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.

CCS Concepts • **Theory of computation** → **Transducers**; *Streaming models*; *Program analysis*; • **General and reference** → Evaluation; Performance; • **Software and its engineering** → Abstraction, modeling and modularity; Source code generation; Domain specific languages

Keywords symbolic transducer, symbolic automaton, comprehension, fusion, deforestation, reachability analysis

PLDI'17, June 18–23, 2017, Barcelona, Spain
© 2017 ACM. 978-1-4503-4988-8/17/06...$15.00
http://dx.doi.org/10.1145/3062341.3062362

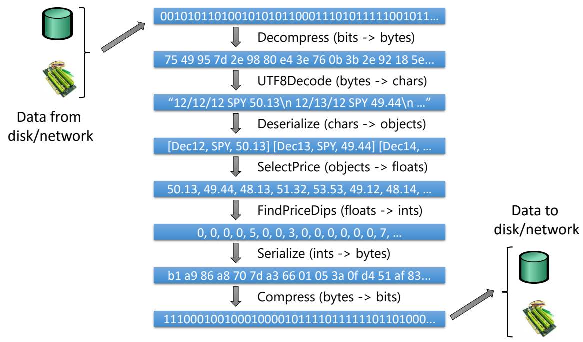
1. Introduction

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.

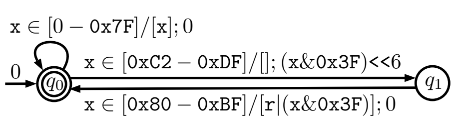
Note that the stages in the pipeline include both "functional" computations that operate on each input independently, such as `SelectPrice`, and "effectful" computations that iterate over the input list while maintaining loop-carried state, such as `Decompress`, `Deserialize`, and `FindPriceDips`. The 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

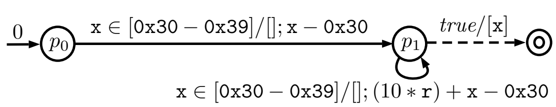



```
output =
   input.Decompress(...).Decode(...).Deserialize(...)
        .SelectPrice(...).FindPriceDips(...)
        .Serialize(...).Compress(...);
```

Figure 1. Motivating example of a log processing pipeline where an input stream of bits goes through various stages to an output stream of bits. This paper allows programmers to declaratively specify this pipeline as a composition of symbolic transducers, as shown at the bottom.



(a) *Utf8Decode*



(b) *ToInt*

Figure 2. Symbolic transducers

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]

The following paragraphs serve as an informal introduction to symbolic transducers. *Utf8Decode* is illustrated in Figure 2(a)[2]. *Utf8Decode* uses two control states q_0 and q_1, where q_0 is both the initial and the final state. The variables $\mathtt{x}$ and $\mathtt{r}$ are used for the current input and current register values, respectively. A transition $p \xrightarrow{\mathtt{x}\in\alpha/s;g} q$ has the following meaning: if the current state is p and the current byte $\mathtt{x}$ is in the range α then enter state q, yield the elements in the sequence s and update the register $\mathtt{r}$ to the value g. Initially $\mathtt{r}$ has the value 0. For example, if the input sequence of bytes is $[\mathtt{0x61}, \mathtt{0xC5}, \mathtt{0x93}]$ then the output sequence of character codes is $[\mathtt{0x61}, ((\mathtt{0xC5}\&\mathtt{0x3F})\ll 6)|(\mathtt{0x93}\&\mathtt{0x3F})]$ that is equal to $[\mathtt{0x61}, \mathtt{0x153}]$ or the string "aœ".

ToInt is illustrated in Figure 2(b). 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 $[\mathtt{r}]$. 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 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 $q_0 \xrightarrow{\mathtt{x}\in[0-\mathtt{0x7F}]/[\mathtt{x}];0} q_0$ with the transition $p_0 \xrightarrow{\mathtt{x}\in[\mathtt{0x30}-\mathtt{0x39}]/[];\mathtt{x}-\mathtt{0x30}} p_1$ produces the transition
$(q_0, p_0) \xrightarrow{\mathtt{x}\in[\mathtt{0x30}-\mathtt{0x39}]/[];(0,\mathtt{x}-\mathtt{0x30})} (q_0, p_1)$
where the fused register is a pair representing the registers of *Utf8Decode* and *ToInt* and the output $\mathtt{x}$ 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
$(q_0, p_1) \xrightarrow{\mathtt{x}\in[\mathtt{0xC2}-\mathtt{0xDF}]/[];(\mathtt{x}\&\mathtt{0x3F})\ll 6,\pi_2(\mathtt{r}))} (q_1, p_1)$
π_1 and π_2 project the first and second element of a pair, respectively. The only possible fusion of transitions from (q_1, p_1) is

$$(q_1, p_1) \xrightarrow{\substack{\mathtt{x}\in[\mathtt{0x80}\,-\,\mathtt{0xBF}]\wedge \\ (\pi_1(\mathtt{r})|(\mathtt{x}\&\mathtt{0x3F}))\in[\mathtt{0x30}\,-\,\mathtt{0x39}]/[]; \\ (0,\,(10\,*\,\pi_2(\mathtt{r}))\,+\,(\pi_1(\mathtt{r})|(\mathtt{x}\&\mathtt{0x3F}))\,-\,\mathtt{0x30})}} (q_0, p_1).$$

However, the state (q_1, p_1) is associated with the register constraint $\exists\mathtt{x}\,(\mathtt{x} \in [\mathtt{0xC2} - \mathtt{0xDF}] \wedge \pi_1(\mathtt{r}) = ((\mathtt{x}\&\mathtt{0x3F})\ll 6))$ 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 back-propagating 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 become more structured and filtered.

[1] UTF8 encodings up to two bytes cover the full range of characters in extended ASCII. In general, UTF8 encodings of up to four bytes cover all Unicode characters.

[2] The operation '&' denotes bitwise-and, the operation '|' denotes bitwise-or, and the operation '$\ll k$' denotes shift-left by k bits.

```
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) {
        r1 = (x & 0x3F) << 6; 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) // ToInt's finalizer
    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 hand-written 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

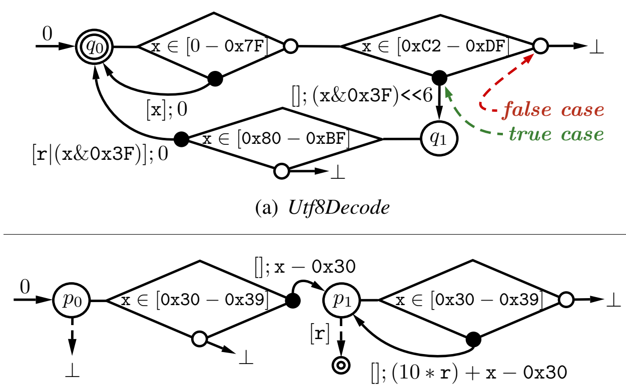
This section formally introduces *branching symbolic transducers* or *BSTs*, as a generalization of *deterministic symbolic finite transducers* or *deterministic SFTs* [43] by incorporating registers. At the same time the definition is a specialization of nondeterministic symbolic transducers [43] since 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 $\perp$ 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.



(b) *ToInt*

Figure 4. BSTs.

to Figure 2(a) after completion.[5] 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 $\mathtt{x} \in [\mathtt{0xC2} - \mathtt{0xDF}]$ 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 τ and σ, $\tau \times \sigma$ and $\tau \to \sigma$ stand for the standard Cartesian product and function types, respectively.[6] The type for Booleans is `bool` with truth values `true` and `false`. Let $\mathcal{T}(\tau)$ denote a given *predefined* set of terms t that denote values $[\![t]\!]$ of type τ. In our implementation we use Z3 [18] 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 $\mathcal{T}((\tau \to \sigma) \to \rho)$, 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 $\mathcal{T}(\tau \to \mathtt{bool})$ is a τ-*predicate*.

Let $[\tau]$ denote the type of finite-length lists of elements of type τ. 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 τ. We assume that if τ is a Cartesian product type $\tau_1 \times \tau_2$ then there are *projection* functions $\pi_1 : \tau \to \tau_1$ and $\pi_2 : \tau \to \tau_2$ and a *pairing* function $\langle\cdot,\cdot\rangle : \tau_1 \to \tau_2 \to \tau_1 \times \tau_2$ with the intended semantics that $[\![\langle t_1, t_2\rangle]\!] = ([\![t_1]\!], [\![t_2]\!])$, and $[\![\pi_1\langle t_1, t_2\rangle]\!] = [\![t_1]\!]$, and $[\![\pi_2\langle t_1, t_2\rangle]\!] = [\![t_2]\!]$. 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 τ denotes a nonempty set and has a default element $\__\tau$ (or $\_$). We write τ both for a type and the denoted set and we write π_1 for $[\![\pi_1]\!]$ 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 ι, o and ρ are the input, output and register types; Q is the finite set of control states; $s^0 \stackrel{\text{def}}{=} (q^0, r^0)$ is the initial state of *state type* $\sigma \stackrel{\text{def}}{=} Q \times \rho$;

$$\delta : Q \to \mathcal{R}(\iota \times \rho, o, Q, \rho), \qquad \$: Q \to \mathcal{R}(\rho, o, Q, \rho)$$

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 *Ite* ("if-then-else") choice and every leaf is either a *Base* case that performs a state transition or an *Undef* case that represents a transition to an implicit rejecting state $\perp$. Such a tree is interpreted as a function of type $\tau \to ([o] \times \sigma) \cup \{\perp\}$ or a *partial function* of type $\tau \to [o] \times \sigma$, where $\sigma = Q \times \rho$. A rule for the transition function has $\tau = \iota \times \rho$ 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:

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

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

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

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

$$[\![\mathit{Undef}]\!]\, v = \perp$$

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 ι. Instead of lifting every input type ι to a sum type of ι 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 ι and refers to the input element and variable `r` is of type ρ 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 $(\mathtt{x} = t)$ the `x` is a literal part of the formula, while t refers to some term.

[5] Completeness of a flat ST means that the disjunction of all the guards of transitions from any given state is equivalent to *true*.

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

Simultaneous substitution of variables $\mathtt{y}_i$ by terms u_i in t is denoted $t\{u_1/\mathtt{y}_1, \ldots, u_n/\mathtt{y}_n\}$.

In *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:

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

The finalizer of *ToInt*, in Figure 4(b), that is shown as the dashed arrow, is the function:

$$\$_{\mathit{ToInt}} = \{p_0 \mapsto \mathit{Undef}, p_1 \mapsto \mathit{Base}([\mathtt{r}], p_1, 0)\},$$

where the final value of the register $\mathtt{r}$ 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 $[\iota] \to [o]$. First, we define the partial functions $\hat{\delta} : \iota \to \sigma \to [o] \times \sigma$ and $\hat{\$} : \sigma \to [o] \times \sigma$ that enable us to provide a declarative definition of $[\![A]\!]$:

$$\hat{\delta}\, a\, (q, b) \stackrel{\text{def}}{=} [\![\delta\, q]\!](a, b); \quad \hat{\$}\, (q, b) \stackrel{\text{def}}{=} [\![\$\, q]\!]\, b.$$

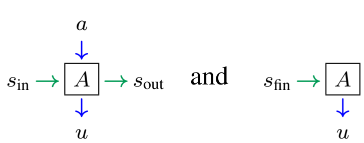
Let $\bar{a} = [a_i]_{i=1}^k$ be a given input list. Then

$$[\![A]\!]\, \bar{a} \stackrel{\text{def}}{=} \pi_1(((\hat{\delta}\, a_1) \oplus \cdots \oplus (\hat{\delta}\, a_k) \oplus \hat{\$})s^0) \tag{1}$$

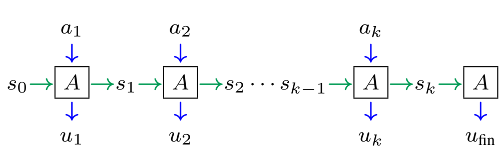
where $\oplus$ is a left-associative operator that composes single-input transduction steps into multi-input transduction steps:

$$\begin{aligned} \oplus &: (\sigma \to [o] \times \sigma) \times (\sigma \to [o] \times \sigma) \to (\sigma \to [o] \times \sigma) \\ F_1 \oplus F_2 &\stackrel{\text{def}}{=} \lambda s. \textbf{let } (u_1, s_1) = (F_1\, s) \textbf{ in} \\ &\qquad\quad \textbf{let } (u_2, s_2) = (F_2\, s_1) \textbf{ in } (u_1 + u_2, s_2) \end{aligned}$$

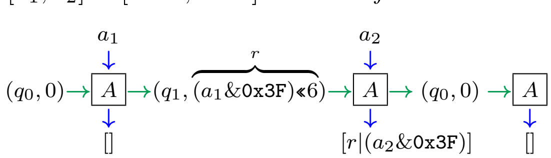
where $+\!\!+$ is list concatenation, $\oplus$ is called *step composition*. If we depict $(\hat{\delta}_A\, a\, s_{\text{in}}) = (u, s_{\text{out}})$ and $(\hat{\$}_A\, s_{\text{fin}}) = (u, \_)$ by



respectively, then



depicts Equation (1), where '$\downarrow$' shows list comprehension, similar to for example `SelectMany` list comprehension in LINQ, and '$\rightarrow$' shows state evolution. For example let $\bar{a} = [a_1, a_2] = [\mathtt{0xC5}, \mathtt{0x93}]$ and A=*Utf8Decode*. Then



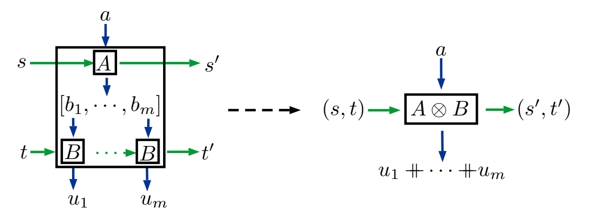


Figure 5. Fusion.

The result is $[] + [((a_1\&\mathtt{0x3F}){\ll}6)|(a_2\&\mathtt{0x3F})] + []$ that equals $[\mathtt{0x153}]$ and represents the string `"œ"`.

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 $\hat{\delta}\, q_\perp = \mathit{Undef}$ and $\hat{\$}\, q_\perp = \mathit{Undef}$ and, for all v, $[\![\mathit{Undef}]\!]v = ([], q_\perp, \_)$. In this case the definition of $[\![A]\!]\bar{a}$ has to be modified so that $[\![A]\!]\bar{a}$ is undefined or $\perp$ whenever $\pi_1(\pi_2(((\hat{\delta}\, a_1) \oplus \cdots \oplus (\hat{\delta}\, a_k) \oplus \hat{\$})s^0)) = q_\perp$ in order to distinguish the accepted inputs from the rejected inputs.

3. Fusion of BSTs

Consider two BSTs A and B such that $o_A = \iota_B$. 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 explain the main 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: $\iota = \iota_A$, $o = o_B$, $\rho = \rho_A \times \rho_B$, $Q \subseteq Q_A \times Q_B$, $r^0 = (r_A^0, r_B^0)$, $q^0 = (q_A^0, q_B^0)$. The goal of the fusion algorithm is to construct $\delta_{A \otimes B}$ and $\$_{A \otimes B}$. See Figure 5.

For each pair (p, q) of control states in $Q_A \times Q_B$ build a fused rule that, given the rule $\delta_A\, p$, symbolically runs δ_B repeatedly, starting from q, over each of the output lists $[v_i]_{i=1}^n$ that occur in the *Base*-subrules of $\delta_A\, p$ as symbolic values. The symbolic values are substituted into the register update and output functions of $(\hat{\delta}_B v_1) \oplus \cdots \oplus (\hat{\delta}_B v_n)$, that is partially evaluated with respect to the control state q, and finally normalized into a rule in $\mathcal{R}(\iota \times \rho, o, Q, \rho)$. 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 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 $\mathit{hex}(x \div 32)$ in one of its branches, where

$$\mathit{hex}(y) = \mathit{if}\ (0 \leq y \leq 9)\ \mathit{then}\ (y + 48)\ \mathit{else}\ (y + 55)$$

if the guard $\gamma(x) = \mathtt{0x100} \leq x \leq \mathtt{0xFFF}$ holds for the input character x. However, in a double-HTML encoder $H \otimes H$, the corresponding composed guard $\gamma(\mathit{hex}(x \div 32)) \wedge \gamma(x)$ for that element is *unsatisfiable*, which requires advanced integer constraint reasoning in order to eliminate that branch. Such pruning requires symbolic techniques outside the scope of the brute force approach.

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 $\textsc{Fuse}_\delta(\gamma, R, q)$ in Figure 6 uses an accumulating context condition γ 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 $\mathbf{SAT}(\gamma \wedge R_1' \neq R_2')$ 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 $\iota{\times}\rho$-predicate. The newly discovered states in the depth first search are added to the *Frontier* in line 8 of the definition of $\textsc{Prod}$ 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 $\textsc{Run}(\gamma, \bar{v}, q, s)$ and $\textsc{Step}(\gamma, v, \mathit{rest}, R, s)$ 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 $\textsc{Step}$ 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' and R_2' are *Undef*, but more complicated conditional cases may arise when R_1' and R_2' are syntactically different but semantically equivalent in the given context γ.

$A \otimes B \stackrel{\text{def}}{=}$

1 **let global** *Frontier* $= \{(q_A^0, q_B^0)\}$
2 **let global** $Q = \{(q_A^0, q_B^0)\}$
3 **let** $\delta = \$ = \emptyset$
4 **while** *Frontier* $\neq \emptyset$
5   **remove** (p, q) **from** *Frontier*
6     $\delta(p, q) \mapsto \textsc{Fuse}_\delta(\mathtt{true}, (\delta_A\, p), q)$
7     $\$(p, q) \mapsto \textsc{Fuse}_\$(\mathtt{true}, (\$_A\, p), q)$
8 **return** $(\iota_A, o_B, \rho_A{\times}\rho_B, Q, (q_A^0, q_B^0), \langle r_A^0, r_B^0\rangle, \delta, \$)$

$\textsc{Fuse}_\delta(\gamma, R, q) : (\mathcal{T}(\iota{\times}\rho{\to}\mathtt{bool}){\times}\mathcal{R}(\iota{\times}\rho_A, o_A, Q_A, \rho_A)$
$\qquad {\times}Q_B) \to \mathcal{R}(\iota{\times}\rho, o, Q, \rho)$

1 **let** $\theta = \{\pi_1(\mathtt{r}{:}\rho)/\mathtt{r}{:}\rho_A\}$
2 **match** R
3   **case** *Undef*: **return** *Undef*
4   **case** $\mathit{Ite}(\varphi, R_1, R_2)$:
5   **let** $R_1' = \textsc{Fuse}_\delta(\gamma \wedge (\varphi\theta), R_1, q)$
6   **let** $R_2' = \textsc{Fuse}_\delta(\gamma \wedge \neg(\varphi\theta), R_2, q)$
7   **if** $\mathbf{SAT}(\gamma \wedge R_1' \neq R_2')$
8   **return** $\mathit{Ite}(\varphi\theta, R_1', R_2')$
9   **else return** R_1'
10   **case** $\mathit{Base}(\bar{v}, p, g)$:
11   **return** $\textsc{Prod}(p, g\theta, \textsc{Run}(\gamma, \bar{v}\theta, q, \pi_2(\mathtt{r}{:}\rho)))$

$\textsc{Prod}(p, g, R)$

1 **match** R
2   **case** *Undef*: **return** *Undef*
3   **case** $\mathit{Ite}(\varphi, R_1, R_2)$:
4   **return** $\mathit{Ite}(\varphi, \textsc{Prod}(p, g, R_1), \textsc{Prod}(p, g, R_2))$
5   **case** $\mathit{Base}(\bar{v}, q, h)$:
6   **if** $(p, q) \notin Q$
7   **add** (p, q) **to** Q
8   **add** (p, q) **to** *Frontier*
9   **return** $\mathit{Base}(\bar{v}, (p, q), (g, h))$

Figure 6. Fusion $A \otimes B$ of BSTs A and B with $o_A = \iota_B$. Definition of $\textsc{Run}$ is given in Figure 7.

Observe how the procedure $\textsc{Fuse}_\delta$ uses γ on lines 5–7: γ is included as a conjunct in every solver call to **SAT** and every recursive call to $\textsc{Fuse}_\delta$. 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 $\textsc{Fuse}_\delta$ 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 $\textsc{Fuse}_\delta$ and $\textsc{Step}$ use the parameter γ in a way such that γ is included as a conjunct in (i) each call to **SAT**, and (ii) each γ argument in recursive calls. Furthermore when $\textsc{Fuse}_\delta$ calls $\textsc{Step}$ on line 11 it passes its γ as an argument. Therefore, each call to $\textsc{Fuse}_\delta$ 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.

RUN$(\gamma, \bar{v}, q, s)$: $(\mathcal{T}(\iota\times\rho\to\mathtt{bool}) \times [\mathcal{T}(\iota\times\rho\to\iota_B)] \times Q_B \times \mathcal{T}(\iota\times\rho\to\rho_B)) \to \mathcal{R}(\iota\times\rho, o, Q_B, \rho_B)$

match $\bar{v}$
    case $[]$: **return** *Base*$([], q, s)$
    case $[v|\mathit{rest}]$: **return** STEP$(\gamma, v, \mathit{rest}, (\delta_B\ q), s)$

STEP$(\gamma, v, \mathit{rest}, R, s)$

let $\theta = \{s/\mathtt{r}{:}\rho_B, v/\mathtt{x}{:}\iota_B\}$
match R
    case *Undef*: **return** *Undef*
    case *Ite*(φ, R_1, R_2):
        let R_1' = **if SAT**$(\gamma \wedge (\varphi\theta))$
                    STEP$(\gamma \wedge (\varphi\theta), v, \mathit{rest}, R_1, s)$
                else *Undef*
        let R_2' = **if SAT**$(\gamma \wedge \neg(\varphi\theta))$
                    STEP$(\gamma \wedge \neg(\varphi\theta), v, \mathit{rest}, R_2, s)$
                else *Undef*
        if SAT$(\gamma \wedge R_1' \neq R_2')$
            return *Ite*$(\varphi\theta, R_1', R_2')$
        else return R_1'
    case *Base*$(\bar{u}, q, g)$:
        return CONCAT$(\bar{u}\theta, \text{RUN}(\gamma, \mathit{rest}, q, g\theta))$

CONCAT$(\bar{u}, R)$

match R
    case *Undef*: **return** *Undef*
    case *Ite*(φ, R_1, R_2):
        return *Ite*$(\varphi, \text{CONCAT}(\bar{u}, R_1), \text{CONCAT}(\bar{u}, R_2))$
    case *Base*$(\bar{v}, q, h)$: **return** *Base*$(\bar{u} + \bar{v}, q, h)$

Figure 7. Step composition of B over a list of symbolic inputs $\bar{v}$ in the context γ.

The fusion procedure for $\$_{A\otimes B}$ is omitted from the presentation, but is similar to the construction of $\delta_{A\otimes B}$. Elements of Q that only lead to non-final control states (control states that only have the *Undef* finalizer) are also removed as "dead-ends" using the standard dead-end elimination algorithm for finite state automata [26].

Theorem 3.1. $[\![A \otimes B]\!] = [\![A]\!] \circ [\![B]\!]$

The main intuition for the proof is that STEP implements a symbolic version of a single step of $\oplus$ and RUN is a symbolic version of a run (of multiple steps) of $\oplus$. Once this connection is proved formally it can be used as a lemma for proving that the transduction semantics given by Equation (1) (Section 2) is preserved by $A \otimes B$.

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 (RBBE) 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.

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:

$$
\begin{aligned}
\mathit{Paths}(\mathit{Undef}) &\stackrel{\text{def}}{=} \emptyset \\
\mathit{Paths}(\mathit{Base}([f_i]_{i=1}^n, g, q)) &\stackrel{\text{def}}{=} \{(\mathtt{true}, g, q)\} \\
\mathit{Paths}(\mathit{Ite}(\varphi, u, v)) &\stackrel{\text{def}}{=} \textstyle\bigcup_{(\psi,g,q)\in \mathit{Paths}(u)}\{(\varphi\wedge\psi, g, q)\} \cup \\
&\phantom{\stackrel{\text{def}}{=}} \textstyle\bigcup_{(\psi,g,q)\in \mathit{Paths}(v)}\{(\neg\varphi\wedge\psi, g, q)\}
\end{aligned}
$$

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

$$
\begin{aligned}
\mathit{Moves}^{\delta}(A) &\stackrel{\text{def}}{=} \textstyle\bigcup_{p\in Q_A}\bigcup_{(\varphi,g,q)\in \mathit{Paths}(\delta_A(p))}\{(p,\varphi,g,q)\} \\
\mathit{Moves}^{\$}(A) &\stackrel{\text{def}}{=} \textstyle\bigcup_{p\in Q_A}\bigcup_{(\varphi,g,q)\in \mathit{Paths}(\$_A(p))}\{(p,\varphi)\}
\end{aligned}
$$

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 8 implements the top-level reachability algorithm. The variable $\mathtt{w} : [\iota_A]$ is used to represent a list of inputs. To check the reachability of a (final) move it calls ISREACHABLE with a $([\iota_A] \times \rho_A)$-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 ISREACHABLE returns `false` then the branch is eliminated by simplifying the corresponding *Ite*(φ, 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 $\perp$ 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 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 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 Ψ from control states to reachability conditions yet to be explored. Initially the control state q_{tgt} is mapped to the predicate φ_{tgt}. Σ maps control states to the predicates that summarize the arguments for which exploration has already been performed or is about to be performed.

ELIMINATE(A)
let U = COMPUTEUNDERAPPROXIMATION(A)
let M = $Moves^{\delta}(A) \cup Moves^{\$}(A) \setminus U$
let $k = |Q_A|$
foreach move (p, φ, g, q) **in** M
  let $\varphi' = (\mathtt{w} \neq \mathtt{[]}) \land \varphi\{\mathtt{Head(w)}/\mathtt{x}\}$
  if ISREACHABLE$(A, p, \varphi', k) =$ `false`
    eliminate the corresponding branch in δ_A
foreach final move (p, φ) **in** M
  let $\varphi' = (\mathtt{w} = \mathtt{[]}) \land \varphi$
  if ISREACHABLE$(A, p, \varphi', k) =$ `false`
    eliminate the corresponding branch in $\$_A$
remove control states with no path from q^0_A

ISREACHABLE$(A, q_{\rm tgt}, \varphi_{\rm tgt}, k)$: $(ST \times Q_A \times \mathcal{T}([\iota_A] \times \rho_A \to \mathtt{bool}) \times \mathtt{int}) \to \mathtt{bool}$
1 **let** $layer = \{q_{\rm tgt}\}$
2 **let** $layer' = \emptyset$
3 **let** $\Psi' = empty = \{q \mapsto \mathtt{false} \mid q \in Q_A\}$
4 **let** $\Sigma = \Psi = empty \uplus \{q_{\rm tgt} \mapsto \varphi_{\rm tgt}\}$
5 **while** $layer \neq \emptyset$
6   **while** $layer \neq \emptyset$
7   **pop** q **from** $layer$
8   **let** $\psi = \Psi[q]$
9   **if** $q = q^0_A \land$ **SAT**$(\psi\{r^0_A/\mathtt{r}\})$
10   **return** `true`
11   **foreach** (p, φ, g, q) **in** $Moves^{\delta}(A)$
12   **if** (φ depends on `r`) **or** (g depends on `x`)
13   **let** $update = g\{\mathtt{Head(w)}/\mathtt{x}\}$
14   **let** $\gamma = (\mathtt{w} \neq \mathtt{[]}) \land \varphi\{\mathtt{Head(w)}/\mathtt{x}\} \land \psi\{\mathtt{Tail(w)}/\mathtt{w}, update/\mathtt{r}\}$
15   **else**
16   **let** $\gamma = \psi\{g\{\_{\iota_A}/\mathtt{x}\}/\mathtt{r}\}$
17   **if SAT**$(\gamma \land \neg\Sigma[p])$
18   **let** $\Sigma[p] = \Sigma[p] \lor \gamma$
19   **let** $\Psi'[p] = \Psi'[p] \lor \gamma$
20   **add** p **to** $layer'$
21   **if** $k = 0 \land layer' \neq \emptyset$
22   **return** $\perp$
23   **let** $k = k - 1$
24   **let** $layer = layer'$
25   **let** $layer' = \emptyset$
26   **let** $\Psi = \Psi'$
27   **let** $\Psi' = empty$
28 **return** `false`

Figure 8. Reachability based branch elimination (RBBE).

Let Δ_A denote the following partial function that extends the transition function $\hat{\delta}_A$ to input lists and omits the output:

$$\begin{array}{rcl} \Delta_A & : & [\iota_A] \times \sigma_A \to \sigma_A \\ \Delta_A([], s) & \stackrel{\text{def}}{=} & s \\ \Delta_A([i|w], s) & \stackrel{\text{def}}{=} & \Delta_A(w, \pi_2(\hat{\delta}_A\, i\, s)) \end{array}$$

A state s is *k-reachable* if there exists $w \in \bigcup_{n\in[0,k]}(\iota_A)^n$ such that $\Delta_A(w, s^0_A) = s$. For example s^0_A is 0-reachable. A state s is *reachable* if it is *k-reachable* for some $k \geq 0$. Given $q \in Q_A$ and an ρ_A-predicate φ, we say that (q, φ) is *(k-)reachable* if there exists a (k-)reachable state (q, r) such that $r \in [\![\varphi]\!]$

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

In this algorithm, Σ enables a crucial *subsumption* checking for predicates (line 17) — if a reachability condition φ for a control state p is subsumed by $\Sigma[p]$, then any search from φ is already covered, so adding φ 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 $\exists\mathtt{w}\,(\Sigma[p])$ (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 φ). When the else case is taken, it means that $\forall\mathtt{w},\mathtt{r}\,(\varphi \Rightarrow \Sigma[p])$ holds, which implies that $\forall\mathtt{r}\,(\exists\mathtt{w}\,\varphi \Rightarrow \exists\mathtt{w}\,\Sigma[p])$ 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<I,O> and override methods named Update and Finish to define δ and \$, 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):

```
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;
  }
}
```

`Update` uses instance variables `i` and `defined` for loop carried state, and `Finish` outputs the final value with C#'s `yield return` keyword. For invalid input an exception is thrown to indicate the input is rejected. ⊠

The code is parsed using the frontend of the Roslyn compiler [6]. The translation to a BST employs a symbolic exploration which captures the state update and outputs represented by each feasible control flow path, while infeasible paths are cut with satisfiability checks using Z3 [18]. The exploration produces an execution tree that corresponds to a BST 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 $\rho_{\texttt{ToInt}} = \texttt{int} \times \texttt{bool}$. Subsequently, the register type is split into $\rho \times \kappa$ where κ is a product of all the types with a small set of values (either `enum` or `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 κ and the new register type becomes ρ. The algorithm is incremental: it starts from the initial values and only considers reachable values of type κ. It is a variant of the ST exploration algorithm discussed in [44, 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.

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

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

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 $\langle o_1, \ldots, o_n \rangle$. 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}(?<int>\d+),(?<bool>\w+),[^\n]*\n)*
```

Here S_1 is `"([^,]*,){2}"` (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

$$\texttt{st:}\mathit{trans}(/\mathit{tag}_1/\mathit{tag}_2/\mathit{tag}_3 \cdots /\mathit{tag}_n)$$

The tag names tag_n specify a path to match in an XML file. *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:

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

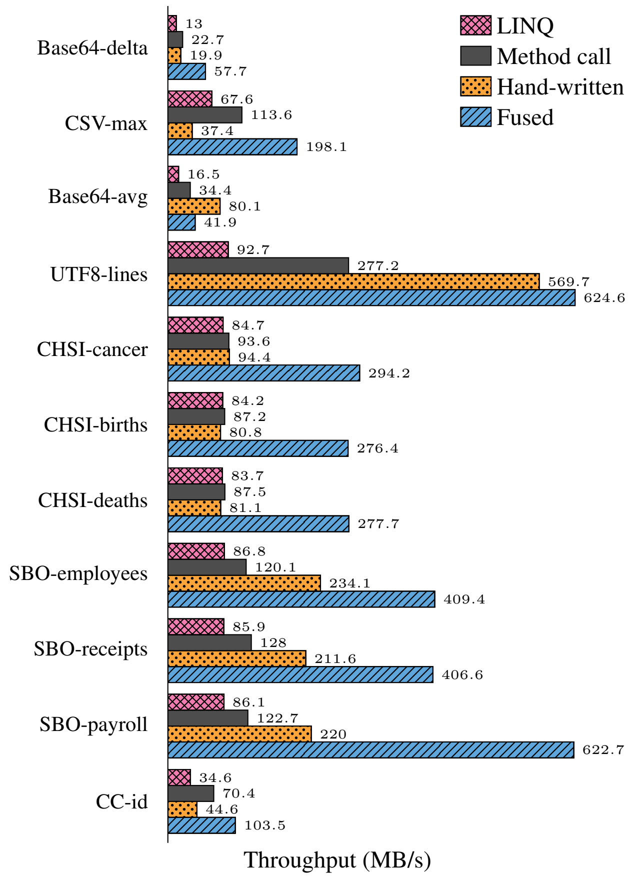


Figure 9. Throughputs for various pipelines

A transducer based on the following XPath expression will parse the populations in the dataset as non-negative integers in decimal notation: `st:int(/cities/city/population)` `int` again maps to the *ToInt* transducer from Figure 4(b). ⊠

6. Evaluation

We have implemented the techniques described above in a tool that translates C# (and our other frontends) into BSTs, 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 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 `int`s 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 `int`s 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 lowest 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

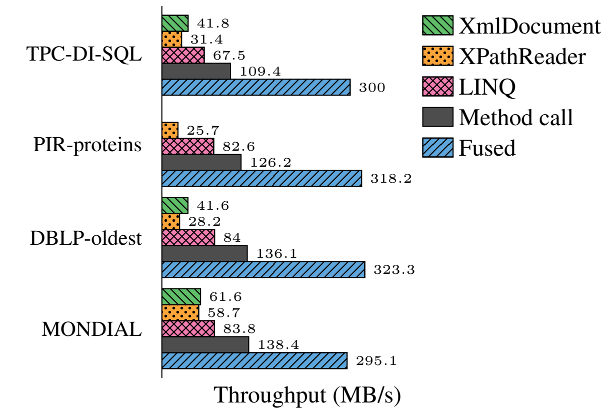


Figure 10. Throughputs for XPath matching pipelines

Pipeline	Rem.	Left	Time	Pipeline	Rem.	Left	Time
Base64-delta	0	77	35s	SBO-employees	7	78	6s
CSV-max	6	65	109s	SBO-receipts	11	117	8s
Base64-avg	0	163	7s	SBO-payroll	10	107	8s
UTF8-lines	1	10	97s	TPC-DI-SQL	238	936	121s
CC-id	1301	5274	63s	PIR-proteins	198	758	37s
CHSI-cancer	113	1134	16s	DBLP-oldest	104	456	8s
CHSI-births	143	1434	16s	MONDIAL	162	662	10s
CHSI-deaths	144	1444	22s				

Figure 11. Branches removed by RBBE and branches left; and total time spent in fusion, RBBE and code generation.

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 `(([^,]*,){5}(?<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. Our fused pipelines are free of such limitations.

The regex comprehension BSTs (Section 5.2) for the Method call pipelines employ C# delegates instead of an `int` and a `switch` due to unacceptable performance with large numbers of cases. This incurs some overhead, but the Method call pipelines are still faster than their LINQ counterparts.

The fused pipelines are significantly faster for all benchmarks, on average 2.7× faster over hand-written ones.

Figure 10 presents throughputs for XML processing scenarios. Four pipelines are compared: **TPC-DI-SQL** uses a dataset that was generated by a tool from the TPC-DI benchmark [35], for which the pipeline extracts ids of accounts from customer records and for each outputs an SQL insert statement; **PIR-proteins** uses a protein dataset from the U.S. based National Biomedical Research Foundation, for which he pipeline extracts the lengths of all proteins in the dataset and outputs the average length; **DBLP-oldest** uses bibliographic information from the Digital Bibliography Library Project, for which the pipeline extracts the publication year of each article and outputs the earliest year; and **MONDIAL** uses a dataset extracted from various geographical Web data sources, for which the pipeline extracts the population of each city in the dataset and outputs the highest population.

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.

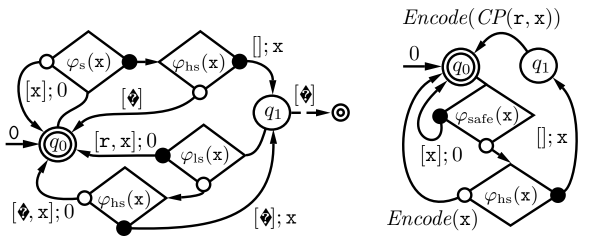
The Fused versions have the highest throughput on all of the XPath benchmarks, with an average speedup of 9× over the streaming XPathReader library. The fact that in the Fused pipelines the XPath matching code is specialized to the query is likely to give it a significant advantage over the XmlDocument 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-



(a) *Rep* (b) *HtmlEncode*

Figure 12. Symbolic transducers for HTML encoding

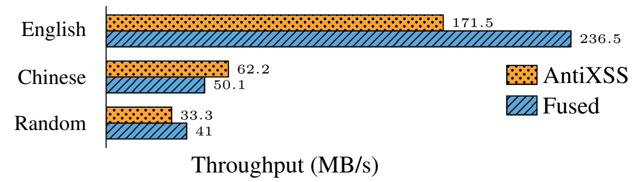


Figure 13. Throughputs for HTML encoding

coded is also *repaired* by replacing any misplaced surrogates by the Unicode replacement character � or $\mathtt{0xFFFD}$ (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 � and *HtmlEncode* assumes that the input is valid Unicode and encodes non-HTML safe characters using the appropriate escape sequences. The input and out types of both *Rep* and *HtmlEncode* is `char`.

The predicate $\varphi_{hs}(x)$ is the *high surrogate* predicate defined by the character range $[\mathtt{0xD800} - \mathtt{0xDBFF}]$ and $\varphi_{ls}(x)$ is the *low surrogate* predicate defined by the character range $[\mathtt{0xDC00} - \mathtt{0xDFFF}]$. 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 $\varphi_s(x)$ is defined as $\varphi_{hs}(x) \vee \varphi_{ls}(x)$ and corresponds to the character range $[\mathtt{0xD800} - \mathtt{0xDFFF}]$. The predicate φ_{safe} is defined as the set:

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

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

$$\begin{array}{l}
\mathit{Encode}(c) = \\
\mathit{Ite}(c = \mathtt{0x22}, \mathit{Base}(\texttt{"\""}, q_0, 0), \\
\mathit{Ite}(c = \mathtt{0x26}, \mathit{Base}(\texttt{"\&"}, q_0, 0), \\
\mathit{Ite}(c = \mathtt{0x3C}, \mathit{Base}(\texttt{"\<"}, q_0, 0), \\
\mathit{Ite}(c = \mathtt{0x3E}, \mathit{Base}(\texttt{"\>"}, q_0, 0), \\
\mathit{Ite}(c < 10, \mathit{Base}(\texttt{"\&\#"} + \mathit{Digits}(c, 1) + \texttt{";"}, q_0, 0), \\
\mathit{Ite}(c < 100, \mathit{Base}(\texttt{"\&\#"} + \mathit{Digits}(c, 2) + \texttt{";"}, q_0, 0), \\
\mathit{Ite}(c < 1000, \mathit{Base}(\texttt{"\&\#"} + \mathit{Digits}(c, 3) + \texttt{";"}, q_0, 0), \\
\mathit{Ite}(c < 10000, \mathit{Base}(\texttt{"\&\#"} + \mathit{Digits}(c, 4) + \texttt{";"}, q_0, 0), \\
\mathit{Ite}(c < 100000, \mathit{Base}(\texttt{"\&\#"} + \mathit{Digits}(c, 5) + \texttt{";"}, q_0, 0), \\
\mathit{Ite}(c < 1000000, \mathit{Base}(\texttt{"\&\#"} + \mathit{Digits}(c, 6) + \texttt{";"}, q_0, 0), \\
\mathit{Base}(\texttt{"\&\#1"} + \mathit{Digits}(c, 6) + \texttt{";"}, q_0, 0))))))))))
\end{array}$$

[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.

where $\mathit{Digits}(c, n)$ is shorthand for a list of expressions that give the n least significant decimal digits as characters. For example $\mathit{Digits}(c, 2) = [((c/10) \mod 10) + \mathtt{0x30}, (c \mod 10) + \mathtt{0x30}]$. *HtmlEncode* instantiates two versions of the pattern as *Encode*$(\mathtt{x})$ and *Encode*$(\mathit{CP}(\mathtt{r}, \mathtt{x}))$, where the function *CP* computes a Unicode *code point* from a high and a low surrogate and is defined as $\mathit{CP}(h, l) = (((h \& \mathtt{0x3FF}) + \mathtt{0x40}) \ll 10) | (l \& \mathtt{0x3FF})$. Note that while both instantiations of *Encode* include some unreachable branches these are removed either by pruning in the fusion or during RBBE. For example, in *Encode*$(\mathit{CP}(\mathtt{r}, \mathtt{x}))$ RBBE eliminates the first eight true branches using the state carried constraint that $\mathit{CP}(\mathtt{r}, \mathtt{x}) \geq \mathtt{0xD0000}$.

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, $\mathit{Th}(A) \cup \mathit{Th}(B)$, 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 ρ into a Cartesian product type $\rho_1 \times \rho_2$ where ρ_1 is a finite type, the primary goal is to reduce register dependency by migrating ρ_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 `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* [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<T>` class. The exact relation to our transduction semantics is that the list comprehension in LINQ corresponds to iterating the step composition operator $\oplus$ (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: $(\hat{\delta}_A\ \texttt{'M'}) \oplus (\hat{\delta}_A\ \texttt{'a'}) \oplus (\hat{\delta}_A\ \texttt{'n'})$. The state of the computation $(\hat{\delta}_A\ \texttt{'M'})$ is threaded through into the computation $(\hat{\delta}_A\ \texttt{'a'})$, 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 `"TWFu"`. This is consistent with the existing semantics of LINQ.

In Figure 3 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()).Concat(Finalize())`, where `Finalize` returns an `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# frontend.

Regular expressions. Our construction of symbolic transducers from regexes is related to the work in [37]. 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* [2]). 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 [15], where an approach for evaluating and reasoning about XPath expressions (extended with regular expressions) based on two-way weak alternating tree automata is presented.

Monads [32, 46] can be used to give an elegant and concise formulation of the semantics of the $\oplus$ and $\otimes$ operators for BSTs that can be summarized as follows. Let $\mathbf{M}^{\sigma}_{\tau}$ be the type $\sigma \to \mathit{Maybe}(\tau \times \sigma)$[8] defined as the tensor product of the state monad and the option monad [27, Theorem 10]. One can now describe $F \oplus G$ using the bind operator $\gg\!=$ of $\mathbf{M}^{\sigma}_{[o]}$ through $F \gg\!= (\lambda x.G \gg\!= (\lambda y.\mathit{return}\ x + y))$. The fuse operator in $A \otimes B$ can be described using the bind operator $\gg\!=$ of $\mathbf{M}^{\sigma_A \times \sigma_B}_{\tau}$ composed with the monad morphisms $\alpha : \mathbf{M}^{\sigma_A}_{\tau} \to \mathbf{M}^{\sigma_A \times \sigma_B}_{\tau}$ and $\beta : \mathbf{M}^{\sigma_B}_{\tau} \to \mathbf{M}^{\sigma_A \times \sigma_B}_{\tau}$, through $\lambda x.((\alpha(A\,x)) \gg\!= (\lambda y.\beta(B\,y)))$. In this discussion a monad based view of a BST A sees A as having the type $[\iota_A] \to \mathbf{M}^{\sigma_A}_{[o_A]}$ and for $A \otimes B$ to be well-defined it is assumed that $o_A = \iota_B$. To the best of our knowledge, this is the first formulation of transducer composition in terms of monads. In particular, the earlier study of transducer composition in [40] using functional programming techniques does not admit such a formulation. We believe this to be the case because the class of transducers studied in [40] is much more abstract.

8. 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 have built a compiler that ingests pipelines written in C# and produces fused code that runs, on average, 3.4× faster than a hand-written baseline and 2.6× faster than code fused with method calls 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.

Acknowledgments

We thank Danel Ahman for helping with the formulation of the $\oplus$ and $\otimes$ operators in terms of monads. We also thank the anonymous reviewers for their helpful and constructive feedback.

[8] We lift the type $\sigma \to \tau$ of a partial function to that of a total function of type $\sigma \to \mathit{Maybe}(\tau)$ by using *option* types.

References

[1] Conduit (Haskell library). `https://github.com/snoyberg/conduit`.

[2] Emoticons. `http://unicode.org/charts/PDF/U1F600.pdf`.

[3] Apache Flink. `https://flink.apache.org/`.

[4] Apache Hadoop. `http://hadoop.apache.org/`.

[5] Highland.js. `http://highlandjs.org/`.

[6] The .NET compiler platform "Roslyn". `https://github.com/dotnet/roslyn`.

[7] Spark Streaming. `http://spark.apache.org/streaming/`.

[8] S. Agrawal, W. Thies, and S. Amarasinghe. Optimizing stream programs using linear state space analysis. In *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'05)*, pages 126–136. ACM, 2005. doi:10.1145/1086297.1086315.

[9] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke. The Stratosphere platform for big data analytics. *The VLDB Journal*, 23(6): 939–964, Dec. 2014. doi:10.1007/s00778-014-0357-y.

[10] R. Alur and J. V. Deshmukh. Nondeterministic streaming string transducers. In *Proceedings of Automata, Languages and Programming: 38th International Colloquium (ICALP 2011)*, volume 6756 of *LNCS*, pages 1–20. Springer, 2011. doi:10.1007/978-3-642-22012-8_1.

[11] R. Alur and P. Černý. Streaming transducers for algorithmic verification of single-pass list-processing programs. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'11)*, pages 599–610. ACM, 2011. doi:10.1145/1926385.1926454.

[12] R. Alur, A. Freilich, and M. Raghothaman. Regular combinators for string transformations. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 9:1–9:10. ACM, 2014. doi:10.1145/2603088.2603151.

[13] R. Alur, L. D'Antoni, and M. Raghothaman. DReX: A declarative language for efficiently evaluating regular string transformations. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'15)*, pages 125–137. ACM, 2015. doi:10.1145/2676726.2676981.

[14] R. Alur and P. Černý. Expressiveness of streaming string transducers. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2010)*, volume 8 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1–12, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.FSTTCS.2010.1.

[15] D. Calvanese, G. Giacomo, M. Lenzerini, and M. Y. Vardi. An automata-theoretic approach to regular XPath. In *Proceedings of the 12th International Symposium on Database Programming Languages (DBPL'09)*, volume 5708 of *LNCS*, pages 18–35. Springer, 2009. doi:10.1007/978-3-642-03793-1_2.

[16] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: From lists to streams to nothing at all. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP'07)*, pages 315–326. ACM, 2007. doi:10.1145/1291151.1291199.

[17] L. D'Antoni and M. Veanes. Extended symbolic finite automata and transducers. *Formal Methods in System Design*, 47 (1):93–119, Aug. 2015. doi:10.1007/s10703-015-0233-4.

[18] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.

[19] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, Jan. 2008. doi:10.1145/1327452.1327492.

[20] D. Debarbieux, O. Gauwin, J. Niehren, T. Sebastian, and M. Zergaoui. Early nested word automata for XPath query answering on XML streams. *Theoretical Computer Science*, 578:100–125, May 2015. doi:10.1016/j.tcs.2015.01.017.

[21] J. Engelfriet and H. J. Hoogeboom. MSO definable string transductions and two-way finite-state transducers. *ACM Transactions on Computational Logic*, 2(2):216–254, Apr. 2001. doi:10.1145/371316.371512.

[22] P. Fradet and S. H. T. Ha. Network fusion. In *Proceedings of Programming Languages and Systems: Second Asian Symposium (APLAS'04)*, volume 3302 of *LNCS*, pages 21–40. Springer, 2004. doi:10.1007/978-3-540-30477-7_3.

[23] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, pages 291–303. ACM, 2002. doi:10.1145/605397.605428.

[24] B. B. Grathwohl, F. Henglein, U. T. Rasmussen, K. A. Søholm, and S. P. Tørholm. Kleenex: Compiling nondeterministic transducers to deterministic streaming transducers. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'16)*, pages 284–297. ACM, 2016. doi:10.1145/2837614.2837647.

[25] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. A catalog of stream processing optimizations. *ACM Computing Surveys*, 46(4):46:1–46:34, Mar. 2014. doi:10.1145/2528412.

[26] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979. ISBN 0321455363.

[27] M. Hyland, G. D. Plotkin, and J. Power. Combining effects: Sum and tensor. *Theoretical Computer Science*, 357(1-3):70–99, July 2006. doi:10.1016/j.tcs.2006.03.013.

[28] G. Mainland, R. Leshchinskiy, and S. Peyton Jones. Exploiting vector instructions with generalized stream fusion. In *Proceedings of the 18th ACM SIGPLAN International Conference*

on Functional Programming (ICFP'13), pages 37–48. ACM, 2013. doi:10.1145/2500365.2500601.

[29] A. Maletti, J. Graehl, M. Hopkins, and K. Knight. The power of extended top-down tree transducers. *SIAM Journal on Computing*, 39(2):410–430, June 2009. doi:10.1137/070699160.

[30] E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling object, relations and XML in the .NET framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD'06)*, pages 706–706. ACM, 2006. doi:10.1145/1142473.1142552.

[31] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS'00)*, pages 11–22. ACM, 2000. doi:10.1145/335168.335171.

[32] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, July 1991. doi:10.1016/0890-5401(91)90052-4.

[33] B. Mozafari, K. Zeng, L. D'antoni, and C. Zaniolo. High-performance complex event processing over hierarchical data. *ACM Transactions on Database Systems*, 38(4):21:1–21:39, Dec. 2013. doi:10.1145/2536779.

[34] D. G. Murray, M. Isard, and Y. Yu. Steno: Automatic optimization of declarative queries. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*, pages 121–131. ACM, 2011. doi:10.1145/1993498.1993513.

[35] M. Poess, T. Rabl, H.-A. Jacobsen, and B. Caufield. TPC-DI: The first industry benchmark for data integration. *Proceedings of the VLDB Endowment*, 7(13):1367–1378, Aug. 2014. doi:10.14778/2733004.2733009.

[36] T. A. Proebsting and S. A. Watterson. Filter fusion. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, pages 119–130. ACM, 1996. doi:10.1145/237721.237760.

[37] Y. Sakuma, Y. Minamide, and A. Voronkov. Translating regular expression matching into transducers. *Journal of Applied Logic*, 10(1):32–51, Mar. 2012. doi:10.1016/j.jal.2011.11.003.

[38] S. Schneider, M. Hirzel, B. Gedik, and K.-L. Wu. Auto-parallelizing stateful distributed streaming applications. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*, pages 53–64. ACM, 2012. doi:10.1145/2370816.2370826.

[39] J. Sermulins, W. Thies, R. Rabbah, and S. Amarasinghe. Cache aware optimization of stream programs. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'05)*, pages 115–126. ACM, 2005. doi:10.1145/1065910.1065927.

[40] O. Shivers and M. Might. Continuations and transducer composition. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06)*, pages 295–307. ACM, 2006. doi:10.1145/1133981.1134016.

[41] J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek. StreamFlex: High-throughput stream programming in Java. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications (OOPSLA'07)*, pages 211–228. ACM, 2007. doi:10.1145/1297027.1297043.

[42] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction (CC'02)*, volume 2304 of *LNCS*, pages 179–196. Springer, 2002. doi:10.1007/3-540-45937-5_14.

[43] M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjorner. Symbolic finite state transducers: Algorithms and applications. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*, pages 137–150. ACM, 2012. doi:10.1145/2103656.2103674.

[44] M. Veanes, T. Mytkowicz, D. Molnar, and B. Livshits. Data-parallel string-manipulating programs. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'15)*, pages 139–152. ACM, 2015. doi:10.1145/2676726.2677014.

[45] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, Jan. 1988. doi:10.1016/0304-3975(90)90147-A.

[46] P. Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming (LFP'90)*, pages 61–78. ACM, 1990. doi:10.1145/91556.91592.

[47] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*, pages 10–10. USENIX Association, 2010. URL `http://www.usenix.org/events/hotcloud10/tech/full_papers/Zaharia.pdf`.