Fusing Effectful Comprehensions *

Olli Saarikivi
Aalto University
olli.saarikivi@aalto.fi

Margus Veanes Todd Mytkowicz
Madan Musuvathi
Microsoft Research
{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×, 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

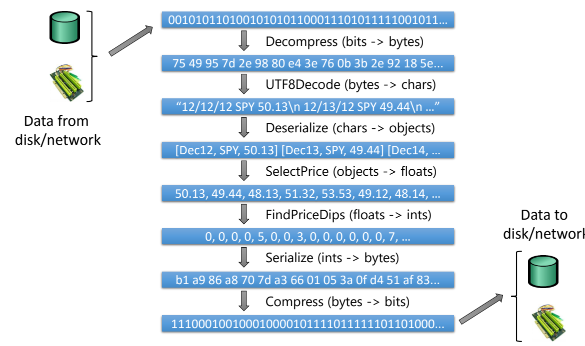
General Terms Algorithms, Languages, Performance, Verification

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 using 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




```
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.

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 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, 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

* Microsoft Research Technical Report no. MSR-TR-2016-55

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]

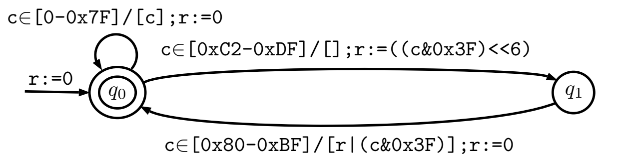


Figure 2. *Utf8Decode* as a flat symbolic transducer.

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 $p \xrightarrow{\texttt{c}\in\alpha/s;\texttt{r:=}g} q$ has the following meaning: if the current state is p and the current byte `c` is in the range α 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,((0xC5&0x3F)<<6)|(0x93&0x3F)]` that is equal to `[0x61,0x153]` or the string `"aœ"`.

ToInt can be illustrated as follows:

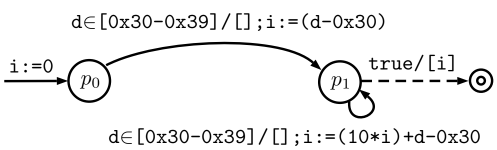


Figure 3. *ToInt* as a flat symbolic transducer.

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 `[i]`. 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 $q_0 \xrightarrow{\texttt{c}\in\texttt{[0-0x7F]/[c];r:=0}} q_0$ with $p_0 \xrightarrow{\texttt{d}\in\texttt{[0x30-0x39]/[];i:=(d-0x30)}} p_1$ produces the fused transition

$$(q_0, p_0) \xrightarrow{\texttt{c}\in\texttt{[0x30-0x39]/[];(r,i):=(0,(c-0x30))}} (q_0, p_1)$$

where the fused register is $(\texttt{r}, \texttt{i})$ and the output `c` 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

$$(q_0, p_1) \xrightarrow{\texttt{c}\in\texttt{[0xC2-0xDF]/[];(r,i):=((c\&0x3F)<<6,i)}} (q_1, p_1)$$

There is one possible fusion of transitions from (q_1, p_1), namely

$$(q_1, p_1) \xrightarrow[\texttt{[];(r,i):=(0,(10*i)+(r|(c\&0x3F))-0x30)}]{\texttt{c}\in\texttt{[0x80-0xBF]}\wedge\texttt{(r|(c\&0x3F))}\in\texttt{[0x30-0x39]}/} (q_0, p_1)$$

However, the state (q_1, p_1) is associated with the register constraint $\exists\texttt{x}\,(\texttt{x} \in \texttt{[0xC2-0xDF]} \wedge \texttt{r} = \texttt{((x\&0x3F)<<6)})$ which together with the guard of the transition from (q_1, p_1) becomes *unsatisfiable*. Thus the transition can be removed from the fused transducer, which in turn implies that the state (q_1, p_1) has become a *dead-end* and the transitions to it can be eliminated, since any execution ending up in (q_1, p_1) is guaranteed to finally reject. Similar reasoning allows us to remove (q_1, p_0) and thus 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.

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-

```
IEnumerable<int> Utf8ToInt(IEnumerable<byte> input) {
  int r = 0; bool multiByte = false;
  var endState = input.SelectMany(c => { // Utf8Decode
    if (!multiByte) {
      if (0 <= c && c <= 0x7F) yield return c;
      else if (0xC2 <= c && c <= 0xDF) {
        r = (c & 0x3F) << 6; multiByte = true;
      } else throw new Exception();
    } else {
      if (0x80 <= c && c <= 0xBF) {
        yield return r | (c & 0x3F); multiByte = false;
      } else throw new Exception();
    }
  }).Aggregate(new { i = 0, defined = false },
               (s, d) => { // ToInt
    if (0x30 <= d && d <= 0x39)
      return new { i = (10 * s.i) + d - 0x30,
                   defined = true };
    else throw new Exception();
  });
  if (!endState.defined) // ToInt's finalizer
    throw new Exception();
  yield return endState.i;
}
```

Figure 4. *Utf8Decode* and *ToInt* in LINQ.[3]

[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 '|' denotes bitwise-or, and the operation '<<k' denotes shift-left by k bits.

[3] We ignore `C#`'s limitation that `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[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 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

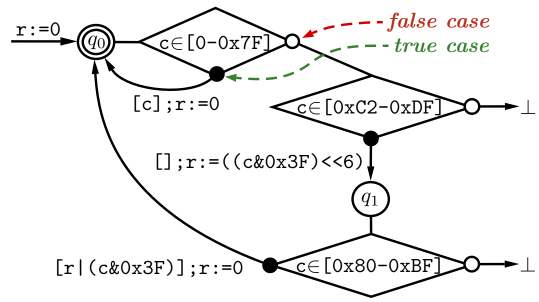


Figure 5. *Utf8Decode* as an ST with branching transitions.

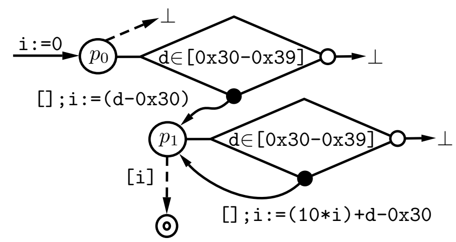


Figure 6. *ToInt* as an ST with branching transitions.

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 subformulas 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 $\perp$ 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 it 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

[4] 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*.

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 [12] 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 background 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 $[t_1, \dots, t_n]$ 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 $\mathtt{first} : \tau \to \tau_1$ and $\mathtt{second} : \tau \to \tau_2$ and a *pairing* function $\langle , \rangle : \tau_1 \to \tau_2 \to \tau_1 \times \tau_2$ with the intended semantics that $[\![\langle t_1, t_2\rangle]\!] = ([\![t_1]\!], [\![t_2]\!])$, and $[\![\mathtt{first}\langle t_1, t_2\rangle]\!] = [\![t_1]\!]$, and $[\![\mathtt{second}\langle t_1, t_2\rangle]\!] = [\![t_2]\!]$. Each type τ denotes a nonempty set and has a default element $\mathtt{default}_\tau$.

The formal definition of a rule is as follows. Given types τ, o, ρ and a finite set (or type) Q, let $\mathcal{R}(\tau, o, Q, \rho)$ denote the smallest set X of *rules* satisfying the following conditions:

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

A rule $r \in \mathcal{R}(\tau, o, Q, \rho)$ denotes a *partial function* $[\![r]\!]$ of type $\tau \to [o] \times Q \times \rho$:[7]

$$[\![\mathit{Undef}]\!]\, v = \bot \text{ for all values } v;$$
$$[\![\mathit{Base}([f_i]_{i=1}^n, q, g)]\!]\, v = ([[\![f_i]\!]\, v]_{i=1}^n, q, [\![g]\!]\, v);$$
$$[\![\mathit{Ite}(\varphi, t, f)]\!]\, v = \begin{cases} [\![t]\!]\, v, & \text{if } [\![\varphi]\!]\, v = \mathtt{true}; \\ [\![f]\!]\, v, & \text{otherwise.} \end{cases}$$

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

- *input type* ι;
- *output type* o;
- *register type* ρ;
- *finite control state set* Q;
- *initial state* (q^0, r^0) of *state type* $\sigma \stackrel{\text{def}}{=} Q \times \rho$, and $s^0 \stackrel{\text{def}}{=} (q^0, r^0)$;
- *transition function* $\delta : Q \to \mathcal{R}(\iota \times \rho, o, Q, \rho)$;
- *finalizer* $\$: Q \to \mathcal{R}(\rho, o, Q, \rho)$;

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 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 adopt 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 $(\mathtt{x} = \varphi)$ the `x` is a literal part of the formula, while φ refers to another formula. Substitution of a variable `y` by a term u in t is denoted $t\{\mathtt{y} \mapsto u\}$.

In *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:

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

The finalizer of *ToInt*, in Figure 6, that is shown as the dashed arrows, 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 `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.

An ST A denotes a *transduction* $[\![A]\!]$ that is a partial function of type $[\iota] \to [o]$. First, we define the following partial semantic 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. Let $\pi_1(a, b) \stackrel{\text{def}}{=} a$. Then

$$[\![A]\!]\, \bar{a} \stackrel{\text{def}}{=} \pi_1(((\hat{\delta}\, a_1) \oplus (\hat{\delta}\, a_2) \oplus \cdots \oplus (\hat{\delta}\, a_k) \oplus \hat{\$})s^0) \qquad (1)$$

where $\oplus$ is a left-associative operator of type

$$(\sigma \to [o] \times \sigma) \times (\sigma \to [o] \times \sigma) \to (\sigma \to [o] \times \sigma)$$

that composes single-input transduction steps into multi-input transduction steps, with the formal definition:

$$F_1 \oplus F_2 \stackrel{\text{def}}{=} \lambda s. \begin{array}{l} \textbf{let } (u_1, s_1) = (F_1\, s) \textbf{ in} \\ \textbf{let } (u_2, s_2) = (F_2\, s_1) \textbf{ in} \\ (u_1 + u_2, s_2) \end{array}$$

where '+' denotes list concatenation. For example, given $\bar{a} = [\mathtt{0xC5}, \mathtt{0x93}]$ and $A = \mathit{Utf8Decode}$, we have that, $s^0 = (q_0, 0)$,

$$\begin{aligned} [\![A]\!]\, \bar{a} &= \pi_1(((\hat{\delta}\, \mathtt{0xC5}) \oplus (\hat{\delta}\, \mathtt{0x93}) \oplus \hat{\$})s^0) \\ &= \pi_1(((\lambda s.([] + [\text{'œ'}], s^0)) \oplus \hat{\$})s^0) \\ &= \pi_1((\lambda s.([] + [\text{'œ'}] + [], s^0))s^0) \\ &= [\text{'œ'}] \end{aligned}$$

We refer to $\oplus$ as *step composition* and revisit it in Section 7. The main intuition about $\oplus$ 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 $\oplus$ corresponds to concatenation, as in a typical `SelectMany` list comprehension in LINQ. If, on the other hand, F_1 and F_2 produce no outputs and only transform the state, then $\oplus$ corresponds to function composition.

3. Fusion of STs

Consider two STs A and B such that $o_A = \iota_B$. We want to *fuse* A and B into a single ST $A \otimes B$ 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

[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 $\tau \to [o] \times Q \times \rho$ to be the type of a total function $\tau \to ([o] \times Q \times \rho)?$ 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, $[\![f]\!](a) \stackrel{\text{def}}{=} b$ if $(a, b) \in [\![f]\!]$ and $[\![f]\!](a) \stackrel{\text{def}}{=} \bot$ 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: $\iota = \iota_A$, $o = o_B$, $\rho = \rho_A \times \rho_B$, $Q \subseteq Q_A \times Q_B$, $r^0 = (r^0_A, r^0_B)$, $q^0 = (q^0_A, q^0_B)$. The goal of the fusion algorithm is to construct $\delta_{A\otimes B}$ and $\$_{A\otimes B}$.

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 $[v_i]^n_{i=1}$ 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

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

if the guard $\gamma(x) = \mathtt{0x100} \le x \le \mathtt{0xFFF}$ 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 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 *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 7 with further helper procedures in Figure 8. Fusion is implemented using depth first search starting from $(p, q) = (q^0_A, q^0_B)$. Only satisfiable parts of composite rules are ever explored. The procedure $\textsc{Fuse}(\gamma, R, q)$ in Figure 7 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 PRODUCT in Figure 7. Elements

$A \otimes B$

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

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

1 **let** $\theta = \{\mathtt{r}{:}\rho_A \mapsto \mathtt{first}(\mathtt{r}{:}\rho)\}$
2 **match** R
3 　　**case** *Undef*: **return** *Undef*
4 　　**case** *Ite*(φ, R_1, R_2):
5 　　　　**let** $R'_1 = \textsc{Fuse}(\gamma \wedge (\varphi\theta), R_1, q)$
6 　　　　**let** $R'_2 = \textsc{Fuse}(\gamma \wedge \neg(\varphi\theta), R_2, q)$
7 　　　　**if SAT**$(\gamma \wedge R'_1 \neq R'_2)$
8 　　　　　　**return** *Ite*$(\varphi\theta, R'_1, R'_2)$
9 　　　　**else return** R'_1
10 　　**case** *Base*$(\bar{v}, p, g)$:
11 　　　　**return** $\textsc{Product}(p, g\theta, \textsc{Run}(\gamma, \bar{v}\theta, q, \mathtt{second}(\mathtt{r}{:}\rho)))$

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

1 **match** R
2 　　**case** *Undef*: **return** *Undef*
3 　　**case** *Ite*(φ, R_1, R_2):
4 　　　　**return** *Ite*$(\varphi, \textsc{Product}(p, g, R_1), \textsc{Product}(p, g, R_2))$
5 　　**case** *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** *Base*$(\bar{v}, (p, q), (g, h))$

Figure 7. Fusion of STs A and B with $o_A = \iota_B$. Definition of RUN is given in Figure 8.

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 8 symbolically execute the step composition operator $\oplus$ for B over the symbolic value list $\bar{v}$ starting from the state (q, s) of B. The satisfiability checks in STEP on lines 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 *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.

Observe how the procedure FUSE uses γ on lines 5–7: γ is included as a conjunct in every solver call to **SAT** and every recursive call to 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 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 FUSE and 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 formula in recursive

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

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

$\textsc{Step}(\gamma, v,$ *rest*$, R, s)$

let $\theta = \{\mathtt{r}{:}\rho_B \mapsto s, \mathtt{x}{:}\iota_B \mapsto v\}$
match R
    case *Undef*: **return** *Undef*
    case *Ite*(φ, R_1, R_2):
        let R'_1 =
            if SAT$(\gamma \wedge (\varphi\theta))$
                $\textsc{Step}(\gamma \wedge (\varphi\theta), v,$ *rest*$, R_1, s)$
            else *Undef*
        let R'_2 =
            if SAT$(\gamma \wedge \neg(\varphi\theta))$
                $\textsc{Step}(\gamma \wedge \neg(\varphi\theta), v,$ *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 $\textsc{Concat}(\bar{u}\theta, \textsc{Run}(\gamma,$ *rest*$, q, g\theta))$

$\textsc{Concat}(\bar{u}, R)$

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

Figure 8. Step composition of B over a list of symbolic inputs $\bar{v}$ in the context γ.

calls.Furthermore when FUSE calls STEP on line 11 it passes its γ as an argument. Therefore, each call to FUSE can use a single solver context incrementally for all satisfiability checks. The structure of pushing and popping the contexts follows the structure of the *Ite*-rules. From our experience using the solver incrementally may decrease the fusion time by an order of magnitude.

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 [16].

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

The proof is omitted for brevity. 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 $A \otimes B$.

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 θ to a sub-term t of $\bar{u}\theta$ in $\textsc{Step}(\gamma, v,$ *rest*$, R,)$ may result in t becoming "constant valued". This can be decided by checking unsatisfiability of the formula $\gamma \wedge \gamma' \wedge t \neq t'$ where the variables in γ' and t' are fresh variants of the variables in γ and t. If the formula is unsatisfiable, then t has a unique value in the context γ, independent of the variables it contains, and can thus be replaced by that value. Such a value can be queried from the solver by considering a model for the formula $\gamma \wedge t = y$ where y is a fresh variable (a model exists since γ 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 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 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 *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:

$$\begin{array}{rcl}
\mathit{Paths} & : & \mathcal{R}(\tau, o, Q, \rho) \to \{(\mathcal{T}(\tau \to \texttt{bool}) \\
 & & \times \mathcal{T}(\tau \to \rho) \times Q)\} \\
\mathit{Paths}(\mathit{Undef}) & \stackrel{\text{def}}{=} & \emptyset \\
\mathit{Paths}(\mathit{Base}([f_i]_{i=1}^{n}, g, q)) & \stackrel{\text{def}}{=} & \{(\texttt{true}, g, q)\} \\
\mathit{Paths}(\mathit{Ite}(\varphi, u, v)) & \stackrel{\text{def}}{=} & \bigcup_{(\psi,g,q)\in \mathit{Paths}(u)} \{(\varphi \wedge \psi, g, q)\} \cup \\
 & & \bigcup_{(\psi,g,q)\in \mathit{Paths} v} \{(\neg\varphi \wedge \psi, g, q)\}
\end{array}$$

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

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

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 $w \in [\iota_A]$ 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

ELIMINATE(A)

1 **let** U = COMPUTEUNDERAPPROXIMATION(A)
2 **let** $M = \mathit{Moves}^{\delta}(A) \cup \mathit{Moves}^{\$}(A) \setminus U$
3 **let** $k = |Q_A|$
4 **foreach** move (p, φ, g, q) **in** M
5   **let** $\varphi' = (\mathtt{w} \neq \mathtt{[]}) \wedge \varphi\{\mathtt{x} \mapsto \mathtt{\mathit{Head}(w)}\}$
6   **if** ISREACHABLE$(A, p, \varphi', k) = \mathtt{false}$
7   eliminate the corresponding branch in δ_A
8 **foreach** final move (p, φ) **in** M
9   **let** $\varphi' = (\mathtt{w} = \mathtt{[]}) \wedge \varphi$
10   **if** ISREACHABLE$(A, p, \varphi', k) = \mathtt{false}$
11   eliminate the corresponding branch in $\$_A$
12 remove control states with no path from q_A^0

Figure 9. Reachability based branch elimination (RBBE).

ISREACHABLE$(A, q_{\mathrm{tgt}}, \varphi_{\mathrm{tgt}}, k)$: $(ST \times Q_A \times \mathcal{T}([\iota_A] \times \rho_A \to \mathtt{bool}) \times \mathtt{int}) \to \mathtt{bool}$

1 **let** $\mathit{layer} = \{q_{\mathrm{tgt}}\}$
2 **let** $\mathit{layer}' = \emptyset$
3 **let** $\Psi' = \mathit{empty} = \{q \mapsto \mathtt{false} \mid q \in Q_A\}$
4 **let** $\Sigma = \Psi = \mathit{empty} \uplus \{q_{\mathrm{tgt}} \mapsto \varphi_{\mathrm{tgt}}\}$
5 **while** $\mathit{layer} \neq \emptyset$
6   **while** $\mathit{layer} \neq \emptyset$
7   **pop** q **from** layer
8   **let** $\psi = \Psi[q]$
9   **if** $q = q_A^0 \wedge$ **SAT**$(\psi\{\mathtt{r} \mapsto r_A^0\})$
10   **return** `true`
11   **foreach** (p, φ, g, q) **in** $\mathit{Moves}^{\delta}(A)$
12   **if** (φ depends on $\mathtt{r}$[8]) **or** (g depends on $\mathtt{x}$[8])
13   **let** $\mathit{update} = g\{\mathtt{x} \mapsto \mathtt{\mathit{Head}(w)}\}$
14   **let** $\gamma = (\mathtt{w} \neq \mathtt{[]}) \wedge \varphi\{\mathtt{x} \mapsto \mathtt{\mathit{Head}(w)}\} \wedge \psi\{\mathtt{w} \mapsto \mathtt{\mathit{Tail}(w)}, \mathtt{r} \mapsto \mathit{update}\}$
15   **else**
16   **let** $\gamma = \psi\{\mathtt{r} \mapsto g\{\mathtt{x} \mapsto \mathtt{default}_{\iota_A}\}\}$
17   **if SAT**$(\gamma \wedge \neg\Sigma[p])$
18   **let** $\Sigma[p] = \Sigma[p] \vee \gamma$
19   **let** $\Psi'[p] = \Psi'[p] \vee \gamma$
20   **add** p **to** layer'
21   **if** $k = 0 \wedge \mathit{layer}' \neq \emptyset$
22   **return** $\bot$
23   **let** $k = k - 1$
24   **let** $\mathit{layer} = \mathit{layer}'$
25   **let** $\mathit{layer}' = \emptyset$
26   **let** $\Psi = \Psi'$
27   **let** $\Psi' = \mathit{empty}$
28 **return** `false`

Figure 10. Checking the reachability of a state predicate.

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 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.

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

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

A state s is *k-reachable (in A)* if there exists $w \in \bigcup_{n \in [0,k]} (\iota_A)^n$ such that $\Delta_A(w, s_A^0) = s$. For example s_A^0 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$(A, q_{\mathrm{tgt}}, \varphi_{\mathrm{tgt}}, k)$ *equals (a)* `true` *then* $(q_{\mathrm{tgt}}, \varphi_{\mathrm{tgt}})$ *is reachable; (b)* `false` *then* $(q_{\mathrm{tgt}}, \varphi_{\mathrm{tgt}})$ *is not reachable; (c)* $\bot$ *then* $(q_{\mathrm{tgt}}, \varphi_{\mathrm{tgt}})$ *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 ψ_{tgt} be the σ_A-predicate $(q = q_{\mathrm{tgt}}) \wedge \varphi_{\mathrm{tgt}}$. The algorithm maintains the following invariant that for all entries $(q \mapsto \varphi) \in \Sigma$ such that $\varphi \neq \mathtt{false}$:

(i) **SAT**(φ) and
(ii) for all $(w, r) \in [\![\varphi]\!]$: $\Delta_A(w, (q, r)) \in [\![\psi_{\mathrm{tgt}}]\!]$

Property (i) follows from the observation that, other than the initial value `false`, only satisfiable predicates are added to $\Sigma[q]$ and that satisfiability remains true under disjunctions. 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 $\exists w\,((w, r_A^0) \in [\![\Sigma[q_A^0]]\!])$. So, by (ii), $\exists w\,(\Delta_A(w, s_A^0) \in [\![\psi_{\mathrm{tgt}}]\!])$. 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 ψ_{tgt} are captured in Σ. This implies that the initial register must be captured in some layer_k predicate $\Psi_k[q^0]$ for there to be a path from the initial state to the target state. The satisfiability test in line 17 ensures that $[\![\varphi]\!] \not\subseteq [\![\Sigma[p]]\!]$. In other words, if the test fails then $[\![\varphi]\!] \subseteq [\![\Sigma[p]]\!]$, 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 $(p, \varphi, g, q) \in \mathit{Moves}^{\delta}(A)$ for which φ is unsatisfiable. Otherwise the definition of γ in line 16 is unsound when φ is unsatisfiable. If φ is satisfiable then $(\exists i\, \varphi\{\mathtt{r} \mapsto \mathtt{default}_{\rho_A}\}) \wedge \psi\{\mathtt{r} \mapsto g\{\mathtt{x} \mapsto \mathtt{default}_{\iota_A}\}\}$ is equivalent to $\psi\{\mathtt{r} \mapsto g\{\mathtt{x} \mapsto \mathtt{default}_{\iota_A}\}\}$.

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-epsilon moves*. Intuitively, the ST is implicitly converted into an εST (ST with input-epsilon moves) although the input-epsilon moves do still count against the bound k. $\square$

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

[8] These checks can be performed with an SMT solver call. For example $\exists \mathtt{i},\mathtt{r},\mathtt{r'}\ (\varphi \neq \varphi\{\mathtt{r} \mapsto \mathtt{r'}\})$ is satisfiable iff φ depends on the register.

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

Figure 11. The `C#` abstract class users extend.

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

$$\forall \mathtt{r}\,(\exists \mathtt{w}\,\varphi \Rightarrow \exists \mathtt{w}\,\Sigma[p]) \tag{2}$$

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 `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 `C#`

We have implemented a translation from a subset of `C#` to STs. Users extend the abstract class in Figure 11, where the `Update` and `Finish` methods respectively define δ 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 *ToInt* transducer from Figure 3:

```
partial class ToInt : Transducer<char,int> {
  bool IsDigit(char c) {
    return 0x30 <= c && c <= 0x39;
  }
  int i = 0; bool defined = false;
  override IEnumerable<int> Update(char d) {
    if (IsDigit(d)) {
      var ones = d - 0x30;
      i = (10 * i) + ones;
    } 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 Roslyn compiler's frontend [4]. The translation to an ST 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 [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 $\rho_{\mathtt{ToInt}} = \mathtt{int} \times \mathtt{bool}$. 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 [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.

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

$$(S_1\texttt{(?<}\mathit{cap}_1\texttt{>}P_1\texttt{)}S_2 \cdots S_n\texttt{(?<}\mathit{cap}_n\texttt{>}P_n\texttt{)}S_{n+1}\texttt{)*}$$

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 $\{\mathit{cap}_i \mapsto A_i\}_{i=1}^{n}$ 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 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 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 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[9] query expressions. Consider an expression X of the form

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

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>
```

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

```
st:int(/cities/city/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<T>` 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

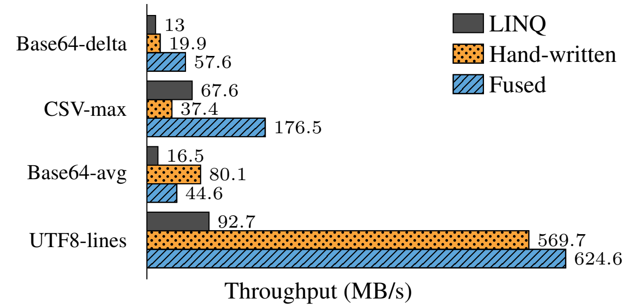


Figure 12. Throughputs for different pipeline versions

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[10] 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 and finds the maximum length of these strings. The output is 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.

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<T>`.

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.

[9] See `https://en.wikipedia.org/wiki/XPath`.

[10] See `https://en.wikipedia.org/wiki/Base64`.

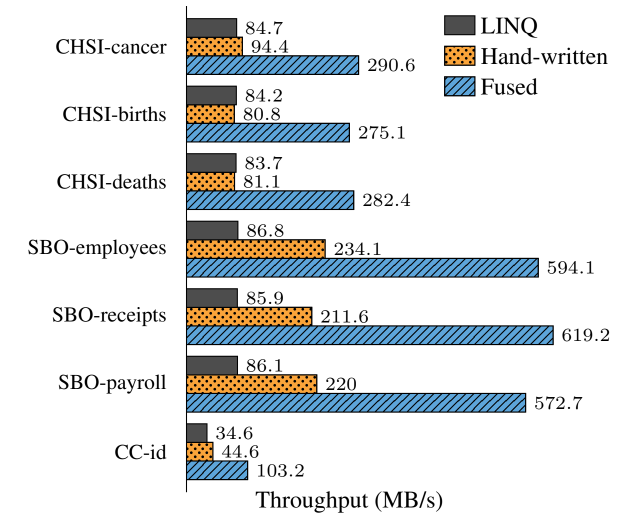


Figure 13. Throughputs for CSV parsing pipelines

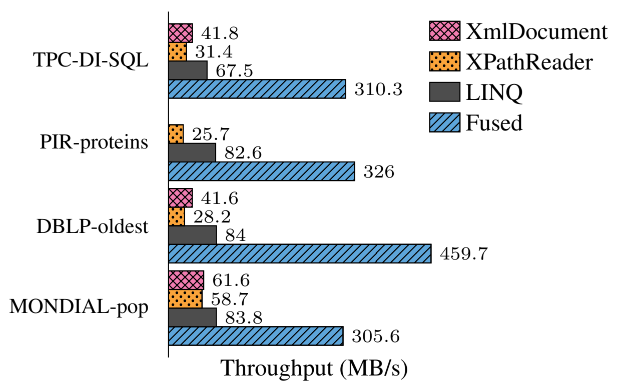


Figure 14. Throughputs for XPath matching pipelines

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 due to their incremental nature.

The fused pipelines are significantly faster for all benchmarks, with the average speedup being over 2.9× over the Hand-written pipelines.

Figure 14 presents throughputs for XML processing scenarios. Four pipelines are compared:

TPC-DI-SQL The dataset was generated by a tool from the TPC-DI benchmark [26]. The pipeline extracts ids of accounts from customer records and for each outputs an SQL insert statement.

Pipeline	Eliminated	Left	Pipeline	Eliminated	Left
Base64-delta	0	77	SBO-employees	7	78
CSV-max	6	65	SBO-receipts	11	117
Base64-avg	0	163	SBO-payroll	10	107
UTF8-lines	1	10	TPC-DI-SQL	238	936
CC-id	1301	5274	PIR-proteins	198	758
CHSI-cancer	113	1134	DBLP-oldest	104	456
CHSI-births	143	1434	MONDIAL-pop	162	662
CHSI-deaths	144	1444			

Figure 15. Branches eliminated by RBBE and branches left.

PIR-proteins The dataset is a protein dataset from the U.S. based National Biomedical Research Foundation. The pipeline extracts the lengths of all proteins in the dataset and outputs the average length.

DBLP-oldest The dataset is bibliographic information from the Digital Bibliography Library Project. The pipeline extracts the publication year of each article and outputs the earliest year.

MONDIAL-pop Mondial is a dataset extracted from various geographical Web data sources. The pipeline extracts the population of each city in the dataset and outputs the highest population.

All of the Fused pipelines in Figure 14 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 due the library running 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. Due to its streaming nature it 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 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 $\oplus$ and $\otimes$ operators in terms of *applicative 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 σ and τ we define $\mathbf{TM}^{\sigma}\tau$ as the type $\sigma \to (\tau \times \sigma)?$[11], which as we will later show is a *transduction monad*. We use the higher order applicative functor [21] operators *pure* and $\star$, defined

[11] ? is the *option type*. We write a wrapped value x as $\lfloor x$ and no value as $\bot$.

as follows:

$$\begin{array}{rcl} \textit{pure} & : & \tau \rightarrow \mathbf{TM}^{\sigma}\tau \\ \textit{pure}\, f & \stackrel{\text{def}}{=} & \lambda s.\!\downharpoonleft\!(f, s) \end{array}$$

$$\begin{array}{rcl} \star & : & \mathbf{TM}^{\sigma}(\tau \rightarrow \tau') \rightarrow \mathbf{TM}^{\sigma}\tau \rightarrow \mathbf{TM}^{\sigma}\tau' \\ F \star X & \stackrel{\text{def}}{=} & \lambda s.\text{let} \downharpoonleft\!(f, s_1) = (F\, s) \text{ in} \\ & & \quad \text{let} \downharpoonleft\!(x, s_2) = (X\, s_1) \text{ in} \\ & & \quad\quad \downharpoonleft\!((f\, x), s_2) \end{array}$$

The intuition is that $\star$ 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 $\oplus$ can also be defined as

$$\begin{array}{rcl} \oplus & : & \mathbf{TM}^{\sigma}\tau \rightarrow \mathbf{TM}^{\sigma}\tau \rightarrow \mathbf{TM}^{\sigma}\tau \\ f \oplus g & \stackrel{\text{def}}{=} & (\textit{pure}\, +) \star f \star g \end{array}$$

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 $\oplus$ is to compose together output results while propagating the effects of the computations "from left to right" as *loop carried state*.

The type of $\oplus$ in this definition (assuming $+$ is concatenation) is

$$(\sigma \rightarrow ([o] \times \sigma)?) \rightarrow (\sigma \rightarrow ([o] \times \sigma)?) \rightarrow (\sigma \rightarrow ([o] \times \sigma)?)$$

Note that the original definition in Section 2 did not wrap the transition functions inside the option type and instead defined them as partial functions. For the functional view we make the functions total by representing rejection with $\bot$.

The *bind* operator for $\mathbf{TM}^{\sigma}\tau$ is

$$\begin{array}{rcl} \gg\!= & : & \mathbf{TM}^{\sigma}\tau_1 \rightarrow (\tau_1 \rightarrow \mathbf{TM}^{\sigma}\tau_2) \rightarrow \mathbf{TM}^{\sigma}\tau_2 \\ F \gg\!= G & \stackrel{\text{def}}{=} & \lambda s.\text{let} \downharpoonleft\!(a, s') = (F\, s) \text{ in } (G\, a\, s') \end{array}$$

We may now view $\mathbf{TM}^{\sigma}\tau$ 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:

$$\begin{array}{rcl} \textit{fuse} & : & ([\iota] \rightarrow \mathbf{TM}^{\sigma_A}[\tau]) \rightarrow ([\tau] \rightarrow \mathbf{TM}^{\sigma_B}[o]) \rightarrow \\ & & ([\iota] \rightarrow \mathbf{TM}^{\sigma_A \times \sigma_B}[o]) \\ \textit{fuse}\ A\ B & \stackrel{\text{def}}{=} & \lambda \bar{x}.\ (A'\, \bar{x}) \gg\!= B' \quad \text{where} \\ A'\, \bar{a} & \stackrel{\text{def}}{=} & \lambda(s_1, s_2).\text{let} \downharpoonleft\!(\bar{b}, s_1') = (A\, \bar{a}\, s_1) \text{ in} \downharpoonleft\!(\bar{b}, (s_1', s_2)) \\ B'\, \bar{b} & \stackrel{\text{def}}{=} & \lambda(s_1', s_2).\text{let} \downharpoonleft\!(\bar{c}, s_2') = (B\, \bar{b}\, s_2) \text{ in} \downharpoonleft\!(\bar{c}, (s_1', s_2')) \end{array}$$

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 $\bar{b}$, 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:

$$(A, s_A^0) \otimes (B, s_B^0) \stackrel{\text{def}}{=} (\textit{fuse}\ A\ B, (s_A^0, s_B^0))$$

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

$$\begin{array}{rcl} \textit{runST} & : & ([\iota] \rightarrow \mathbf{TM}^{\sigma}[o]) \times \sigma \rightarrow [\iota] \rightarrow [o] \\ \textit{runST}\,(A, s) & \stackrel{\text{def}}{=} & \lambda \bar{x}.\text{let} \downharpoonleft\!(\bar{y}, s') = (A\, \bar{x}\, s) \text{ in } \bar{y} \end{array}$$

Effectively, given an ST A, $(\textit{runST}\,(A, s_A^0))$ is its denotation $[\![A]\!]$.

In functional languages the state monad is typically implemented using lazy evaluation and *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 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 [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 functional 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 $\mathbf{TM}^{\sigma}\tau$ is more-or-less the type for the *maybe state monad* parameterized with the state type σ and the output type τ, 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 `IEnumerable<T>` class. The exact relation to the transduction monad 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:

$$(\hat{\delta}_A\ \text{`}\mathtt{M}\text{'}) \oplus (\hat{\delta}_A\ \text{`}\mathtt{a}\text{'}) \oplus (\hat{\delta}_A\ \text{`}\mathtt{n}\text{'})$$

The state of the computation $(\hat{\delta}_A\ \text{`}\mathtt{M}\text{'})$ is threaded through into the computation $(\hat{\delta}_A\ \text{`}\mathtt{a}\text{'})$, 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 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()).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.

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 `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 $\oplus$ 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#` frontend (Section 5.1).

[12] See `http://unicode.org/charts/PDF/U1F600.pdf`

References

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

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

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

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

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

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

[17] S. P. Jones and P. Wadler. Comprehensive comprehensions. In *Proceedings of the ACM SIGPLAN Workshop on Haskell (Haskell'07)*, pages 61–72. ACM, 2007. doi:10.1145/1291201.1291209.

[18] S. Lindley, P. Wadler, and J. Yallop. Idioms are oblivious, arrows are meticulous, monads are promiscuous. In *Proceedings of the Second Workshop on Mathematically Structured Functional Programming (MSFP'08)*, volume 229 of *ENTCS*, pages 97–117. Elsevier, 2011. doi:10.1016/j.entcs.2011.02.018.

[19] 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.

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

[21] C. Mcbride and R. Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, Jan. 2008. doi:10.1017/S0956796807006326.

[22] 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.

[23] 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.

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] M. Veanes, P. d. Halleux, and N. Tillmann. Rex: Symbolic regular expression explorer. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation (ICST'10)*, pages 498–507. IEEE Computer Society, 2010. doi:10.1109/ICST.2010.15.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] P. Wadler. Monads for functional programming. In *Advanced Functional Programming: First International Spring School on Advanced Functional Programming Techniques, Tutorial Text*, volume 925 of *LNCS*, pages 24–52. Springer, 1995. doi:10.1007/3-540-59451-5_2.

[38] 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.