence may only depend on the register value and there is no register update.

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 \([12]\) 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 \([10, 19]\) or in composition of symbolic finite state transducers \([33]\).

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 compre-

---

1. Up to two byte UTF8 encodings cover the full range of characters in extended ASCII. In general there are up to four byte encodings to cover all Unicode characters.
2. The operation ‘&’ denotes bitwise-and, the operation ‘\(^\text{\text{'}}\)’ denotes bitwise-or, and the operation ‘\(^{<<}k\)’ denotes shift-left by \(k\) bits.
3. We ignore C#'s limitation that \(\text{yield}\) is not allowed in lambda functions.
hensions. Figure 4 presents a function implementing a pipeline of the Utf8Decode and ToInt comprehensions using C#’s LINQ [22] library\(^2\). 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 8). 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 (Base64) encoded data. On average, our fused code is 3× faster than reasonable hand-written code and 5× faster than a LINQ implementation. 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.

Finally, we formalize the semantics of transducers and their composition using applicative functors and monads in a purely functional style. A transduction monad extends state monads with composition mechanisms allowing us to compose transducers. We found this connection to the functional programming world important because it explains the problem from a very different angle and lets us formalize composition unambiguously and succinctly. The functional view also provides a way to explain composition in a more declarative style, as opposed to automata based formulations that are mostly operational.

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 monadic formalization of transducers and their compositions.
- A comprehensive evaluation demonstrating the efficacy of our approach.

2. Symbolic Transducers

This section formally introduces symbolic transducers or STs, as a generalization of symbolic finite transducers or SFTs. The definition used here differs in the following key aspect from the original introduction of STs [33] — it is specialized for deterministic STs. This 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:

- it makes determinism an integral part of the definition of an ST, rather than a property of an ST;
- 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 sub-formulas would have to be selected from a potentially large search space. Moreover, the choices may be data-dependent, and ultimately depend on 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. Instead of two flat transitions from state \(q_0\) (one looping back to state \(q_0\) and one transitioning to state \(q_1\)) the ST has a single rule from each state, as illustrated in Figure 5, where \(\bot\) corresponds to an implicit rejecting state that would be added to Figure 2 after completion.\(^5\)

In Utf8Decode the order of the two input byte conditions from state \(q_0\) is important when considering inputs that mostly consists of ASCII characters in which case the second condition is rarely evaluated. The initial register value is 0. The basic rules are the leaf transitions of the branches and are labeled \(s; r := g\) where \(s\) is the output sequence and \(g\) the updated register value.

Figure 6 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

\(^2\) The code for other list comprehension libraries, such as Java 8’s Streams API, is largely similar.

\(^5\) Completeness of a flat ST means that the disjunction of all the guards of transitions from any given state is equivalent to \(true\).
We indicate the component of an ST by using the ST as a subscript,
\( \tau \) first.

In general our theory and algorithms work with any decidable background theory. A term in \( T(\tau \to \text{bool}) \) 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 or value of type \( \tau \). We assume that if \( \tau \) is a Cartesian product type \( \tau_1 \times \tau_2 \) then there are projection functions first : \( \tau \to \tau_1 \) and second : \( \tau \to \tau_2 \) and a pairing function \( \langle \cdot \rangle : \tau \to \tau_1 \times \tau_2 \) with the intended semantics that \( \langle \, ([t_1], [t_2]) \rangle = ([t_1], [t_2]) \), and \( [\text{first}(t, t_2)] = [t_1] \), and \( [\text{second}(t, t_2)] = [t_2] \). Each type \( \tau \) denotes a nonempty set and has a default element \( \text{default}_\tau \).

The formal definition of a rule is as follows. Given types \( \alpha, \rho \) and a finite set (or type) \( Q \), let \( R(\alpha, \sigma, Q, \rho) \) denote the smallest set of rules satisfying the following conditions:

- **Undef** \( \in X \);
- for \( n \geq 0 \) if \( f_i \in R(\alpha, \sigma, Q, \rho) \) and \( q \in Q \) then \( \text{Base}(f_i, q, g) \in X \);
- if \( \phi \in R(\alpha, \sigma, Q, \rho) \) and \( t, f \in X \) then \( \text{Ite}(\phi, t, f) \in X \).

A rule \( r \in R(\alpha, \sigma, Q, \rho) \) denotes a partial function \( [r] \) of type \( \alpha \times Q \times \rho \)\
\[
[\text{Undef}] v = \bot \quad \text{for all values } v;
\]
\[
[\text{Base}(f_i, q, g)] v = ([f_i] v, q, [g] v);
\]
\[
[\text{Ite}(\phi, t, f)] v = \begin{cases} [t] v, & \text{if } \phi v = \text{true}; \\ [f] v, & \text{otherwise}. \end{cases}
\]

A symbolic transducer (ST) is a tuple \( (\sigma, \rho, Q, Q, r^0, \delta, \$) \) where the components are:

- **input type** \( \sigma \);
- **output type** \( \rho \);
- **register type** \( \rho \);
- **finite control state set** \( Q \);
- **initial state** \( (q^0, r^0) \) of state type \( \sigma^0 = Q \times \rho \), and \( s^0 = (q^0, r^0) \);
- **transduction function** \( \delta : Q \to R(\alpha, \rho, Q, Q, \rho) \);
- **finalizer** \( \$ : Q : R(\alpha, \rho, Q, Q, \rho) \).

We indicate the component of an ST by using the ST as a subscript, unless the ST is clear from the context.

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 \( \tau \). Instead

\[ \text{of lifting every input type } \tau \text{ to a sum type of } \tau \text{ and an end-of-input symbol, end-of-input is handled separately by the finalizer.} \]

We adopt the following variable naming conventions of terms occurring in rules. In a term \( t \) occurring in a rule, variable \( x \) is of type \( \tau \) and refers to the input element and variable \( y \) 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 \((x = \varphi)\) the \( x \) is a literal part of the formula, while \( \varphi \) refers to another formula. Substitution of a variable \( y \) by a term \( t \) is denoted \( t(y \mapsto u) \).

In \texttt{UTF8Decode}, in Figure 5, 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:
\[
\texttt{UTF8Decode} = \{ q_0 \mapsto \text{Base }[[], q_0, 0], q_1 \mapsto \text{Undef} \}
\]

The finalizer of \texttt{Tohn}, in Figure 6, that is shown as the dashed arrows, is the function:
\[
\texttt{Tohn} = \{ p_0 \mapsto \text{Undef}, p_1 \mapsto \text{Base }[[], p_1, 0] \}
\]

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.

An ST \( A \) denotes a transduction \( \langle A \rangle \) that is a partial function of type \( [\alpha] \to [\sigma] \). First, we define the following partial semantic functions \( \delta : \sigma \to [\alpha] \times [\sigma] \) and \( \$ : [\sigma] \to [\alpha] \times [\sigma] \) that enable us to provide a declarative definition of \( \langle A \rangle \):
\[
\hat{\delta} a (q, b) \overset{\Delta}{=} \hat{\delta} a (q, b); \quad \hat{\$} (q, b) \overset{\Delta}{=} \hat{\$} q \).
\]

Let \( a \overset{\Delta}{=} \{a_i\}_{i=1}^k \) be a given input list. Let \( \pi_1 (a, b) \overset{\Delta}{=} a \). Then
\[
\langle A \rangle a \overset{\Delta}{=} \pi_1 ((\hat{\delta} a_1) \uplus (\hat{\delta} a_2) \uplus \cdots \uplus (\hat{\delta} a_k) \uplus \hat{\$} s_0) \quad (1)
\]

that \( \uplus \) is a left-associative operator of type \( (\sigma \to [\alpha] \times [\sigma]) \times (\sigma \to [\alpha] \times [\sigma]) \to (\sigma \to [\alpha] \times [\sigma]) \)

that composes single-input transduction steps into multi-input transduction steps, with the formal definition:
\[
F_1 \uplus F_2 \overset{\Delta}{=} \lambda s \left( \begin{array}{l}
\text{let } (u_1, s_1) = (F_1 s) \text{ in } \\
\text{let } (u_2, s_2) = (F_2 s_1) \text{ in } \\
(u_1 + u_2, s_2)
\end{array} \right)
\]

where \( \cdot \) is list concatenation. For example, given \( a \overset{\Delta}\longleftrightarrow \{0 \cdot 0 \cdot 5, 0 \cdot 9 \cdot 3\} \) and \( A = \text{UTF8Decode} \), we have that, \( s_0 = (q_0, 0) \),
\[
\langle A \rangle a \overset{\Delta}{=} \pi_1 ((\hat{\delta} 0 \cdot 0 \cdot 5) \uplus (\hat{\delta} 0 \cdot 9 \cdot 3) \uplus \hat{\$} s_0) = \pi_1 ((\hat{\$} \langle \langle \rangle \rangle) \uplus \langle [a], 0 \rangle) = \pi_1 ((\hat{\$} \langle \langle \rangle \rangle) \uplus \langle [a], 0 \rangle) = \langle [a] \rangle
\]

We refer to \( \uplus \) as step composition and revisit it in Section 7. The main intuition about \( \uplus \) is that it combines function composition with list comprehension in the following sense. If the arguments \( F_1 \) and \( F_2 \) do not depend on the state \( s \) then \( \uplus \) corresponds to concatenation, as in a typical \texttt{SelectMany} list comprehension in LINQ. If, on the other hand, \( F_1 \) and \( F_2 \) produce no outputs and only transform the state, then \( \uplus \) corresponds to function composition.

3. Fusion of STs

Consider two STs \( A \) and \( B \) such that \( o_A = i_B \). We want to fuse \( A \) and \( B \) into a single ST \( A \oplus B \) such that \( [A \oplus B] \) is equivalent to \( [A] \circ [B] \), i.e., \( \text{lex}([B]) ([A]) ([x]) \). We first explain the main idea behind the construction. We then explain the incremental algorithm that makes the composition scale in practice.

The control-state

6 As usual, \( \to \) is right-associative. We assume that \( \times \) is also right-associative and has higher precedence than \( \to \).

7 We can lift the rule type \( \tau \to [\alpha] \times \rho \) to be the type of a total function \( \tau \to ([\alpha] \times \rho) \) by using option types, but here we work directly with partial functions \( f \) of type \( \tau_1 \to \tau_2 \) as relations of type \( \tau_1 \times \tau_2 \) with the understanding that if \( (a, b), (a', b') \in [f] \) then \( b = b' \). Moreover, \([f](a) = b \) if \( (a, b) \in [f] \) and \([f](a) \overset{\Delta}{=} \emptyset \) if \( b(a, b) \in [f] \) = \emptyset.
complexity of the algorithm is $|Q|^2$. Typically $|Q|$ is in the range of 100-1000. The worst-case complexity with respect to the size of the rules is also quadratical, 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: $\ell = t_A$, $o = o_B$, $p = p_A \times p_B$, $q \subseteq Q_A \times Q_B$, $\ell' = (r_A', \delta_A')$, $q' = (q_A', q_B')$.

The goal of the fusion algorithm is to construct $\delta_A \otimes B$ and $\delta_A B$.

For each pair $(p, q)$ of control states in $Q_A \times Q_B$, build a fused rule that, given the rule $\delta_A p$, symbolically runs $\delta_B q$ treating all of the output lists $[v_i]_{i=1}^n$ that occur in the Base-subrules of $\delta_A p$ as symbolic variables. 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 $R(x \times p, o, Q, \rho)$. 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-label explosion. Just consider self-composition of an encoder (say, with a single control state) that may output $n$ elements for some input element. Then the composition may potentially output $n^2$ elements for some input element, but most of those cases may be infeasible due to symbolic constraints imposed by the output functions and their guards in $A$ when considered as inputs of $B$. For example, an HTML encoder $H$ may output a character with code $\text{hex}(63 \div 32)$ in one of its branches, where

$$\text{hex}(x) = \begin{cases} 0 & \text{if } 0 \leq x \leq 9 \\ y & \text{else } y \leq 55 \end{cases}$$

if the guard $\gamma(x) = 0 \leq x \leq 02FF$ holds for the input character $x$. However, in a double-HTML encoder $H \otimes H$, the corresponding composed guard $\gamma(\text{hex}(x \div 32)) \land \gamma(x)$ for that element is unsatisfiable, which requires nontrivial integer linear constraint reasoning in order to eliminate that branch. Such pruning requires incremental symbolic techniques outside the scope of the brute force approach.

3.2 Incremental fusion

There are several key optimizations used in the construction of composed rules, powered by the use of the solver for deciding satisfiability and for model generation of predicates. One technique is to incrementally check for unsatisfiability and validity of guards of newly formed $\text{It}e$-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 7 with further helper procedures in Figure 8. Fusion is implemented using depth first search starting from $(p, q) = (q_A', q_B')$. Only satisfiable parts of composite rules are ever explored. The procedure $\text{FUSE} (\gamma, R, q)$ in Figure 7 uses an accumulating context condition $\gamma$ for a branch of an $\text{It}e$-rule of $A$ with $R$ as the unexplored subrule in that context, and $q$ is a control state of $B$. If the condition $\text{SAT} (\gamma \land R'_1 \neq R'_2)$ is false then for all $(x, r) \in [\gamma]$, $[R'_1] (x, r) = [R'_2] (x, r)$, so the branching condition is redundant. The condition $R'_1 \neq R'_2$ is itself, w.l.o.g. expressible as a $\exists x r$-predicate. The newly discovered states in the depth first search are added to the Frontier in line 8 of the definition of PRODUCT in Figure 7.

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 $\text{RUN} (\gamma, \bar{v}, q, s)$ and $\text{STEP} (\gamma, \bar{v}, \text{rest}, R, s)$ procedures shown in Figure 8 symbolically execute the step composition operator $\otimes$ for $B$ over the symbolic value list $\bar{v}$ starting from the state $(q, s)$ of $B$. The satisfiability checks in $\text{STEP}$ on lines 6 and 10 maintain that the constructed rules only have branches that are feasible and non-redundant. A trivial case of redundancy is when both $R'_1$ and $R'_2$ are Unred, but more complicated conditional cases may arise when $R'_1$ and $R'_2$ are syntactically different but semantically equivalent in the given context $\gamma$.

Observe how the procedure $\text{FUSE}$ uses $\gamma$ on lines 5–7: $\gamma$ is included as a conjunct in every solver call to $\text{SAT}$ and every recursive call to $\text{FUSE}$. This pattern of use allows incremental SMT solving, where the solver is used in such a way that subsequent solver calls can reuse clauses learned during previous calls. For example, on line 5 in $\text{FUSE}$ 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 $\text{FUSE}$ and $\text{STEP}$ use the parameter $\gamma$ in a way such that $\gamma$ is included as a conjunct in (i) each call to $\text{SAT}$, and (ii) each $\gamma$ argument formula in recursive
Theorem 3.1.

$$[A \otimes B] = [A] \circ [B]$$

3.3 Implementation remarks

The incremental satisfiability checks that are performed during ST fusion are critical for the overall feasibility of the algorithm. In almost all of our case studies, the algorithm would not terminate otherwise. Several further optimizations are possible to locally improve the succinctness of the generated ST. One such optimization is what we call symbolic constant propagation: applying the substitution $$\theta$$ to a sub-term of $$\phi$$ in $$\text{STEP}(\gamma, v, rest, R_c)$$ may result in $$t$$ becoming "constant valued". This can be decided by checking unsatisfiability of the formula $$\gamma \land t = y$$ where $$y$$ is a fresh variable (a model exists since $$\gamma$$ is satisfiable), and extracting the value of $$y$$ from that model. In code generation, we have witnessed that symbolic constant propagation may add significant performance improvements by avoiding unnecessary expression evaluation.

4. Reachability Based Branch Elimination

Fusing already removes many unsatisfiable branches. Still, the resulting STs 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 states in the source states together with the branch is unsatisfiable. In this section we present a reachability based branch elimination (RBBE) algorithm, that proves the unreachability of and removes such branches in the target ST. The algorithm is a combination of symbolic forward reachability and backward reachability algorithms adapted to STs.

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

$$\text{Paths} \triangleq \{ (\tau \rightarrow \text{bool}) \times (\tau \rightarrow \rho) \}$$

$$\text{Paths}(\text{Base})(\text{true}, g, q) \triangleq \emptyset$$

$$\text{Paths}((\phi, u, v)) \triangleq \bigcup_{A \in Q_A} \{ ((\phi \land \psi, g, q)) \}$$

$$\text{Paths}((\psi, g, q)) \triangleq \bigcup_{A \in Q_A} \{ (\psi \land \phi, g, q) \}$$

Since outputs do not affect reachability they are dropped from the flattened representation. Given an ST $$A$$ let there be the following:

$$\text{Moves}^1(A) \triangleq \bigcup_{P \in Q_A} \{ (\phi, u, v) \}$$

$$\text{Moves}^2(A) \triangleq \bigcup_{P \in Q_A} \{ (p, \phi, g, q) \}$$

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

The Eliminate procedure in Figure 9 implements the top-level reachability algorithm. The variable $$w \in [A]_r$$ is used to represent a list of inputs. To check the reachability of a (final) move it calls ISREACHABLE with a $$([A]_r \times \rho)$$-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 ISREACHABLE returns false then the branch is eliminated by simplifying the corresponding $$\text{It}e(\phi, u, v)$$, where $$u$$ (or $$v$$) is the unreachable base rule, into $$v$$ (or $$u$$). Note that if ISREACHABLE hits the bound $$k$$ then it returns $$\bot$$ and the branch can not be safely removed.

To minimize calls to ISREACHABLE, ELIMINATE uses a more efficient 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
ELIMINATE(A)
1 let U = COMPUTEUNDERAPPROXIMATION(A)
2 let M = Moves^3(A) \cup Moves^2(A) \cup U
3 let k = |Q_A|
4 foreach move (p, ϕ, q, g) in M
5 let ϕ' = (w \in Q_A) \land \phi(x \mapsto \text{Head}(w))
6 if ISREACHABLE(A, ϕ', k) = false
7 eliminate the corresponding branch in δ_A
8 foreach final move (p, ϕ) in M
9 let ϕ' = (w \in Q_A) \land \phi
10 if ISREACHABLE(A, p, ϕ', k) = false
11 eliminate the corresponding branch in δ_A
12 remove control states with no path from q_A

Figure 9. Reachability based elimination (RBB).

ISREACHABLE(A, qtgt, φtgt, k) : (ST × Q_A × T([|A|] × ρ_A → bool)) × int) → bool
1 let layer = qtgt
2 let layer' = 0
3 let Ψ' = empty = {q \mapsto false | q \in Q_A}
4 let Σ = Ψ = empty w | qtgt \mapsto φtgt
5 while layer ≠ 0
6 while layer ≠ 0
7 pop q from layer
8 let ψ = Ψ[q]
9 if q = q_A × SAT(ψ[x \mapsto r_A])
10 return true
11 foreach (p, ϕ, g, q) in Moves^k(A)
12 if ϕ depends on x or g depends on x
13 let update = g[x \mapsto \text{Head}(w)]
14 let γ = (w \in Q_A) \land ϕ(x \mapsto \text{Head}(w)) \land ψ[w \mapsto \text{Tail}(w), x \mapsto update]
15 else
16 let γ = ψ[x \mapsto g[x \mapsto \text{default}_{\text{def}}]]
17 if SAT(γ \land ¬Σ[p])
18 let Σ[p] = Σ[p] \lor γ
19 let Ψ'[p] = Ψ'[p] \lor γ
20 add p to layer
21 if k = 0 \land layer' ≠ 0
22 return ⊥
23 let k = k - 1
24 let layer = layer'
25 let layer' = 0
26 let Ψ = Ψ'
27 let Ψ' = empty
28 return false

Figure 10. Checking the reachability of a state predicate.

These checks can be performed with an SMT solver call. For example, \exists x.r(x) (ϕ \neq ϕ[x \mapsto r]) is satisfiable iff ϕ depends on the register.

The same control state, hopefully capturing different ways of entering the control state. While more sophisticated under approximations are possible, this basic version was adequate for our experiments.

The ISREACHABLE procedure in Figure 10 performs a backward breadth-first traversal on A, exploring the states one layer at a time. Each layer is associated with the map Ψ from control states to reachability conditions yet to be explored. Initially the control state qtgt is mapped to the predicate φtgt. Σ maps control states to the predicates that summarize the arguments for which exploration has already been performed or is about to be performed.

Let δ_A denote the following partial function that extends the transition function δ of the input list and omits the output part:

δ_A : [v_A] × σ_A → σ_A
δ_A([i], s) = \left\{ δ_A(w, σ_A(i)) \right\}

A state s is k-reachable (in A) if there exists w ∈ \bigcup_{i\in[0,k]}(i, A) such that δ_A(w, s_A') = s. For example, s_A'[i] is 0-reachable. A state s is reachable if it is k-reachable for some k ≥ 0. Given q ∈ Q_A and an ρ_A-predicate ϕ, we say that (q, ϕ) is (k-reachable) if there exists a (k-)reachable state (s, r') such that r' ∈ Φ[ϕ].

Theorem 4.1. If ISREACHABLE(A, qtgt, φtgt, k) equals (a) true when (φtgt, φtgt) is reachable; (b) false then (φtgt, φtgt) is not reachable; (c) ⊥ then (φtgt, φtgt) is not k-reachable.

Proof. First, we prove the theorem with one optimization turned off.The branch condition in line 12 always returns true. Let ψtgt be the ρ_A-predicate (q \mapsto φtgt ∧ ψtgt). The algorithm maintains the following invariant that for all entries (q \mapsto φ) ∈ Σ such that φ ≠ false:

(i) SAT(φ) and (ii) for all (w, r) ∈ [φ]; (S_A, (w, q, r)) \in [ψtgt]

Property (i) follows from the observation that, other than the initial value false, only satisfiable predicates are added to Σ[q] and that satisfiability remains true under conjunctions. Property (ii) follows by induction over (w) using the definition of Δ_A and that the construction of γ in line 14 is the weakest precondition with respect to ψ and the given move from p.

Now (a) follows from the fact that if the procedure terminated in line 10 then \exists w ((w, r_A) \in Σ[q_A]). So, by (ii), \exists w (Δ_A(w, r_A) \in [ψtgt]). The proof of (c) is by induction over k, showing that all possible behaviors for input lists of up to length k that from some state lead to ψtgt are captured in Σ. This implies that the initial register must be captured in some layer, predicate Ψ_k[q_A] for there to be a path from the initial state to the target state. The satisfiability test in line 17 ensures that [φ] \subseteq Σ[p]. In other words, if the test fails then [φ] \subseteq Σ[p], so no behavior is lost by excluding φ in that case. Statement (b) follows from (c), because if false is returned for k, then false is returned for any bound greater than k.

The condition in line 12 filters out input-noise: the else-case is taken in line 16 if the input element does not affect the register update, which is when the guard does not depend on the register and the register update does not depend on the input element. In this case, the summaries in Σ may accept shorter words than what is required by Δ_A, but the register part of the predicate is not affected by omitting the input element because it does not influence it. Here we need to assume that there is no (p, ϕ, g, q) ∈ Moves^k(A) for which ϕ is unsatisfiable. Otherwise the definition of γ in line 16 is unsound when ϕ is unsatisfiable. If ϕ is satisfiable then (\exists x.r(x) (ϕ[x \mapsto \text{default}_{\text{def}}] \land ψ[x \mapsto g[x \mapsto \text{default}_{\text{def}}]]) is equivalent to ψ[x \mapsto g[x \mapsto \text{default}_{\text{def}}]]

The statement (ii) can no longer be used directly, but must be modified to count for the omitted input elements, that become much like input-eplosion moves. Intuitively, the ST is implicitly converted into an εST (ST with input-eplosion moves) although the input-eplosion moves do still count against the bound k.

In this algorithm, Σ enforces a crucial subsumption checking for predicates (line 17) — if a reachability condition ϕ for a control state p is subsumed by Σ[p], then any search from ϕ is already covered, so adding ϕ to the next layer would be redundant. A subtlety is to
We have explored several frontends for specifying effectful comprehensions. When the else case is taken, it means that \( \Sigma[p] \) works because it is sufficient in the else case (when we omit \( \phi \)). When the else case is taken, it means that

\[
\forall w, x (\varphi \Rightarrow \Sigma[p])
\]

holds, which implies that

\[
\forall x (\exists w \varphi \Rightarrow \exists w \Sigma[p])
\]

holds. Condition (2) 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 that translates imperative \( \text{C#} \) code to STs. This pattern matches interfaces present in existing streaming frameworks, which we discuss in Section 8. Some comprehensions can be more efficiently specified with a specialized frontend. In Section 5.2 we translate regexes with named captures into STs, while Section 5.3 presents a similar approach for XPath queries.

5.1 Effectful Comprehensions as \( \text{C#} \)

We have implemented a translation from a subset of \( \text{C#} \) to STs. Users extend the abstract class in Figure 11, where the Update and Finish methods respectively define \( \delta \) and \( \$ \). 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 \( \text{ToInt} \) transducer from Figure 3:

```
abstract class Transducer<I,O> {
    abstract IEnumerable<O> Update(I datum);
    virtual IEnumerable<O> Finish() { yield break; }
}
```

Figure 11. The \( \text{C#} \) abstract class users extend.

The code is parsed using the Roslyn compiler’s frontend \[54\] and \( \text{C#} \) is 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

\[
\forall x (\exists w \varphi \Rightarrow \exists w \Sigma[p])
\]

holds. Condition (2) is the necessary condition needed to preserve all register values.

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

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

where \( S_i \) and \( P_j \) are regular expressions such that no \( P_j \) accepts the empty string and there is no ambiguity about where each \( S_i \) ends or where each \( P_j \) 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 \( P_j \) 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 \( \{cap_i \mapsto A_i\} \) constructs a fused transducer that parses strings matching \( P \) into \( n \)-tuples \( \langle o_1, \ldots, o_n \rangle \). The algorithm works as follows:

1. Parse and translate the regex into a finite symbolic automaton \[32\].
2. Keep track of which parts of the resulting automaton accept the patterns \( P_i \). The input values accepted inside any such part of the automaton represents a match of the capture group (with no ambiguity due to our assumptions).
3. Fuse each identified part of the automaton separately with the appropriate ST \( A_i \). The start and end of a capture group match respectively trigger initialization and finalization of the ST.

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

Example 5.2. The following regex illustrates a case that parses a 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 control flow path, while infeasable paths are cut with satisfiability checks using Z3 \[12\]. The exploration produces an execution tree that corresponds to an ST with a single control state and a branching rule such that each internal node is an Ite-rule and each leaf node is either an Undef-rule (if the path ended with a throw statement) or a Base-rule (otherwise).

The register type is the product of all the field types. For example \( pr\text{class} = \text{int} \times \text{bool} \). Subsequently, the register type is split into \( \rho \times \kappa \) where \( \kappa \) is a product of all the types with a small set of values (either \( \text{enum} \) or \( \text{bool} \) types). An algorithm called (finite) exploration is used to partially evaluate the transition function so that the new control state set \( Q \) becomes a finite set of elements representing values of type \( \kappa \) and the new register type becomes \( \rho \). The algorithm is incremental: it starts from the initial values and only considers reachable values of type \( \kappa \). It is a variant of the ST exploration algorithm discussed in \[34, Figure 4\] but without grouping. The intent here is not to attempt to completely eliminate registers because that is undecidable, while finite exploration is guaranteed to terminate.

The supported \( \text{C#} \) subset includes:

- Integral types, booleans and structs; and their operators.
- All control flow constructs except try-catch.
- Calls into pure and side effect free functions.

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

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

where \( S_i \) and \( P_j \) are regular expressions such that no \( P_j \) accepts the empty string and there is no ambiguity about where each \( S_i \) ends or where each \( P_j \) 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 \( P_j \) 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 \( \{cap_i \mapsto A_i\} \) constructs a fused transducer that parses strings matching \( P \) into \( n \)-tuples \( \langle o_1, \ldots, o_n \rangle \). The algorithm works as follows:

1. Parse and translate the regex into a finite symbolic automaton \[32\].
2. Keep track of which parts of the resulting automaton accept the patterns \( P_i \). The input values accepted inside any such part of the automaton represents a match of the capture group (with no ambiguity due to our assumptions).
3. Fuse each identified part of the automaton separately with the appropriate ST \( A_i \). The start and end of a capture group match respectively trigger initialization and finalization of the ST.

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

Example 5.2. The following regex illustrates a case that parses a 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 control flow path, while infeasable paths are cut with satisfiability checks using Z3 [12]. The exploration produces an execution tree that corresponds to an ST with a single control state and a branching rule such that each internal node is an Ite-rule and each leaf node is either an Undef-rule (if the path ended with a throw statement) or a Base-rule (otherwise).

The register type is the product of all the field types. For example \( pr\text{class} = \text{int} \times \text{bool} \). Subsequently, the register type is split into \( \rho \times \kappa \) where \( \kappa \) is a product of all the types with a small set of values (either \( \text{enum} \) or \( \text{bool} \) types). An algorithm called (finite) exploration is used to partially evaluate the transition function so that the new control state set \( Q \) becomes a finite set of elements representing values of type \( \kappa \) and the new register type becomes \( \rho \). The algorithm is incremental: it starts from the initial values and only considers reachable values of type \( \kappa \). It is a variant of the ST exploration algorithm discussed in [34, Figure 4] but without grouping. The intent here is not to attempt to completely eliminate registers because that is undecidable, while finite exploration is guaranteed to terminate.

The supported C# subset includes:

- Integral types, booleans and structs; and their operators.
- All control flow constructs except try-catch.
- Calls into pure and side effect free functions.
integer in decimal notation and the substring in the fourth column is parsed as a Boolean:

\[
((\lceil,\rceil,\rceil,?<\text{int}>\rceil,\ldots,?<\text{int}>\rceil,?<\text{bool}>\rceil,?<\text{bool}>\rceil,?\text{\emph{\textbackslash n}})\rceil)\rceil
\]

Here \( S_1 \) is "\( ([\lceil,\rceil,\rceil,?\text{\emph{\textbackslash n}}] \rceil,\ldots,?\text{\emph{\textbackslash n}}) \rceil \) (skip to the third column), \( S_2 \) is "\( ,\)" (skip to the next column), and \( S_3 \) is "\( ,?\text{\emph{\textbackslash n}}\rceil \)" (skip remaining columns until EOL). The capture int is mapped to the transducer ToInt from Figure 3 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 expressions. Consider an expression \( X \) of the form

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

The tag names \( \text{tag}_o \) 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 matcher 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:

```xml
<cities>
  <city name='Roslyn'>
    <timezone>PST</timezone>
    <population>893</population>
  </city>
  <city name='Santa Barbara'>
    <population>88410</population>
  </city>
</cities>
```

A transducer based on the following XPath expression will extract the populations in the dataset:

\[
\text{st:int}(//\text{cities}/\text{city}/\text{population})
\]

int again maps to the ToInt transducer from Figure 3.

6. Evaluation

We have implemented the techniques described above in a tool that translates C# (and our other frontends) into STs, fuses them and finally generates efficient C# code. 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 12 presents throughputs for three variations of each pipeline. For LINQ the pipelines communicate with IEnumerable\(\langle T\rangle\) and yield. The Hand-written pipelines are straightforward implementations using arrays as buffers between phases. The fused and optimized pipelines are labeled Fused. The individual pipeline stages in the LINQ and Fused pipelines use code generated from STs 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. Four pipelines were benchmarked:

- **Base64-avg** calculates a running average (window of 10) for Base64\(^9\) 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 and finds the maximum length of these strings. The output is 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.
- **UTF8-lines** decodes an UTF-8 encoded file to UTF-16 and counts the number of newline characters. The output is a single UTF-8 encoded decimal formatted integer.

For Figure 12 we sampled the pipelines 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 lowest throughput. We believe this is due to the overhead associated with passing values through IEnumerable\(\langle T\rangle\).

Figure 13 presents a more detailed comparison of CSV parsing scenarios. Pipelines for three different datasets are compared:

- **CHSI** is a dataset on health indicators from the U.S. Department of Health & Human Services. The three pipelines produce the average lung cancer deaths, minimum births and maximum total deaths for counties in the dataset.
- **SBO** is a dataset on business owners from the U.S. Census Bureau. The three pipelines find the maximum employees, minimum gross receipts and average payroll for businesses in the dataset.
- **CC** is a dataset of consumer complaints received by the U.S. Consumer Financial Protection Bureau. The pipeline produces the maximum value for the ID column.

Each of the Fused pipelines in Figure 13 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.


\(^{10}\)See https://en.wikipedia.org/wiki/Base64.
Four pipelines are compared:

- CHSI-cancer
- CHSI-births
- CHSI-deaths
- SBO-employees
- TPC-DI-SQL
- DBLP-oldest
- MONDIAL-pop
- SBO-receipts
- SBO-payroll
- PIR-proteins

The original SBO dataset is 744 MB, which caused the .NET framework’s XmlDocument to run out of memory. To work around this problem, we used an XPath based transducer 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. An optimization we implemented for the Hand-written pipelines is that the regular expression is matched for the whole dataset and all of 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. Our fused pipelines are free of such limitations due to their incremental nature.

The fused pipelines are significantly faster for all benchmarks, with an average speedup of 11× 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 LINQ 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, as the speedup will make up for the compilation time.

Figure 15 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.

### 7. Symbolic Transducers and Monads

This section provides redefinitions of the \( \odot \) and \( \otimes \) operators in terms of application functors and monads. In addition to being concise, these definitions provide a link between an automata-theoretic view and a functional view of symbolic transducers. This is to our knowledge the first time transductions have been successfully related to monads, which has been unsuccessfully attempted before. For more discussion and the exact connection to LINQ’s list monad see Section 8.

Given types \( \sigma \) and \( \tau \) we define \( \text{TM}^* \sigma \tau \) as the type \( \sigma \to (\tau \times \sigma) \tau^\perp \), which as we will later show is a transduction monad. We use the higher order applicative functor [21] operators pure and \( \star \), defined

\[ \text{pure} \quad \sigma \to (\tau \times \sigma) \tau^\perp \]

\[ \star \quad (\tau \times \sigma) \tau^\perp \to \tau \]
as follows:

\[
\begin{align*}
\text{pure} & : \tau \rightarrow \text{TM}^\tau \\
\text{pure } f & \triangleq \lambda s. |(f; s) \\
\ast & : \text{TM}^\tau (\tau \rightarrow \tau') \rightarrow \text{TM}^\tau \rightarrow \text{TM}^\tau \tau' \\
F \ast X & \triangleq \lambda s. \text{let } |(f; s) = (F s) \text{ in } \\
& \quad |(x, s_2) = (X s_1) \text{ in } \\
& \quad |(f x, s_2)
\end{align*}
\]

The intuition is that \(\ast\) captures side-effects of \(F\) in \(s_1\) and propagates them to \(X\) which produces side-effects \(s_2\) while the output is \((f x)\).

Now the step composition operator \(\odot\) can also be defined as

\[
\begin{align*}
\odot & : \text{TM}^\tau \rightarrow \text{TM}^\tau \rightarrow \text{TM}^\tau \\
f \odot g & \triangleq \langle \text{pure } + \rangle \ast f \ast g
\end{align*}
\]

where + denotes list concatenation, although other operators, such as addition, maximum, and minimum, could be used. One reason why such operators are interesting is that they allow us to define aggregation operations without explicitly using state for accumulating the intermediate result. Regardless of the operator used as +, the purpose of \(\odot\) is to compose together output results while propagating the effects of the computations “from left to right” as loop carried state.

The bind operator for \(\text{TM}\) is

\[
\begin{align*}
\gg & : \text{TM}^\tau \tau \rightarrow (\tau_1 \rightarrow \text{TM}^\tau \tau_2) \rightarrow \text{TM}^\tau \tau_2 \\
F \gg G & \triangleq \lambda s. \text{let } |(a, s') = (F s) \text{ in } (G a s')
\end{align*}
\]

We may now view \(\text{TM}^\tau\) as a transduction monad with the given bind operator and whose unit operator is \(pure\). It follows from the definitions that the monad laws hold. One can view this monad as a combination of the state monad and the option monad.

The fusion composition operator \(\otimes\) (Section 3) can be defined using the bind operator. First let there be:

\[
\begin{align*}
\text{fuse} & : (\langle \tau \rangle \rightarrow \text{TM}^\tau [\tau]) \rightarrow (\tau \rightarrow \text{TM}^\tau \rho [\rho]) \rightarrow \\
& \quad (\langle \tau \rangle \rightarrow \text{TM}^\tau \times \rho [\rho]) \\
\text{fuse } A B & \triangleq \lambda \bar{x}. (A \bar{x}) \gg B' \text{ where } \\
A' \bar{a} & \triangleq \lambda(s_1, s_2). \text{let } |(\bar{b}, s_1') = (A \bar{s} s_1) \text{ in } |(\bar{b}, (s_1', s_2)) \\
B' \bar{b} & \triangleq \lambda(s_1', s_2). \text{let } |(\bar{c}, s_2') = (B \bar{b} s_2) \text{ in } |(\bar{c}, (s_1', s_2'))
\end{align*}
\]

Note how in fuse the ST \(A\) uses its own state that is disjoint from the state of \(B\), and the function builds the disjoint sum of the states. Further, notice that the output \(\bar{b}\) of \(A\) may depend on the state \(s_1\), so the state \(s_2\) may, through \(b\), depend on \(s_1\), whereas \(s_1'\) does not depend on \(s_2\). The latter property is integral to the fusion algorithm in Section 3. Now \(\otimes\) can be defined as:

\[
\langle A, s_0^A \rangle \otimes \langle B, s_0^B \rangle \triangleq (\text{fuse } A B, (s_0^A, s_0^B))
\]

Note that here we represent an ST \(A\) as a pair of a function of type \(\langle \tau \rangle \rightarrow \text{TM}^\tau [\rho A]\) and the initial state of \(A\). To run transducers represented like this the following can be used:

\[
\begin{align*}
\text{runST} & : (\langle \tau \rangle \rightarrow \text{TM}^\tau [\rho]) \times \sigma \rightarrow [\tau] \rightarrow \rho \\
\text{runST } (A, s) & \triangleq \lambda \bar{x}. \text{let } |(\bar{y}, s') = (A \bar{x} s) \text{ in } \bar{y}
\end{align*}
\]

Effectively, given an ST \(A\), \((\text{runST } (A, s_0^A))\) is its denotation \([A]\).

In functional languages the state monad is typically implemented using lazy evaluation and \(\text{fuse}\) could in principle be implemented similarly. In contrast to these languages wherein unfeasible paths are never explored by virtue of lazy evaluation, the fusion algorithm in Section 3 implements a statically optimized binding operator for the transduction monad which statically prunes unfeasible paths. We believe similar static fusion techniques could also be applied to code written using the state monad.

8. Related Work

Symbolic transducers: were originally defined in flat form in [33]. The main focus of the work in [33] is on symbolic finite transducers or SFTs, for analysis of string sanitizers. It is noted in [33] that STs are closed under composition, but, to the best of our knowledge, no algorithm for fusing STs has been studied prior to our work. Prior work on STs has focused on register exploration and input grouping that are orthogonal problems [11, 34]. 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 [11] and parallelization [34]. Efficient fusion of STs has, to the best of our knowledge, not been studied prior to our work.

**Streaming:** There is a large body of work on stream-processing [14, 20, 23, 24, 30]. There is also recent work on a domain specific language DReX [8] for expressing regular string transformations. Stream computations with internal state have been studied before. The work in [10] 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 [19], 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 [7] and Spark Streaming [5] 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 [3] 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 STs. Implementing frontends similar to the C# one (Section 5.1) for these libraries would allow code written for them to use our backend.

StreamIt [31] is a programming language and compiler for signal processing applications. StreamIt composes pipelines of stateless filters with the aim of reducing communication overhead. In [6] composition is extended to filters with a linear state space representation, i.e., ones where the outputs and state updates are linear operations. The composition retains the linear state space representation with a linear increase in size.

In contrast to StreamIt, we can compose any stateful filters where the state update is over a decidable theory, and instead of linear algebra we use SMT solvers for our analysis. We view the work done by the StreamIt group as complimentary to ours: the composition and optimization techniques for symbolic transducers could be used as an additional backend module in the StreamIt compiler for stateful filters which are not amenable to a linear state space representation.

**Monads:** have had a huge impact on programming paradigms and techniques in general after they were introduced into the func-
tional programming world by Wadler [36]. One of the core contributions of monads is that they provide a type discipline by which one can enforce a separation of computational concerns in a clean functional style. A prime example is the state monad [37]. Another very useful monad is the maybe monad [36]. Our transduction monad type $\text{TM}^\tau$ is more-or-less the type for the maybe state monad parameterized with the state type $\tau$ and the output type $\tau$, and extended with extra composition operations for step and fusion composition. The fusion composition operator $\otimes$ is based on the monad binding operator $\gg=$ but is itself not a binding operator because it uses different monad state types. The “maybe” part in the transduction monad reflects the fact that (deterministic) transducers are typically partial functions and their composition (that corresponds exactly to fusion composition here) is often treated as a special case of relational composition.

LINQ [22] uses the list monad (or list comprehension [36]) 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 $\text{IEnumerable}<T>$ class. The exact relation to the transduction monad is that the list comprehension in LINQ corresponds to iterating the step composition operator $\otimes$ (Section 2) over the input list. Step composition handles loop carried state. The LINQ query

```
"Man".SelectMany(A.Update)
```

corresponds to the following transduction or effectful comprehension, provided that we apply it to the initial state of $A$:

$$(\delta_A \ 'M') \otimes (\delta_A \ 'a') \otimes (\delta_A \ 'n')$$

The state of the computation $(\delta_A \ 'n')$ is threaded through into the computation $(\delta_A \ 'a')$, etc. For example, if we take $A$ to be the Base64 encoder, and we start from the initial state (at the point when no characters have been read so far) then the output would be the string “TMFu”. This is consistent with the existing semantics of LINQ.

In Figure 4 in Section 1 the finalizer for ToInt can be implemented as a separate piece of code after the state has been aggregated. However, for transducers whose Update function produces output the following pattern would be natural: SelectMany(i => Update(i)).Concat(Finalize()). where Finalize returns an $\text{IEnumerable}<T>$. This pattern is semantically correct, but relies on the fact that Concat evaluates its parameter lazily. With eager evaluation, Finalize would access state before Update had been called for all inputs. We feel this reliance on subtle semantics makes LINQ a poor match for writing effectful comprehensions. This is another concern we address with our C# front-end.

Fusion: For fusion of symbolic transducers there is related work on filter fusion [27] and deforestation [35]. Fusion of symbolic transducers can be viewed as an extended form of filter fusion that incorporates loop carried state and advanced constraint satisfaction techniques into the classical framework.

The Steno library in [25] implements deforestation for LINQ queries and achieves speedups from removing the $\text{IEnumerable}$ 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 [15] 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 computational complexities.

The work in [29] is related to our work regarding motivation. The difference is in the execution, 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. We build on some of the work in [33] by extending it with an incremental fusion algorithm and reachability analysis. The authors of [29] were not able to relate their work to monads but use the SML type system in general. In our case the definition of the step composition operator $\otimes$ uses applicative functors or idioms [18, 21] — it does not require full monad functionality.

Regex: Our construction of symbolic transducers from regexes is related to the work in [28]. On one hand our algorithm only handles a special class of regexes, but on the other hand it supports full Unicode by using the .NET regex parser and represents guards by predicates over 16-bit bit-vectors (i.e., the char type). Regexes are very handy for capturing custom patterns, for example for some specific CSV file or some specific alphabet (such as the emoticon alphabet\(^{12}\)). This is reminiscent to handling hierarchical data, such as XML, but with more relaxed rules, e.g., a line in a custom CSV file may (or may not) end with a comma.

To handle XML data we use transducers generated from a subset of the XPath query language. For a full automata theoretic treatment of XPath see [9], where an approach for evaluating and reasoning about XPath expressions (extended with regular expressions) based on two-way weak alternating tree automata is presented.

List comprehensions have also been extended with ORDER BY and GROUP BY constructs [17] that are also supported in LINQ. It is an ongoing research topic for us to investigate whether symbolic transducers can be extended similarly and, if so, to understand what the potential payoffs are.

9. Conclusion

Good abstractions let a programmer easily express their intent as a program and at the same time let a runtime system compile that program for efficient execution. This paper puts forth effectful comprehensions as an abstraction for expressing possibly-stateful data-processing pipelines. We present fusion and branch elimination algorithms for these effectful comprehensions, which allow us to compile large pipelines into efficient code.

We use symbolic transducers to represent individual and fused stages in a data-processing pipeline, which we additionally formalize with transduction monads. The monadic view provides very concise semantics for transductions and their compositions. On the other hand, our fusion and branch elimination algorithms use an automata-theoretic view, which allows them to exploit the separation of control-state from other state.

We have built a compiler that ingests pipelines written in C# and produces fused code that runs, on average, $3 \times$ faster than a hand-written baseline and $5 \times$ faster than LINQ on a variety of data processing programs. In the future we will explore more extensive optimizations that rely on background theory reasoning to prove program properties. One such optimization we excluded from this paper due to space constraints exploits minimization of symbolic finite automata to simplify control flow.

In the future we intend to explore hierarchical compositions, i.e., parts of an effectful comprehension being specified in terms of another. In Sections 5.2 and 5.3 we use a specific pattern of hierarchical composition for which fusion is straightforward. We aim to expand this work to allow hierarchical compositions in our general C# front-end (Section 5.1).

\(^{12}\) See http://unicode.org/charts/PDF/U1F600.pdf
References


