Kalpa Publications in Computing
Volume 1, 2017, Pages 86–99
LPAR-21S: IWIL Workshop
and LPAR Short Presentations

Translating C# to Branching Symbolic Transducers

Olli Saarikivi[1] and Margus Veanes[2]

[1] Helsinki Institute for Information Technology HIIT
Aalto University
olli.saarikivi@aalto.fi
[2] Microsoft Research
margus@microsoft.com

Abstract

The paper looks at tooling aspects of transforming C# programs into symbolic transducers with branching rules (BSTs). The latter are used for describing list comprehensions that incorporate loop-carried state. One concrete application is log analysis where input streams of data are transformed into output streams of data via intermediate pipelines of transducers. The paper presents algorithms for translating C# to BSTs, and for exposing control state in BSTs.

1 Introduction

This paper discusses some of the algorithmic support underlying the tool introduced in [14] that implements *effectful comprehensions* and introduces the notion of *symbolic transducers with branching rules* (BSTs). Effectful comprehensions provide an elegant way to describe list comprehensions that incorporate loop-carried state. As a motivation, consider the problem of analyzing logs. The log on the disk is compressed, and thus the user has to first decompress the input stream of bits into bytes. Then the bytes are decoded into characters, and finally sequences of characters are deserialized or parsed into objects in a higher-level language such as C#. Such processing from input stream of bits to output stream of bits with intermediate layers of objects is not uncommon today [8, 1, 19, 5], and applying fusion to such pipelines can be beneficial [13, 18]. In order for such fusion techniques to be widely applicable to real world programs there must be an accessible way to specify effectful comprehensions.

While efficient fusion of transducers is important and improves efficiency akin to filter fusion [13] and deforestation [18], so is the aspect of *transforming* C# programs (that are used in the frontend) into transducers (that are used in the backend). This latter aspect and the underlying tool support and algorithms used for that is the primary focus of this paper. We present a C# interface for specifying effectful comprehensions that encapsulates state usage. The interface is similar to ones found in existing streaming libraries. We describe the algorithms that are used to translate programs that implement this interface into symbolic transducers. There are two levels of transformations. First we show how we translate C# programs into BSTs and then how we further transform the generated BSTs to expose control states, by eliminating register dependencies, where we study partial and full register exploration algorithms for BSTs.

2 Branching Symbolic Transducers

We here formally define branching symbolic transducers or *BST*s and give examples of how *BST*s capture behavior of programs. For the background logic of *BST*s we assume a *background structure* that has an effectively enumerable *background universe* $\mathbb{U}$, and is equipped with a language of function and relation symbols with fixed interpretations.

We use τ, ι and o to denote types, and we write $\mathbb{U}_{\boldsymbol{\tau}}$ for the corresponding sub-universe of elements of type τ. The Boolean type is `bool`, with $\mathbb{U}_{\texttt{bool}} = \{\texttt{true}, \texttt{false}\}$, the integer type is `int`, and the type of k-bit bit-vectors is `bvk`. The Cartesian product type of types ι and o is $\iota \times o$. We use $\langle \cdot, \ldots, \cdot \rangle : \tau_1 \to \cdots \to \tau_n \to \tau_1 \times \cdots \times \tau_n$ as constructors for Cartesian product (i.e. tuple) types. For projecting the nth element of a Cartesian product term x we use $\pi_n(x)$. The type ι^* is the type for finite sequences of elements of type ι. The universe $\mathbb{U}_{(\boldsymbol{\iota}^*)}$ is the Kleene closure $(\mathbb{U}_{\boldsymbol{\iota}})^*$ of the universe $\mathbb{U}_{\boldsymbol{\iota}}$. We also write type $\iota^{\leq k}$ as a semantic subtype of ι^* of sequences of elements of length *at most* $k \geq 0$.

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

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

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

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

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

- r is the rule *Undef*.
- r is a rule $Base(f\!:\!\iota \to o)$.
- r is a rule $Ite(\varphi\!:\!\iota \to \texttt{bool}, r_1\!:\!\iota \to o, r_2\!:\!\iota \to o)$.

A rule $r\!:\!\iota \to o$ represents a function $[\![r]\!]$ from $\mathbb{U}_{\boldsymbol{\iota}}$ to $\mathbb{U}_{\boldsymbol{o}} \cup \{\bot\}$ For all $a \in \mathbb{U}_{\boldsymbol{\iota}}$:

$$\begin{array}{rcl} [\![\mathit{Undef}]\!](a) & \stackrel{\text{def}}{=} & \bot \\ [\![\mathit{Base}(f)]\!](a) & \stackrel{\text{def}}{=} & [\![f]\!](a) \\ [\![\mathit{Ite}(\varphi, r_1, r_2)]\!](a) & \stackrel{\text{def}}{=} & \begin{cases} [\![r_1]\!](a), & \text{if } a \in [\![\varphi]\!]; \\ [\![r_2]\!](a), & \text{otherwise.} \end{cases} \end{array}$$

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

Definition 1. A *Symbolic Branching Transducer* or *BST* A with *input* type ι, *output* type o and *state* type τ is a tuple (q^0, R, F), where

- $q^0 \in \mathbb{U}_{\boldsymbol{\tau}}$ is the *initial state*;
- R is an *input rule* of type $(\iota \times \tau) \to (o^{\leq k} \times \tau)$, for some $k \geq 0$;
- F is a *final rule* of type $\tau \to o^{\leq k}$, for some $k \geq 0$.

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

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

We write $A^{\sigma/\gamma;\tau}$ to indicate the input/output types σ/γ and the state type τ of a *BST* A. In the following we use the abbreviations $\Sigma = \mathbb{U}_{\boldsymbol{\iota}}$, $\Gamma = \mathbb{U}_{\boldsymbol{o}}$ and $Q = \mathbb{U}_{\boldsymbol{\tau}}$.

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

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

Definition 2. The *transduction* of a *BST* A, $\mathscr{T}_A$, is a function from Σ^* to $\Gamma^* \cup \{\perp\}$:

$$\mathscr{T}_A(\boldsymbol{a}) \stackrel{\text{def}}{=} \begin{cases} \boldsymbol{b}\cdot\boldsymbol{c} & \text{if } \exists\, q \in Q,\ \boldsymbol{b},\boldsymbol{c} \in \Gamma^* \text{ such that } (q^0_A \xrightarrow{\boldsymbol{a}/\boldsymbol{b}}\!\!\!\!\rightarrow_A q \xrightarrow{/\boldsymbol{c}}_A) \\ \perp & \text{otherwise} \end{cases}$$

⊠

*BST*s are inherently deterministic and single-valued as rules are functions and according to Definition 2 all of the input is consumed before the final rule is applied.

The following example illustrates the use of *BST*s on a typical string transformation scenario and illustrates the fragment of C# that we use for defining *BST*s in this paper.

Example 2.1. The C# program in Figure 1 corresponds to a *BST* that decodes certain occurrences of pairs of digits between `5` and `9` by their corresponding ASCII letters. For example `DecodeDigitPairs("a77")` is `"aM"`.

Let f be the λ-term $\lambda(x,y).((10*(y-48))+(x-48))$ and let φ be the predicate $\lambda(x,y).(\texttt{'5'} \leq x \leq \texttt{'9'})$. The tree of `if else` statements in the `Update` method maps directly to the following input rule where we lift the λ-prefix to be in the front:

$$\begin{aligned}\lambda(x,y).\mathit{Ite}(y=0,\quad & \mathit{Ite}(\varphi(x,y), \mathit{Base}([], x), \mathit{Base}([x], y)),\\ & \mathit{Ite}(\varphi(x,y), \mathit{Base}([f(x,y)], 0), \mathit{Base}([x], y)))\end{aligned}$$

```
partial class DecodeDigitPairs : Transducer<char,char> {
  char prev = 0;
  public override IEnumerable<char> Update(char x) {
    if (prev == 0) {    // no previous digit was recorded
      if ('5' <= x && x <= '9') {
        prev = x;       // store the digit
      } else {
        yield return x; // output directly
      }
    } else {            // prev != 0 so prev is the previous digit
      if ('5' <= x && x <= '9') {
        yield return (char)((10*(prev-48))+(x-48));
        prev = 0;
      } else {
        yield return x; // output directly
      }
    }
  }
  public override IEnumerable<char> Finish() {
    if (prev != 0) yield return prev;
  }
}
```

Figure 1: Sample C# transducer.

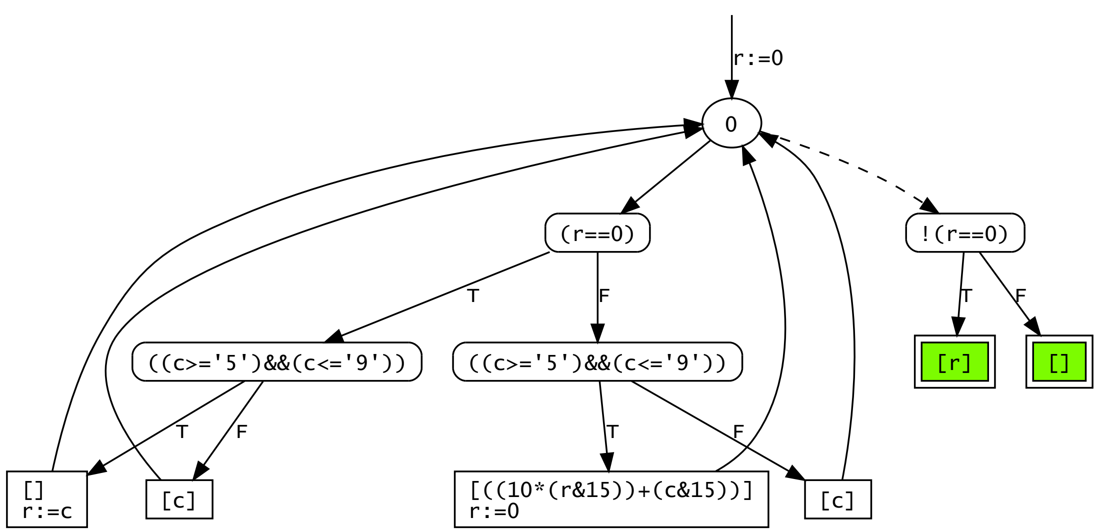

Figure 2: Depiction of the *BST* in Figure 1. Dashed arrows correspond to final rules. Oval nodes correspond to branch conditions and rectangular nodes correspond to basic rules.

Similarly, the final rule corresponds to the `Finish` method:

$$\lambda y.Ite(y \neq 0, [y], [])$$

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

Figure 3: The abstract class for transducers that `DecodeDigitPairs` Figure 1 extends.

The graphical illustration of the *BST* for `DecodeDigitPairs` is shown in Figure 2. All graphs in the paper are produced automatically from our analysis framework. ⊠

3 C# to BSTs

In this section we present a procedure for translating a transducer specified in C# (see Figure 1), into an equivalent BST (see Figure 2). The C# code is in the form of a `class` implementing the `Transducer<I,O>` interface in Figure 3. The code must:

- produce output via `yield return` statement,
- not reference variables apart from its parameters, local variables and non-`static` fields,
- not call functions outside the class or any non-pure functions (purity is checked).

To translate C# into BSTs the procedure has to be able to lift types and operations on them in C# into those in a background logic for a BST. If the background logic is defined by what is supported in Z3 the lifting could for example lift:

- `int` into 32-bit bitvectors,
- `bool` into the Boolean type,
- `struct` into tuples (or algebraic datatypes) of the component types,

The following explanation assumes that an appropriate lifting is available, but does not go into details.

We write a function that maps $a_1, \ldots, a_n$ to $b_1, \ldots, b_n$ as $\{a_1 \mapsto b_1, \ldots, a_n \mapsto b_n\}$. Given a function f we write the modified function that maps a to b as $f\{a \mapsto b\}$.

The entry point to the procedure is ToBST in Figure 4. Given a program P it constructs *control flow graphs* (CFGs) [6] for the `Update` and `Finish` methods (using GetCFG) and calls ToRule$_R$ (or ToRule$_F$) to translate the C# code into rules for a BST. The state type τ for the final BST is a Cartesian product type of the lifted field types. ToBST also maps (see line 4) the fields of P into an initial variable mapping, where each field is mapped to a term that projects the appropriate value out of the state. This initial mapping represents an identity transformation on the state. To construct the initial state q_0, ToBST lifts the initial values of the fields of P into the background logic and constructs the appropriate product from them. Finally, ToBST returns a BST with q_0, where the input and final rules implement the `Update` and `Finish` methods respectively.

The main procedure for translating C# into input rules is ToRule$_R$ in Figure 5. In addition to a *basic block* B from a CFG and the current variable assignment *vars*, each call to ToRule$_R$ is passed a *path constraint* φ_{path}. As ToRule$_R$ recursively calls itself to explore further basic blocks, the recursion structure will correspond to the tree of possible executions of the current

$\textsc{ToBST}(P)$

$(\mathit{in}, \mathit{out}) := ($`I`,`O`$)$ as defined by P's base class `Transducer<I,O>`
$B_{\mathit{update}} := \textsc{GetCFG}($method `IEnumerable<`$\mathit{out}$`> Update(`$\mathit{in}$` input)` from $P)$
$B_{\mathit{finish}} := \textsc{GetCFG}($method `IEnumerable<`$\mathit{out}$`> Finish()` from $P)$
$\mathit{vars} := \{i$th field of $P \mapsto \lambda(x,y).\pi_i(y) \mid i = 1 \ldots m$, where m is the number of fields in $P\}$
$R := \textsc{ToRule}_R(P, B_{\mathit{update}}, \mathit{vars}\{$`input` $\mapsto \lambda(x,y).x\}, [], \texttt{true})$
$F := \textsc{ToRule}_F(P, B_{\mathit{finish}}, \mathit{vars}, [], \texttt{true})$
$q_0 := \langle z$ lifted into the background logic $\mid z \in$ the fields of $P\rangle$
return (q_0, R, F)

Figure 4: Translation of C# into a BST

$\textsc{ToRule}_R(P, B, \mathit{vars}, \bar{u}, \varphi_{\mathit{path}})$

1 **if** $\neg\textsc{IsSat}(\varphi_{\mathit{path}})$
2   **return** $\bot$
3 **for** each *stmt* in the statements of B
4     $(\mathit{vars}, \bar{w}) := \textsc{EvalStmt}(\mathit{stmt}, \mathit{vars})$
5     $\bar{u} := \bar{u} + \bar{w}$
6 **match** the terminator of B
7   **case** `if (`cond`)` B_{true} `else` B_{false}:
8         $(\mathit{vars}, \bar{u}, \varphi_{\mathit{cond}}) := \textsc{Eval}(\mathit{cond}, \mathit{vars})$
9         $R_{\mathit{true}} := \textsc{ToRule}_R(P, B_{\mathit{true}}, \mathit{vars}, \bar{u}, \varphi_{\mathit{path}} \wedge \varphi_{\mathit{cond}})$
10         $R_{\mathit{false}} := \textsc{ToRule}_R(P, B_{\mathit{false}}, \mathit{vars}, \bar{u}, \varphi_{\mathit{path}} \wedge \neg\varphi_{\mathit{cond}})$
11   **if** $R_{\mathit{true}} = \bot$ **return** R_{false}
12   **elseif** $R_{\mathit{false}} = \bot$ **return** R_{true}
13   **else return** $\mathit{Ite}(\varphi_{\mathit{cond}}, R_{\mathit{true}}, R_{\mathit{false}})$
14   **case** `goto` B_{target}:
15   **return** $\textsc{ToRule}_R(P, B_{\mathit{target}}, \mathit{vars}, \bar{u}, \varphi_{\mathit{path}})$
16   **case** `yield break`:
17   **return** $\mathit{Base}(\lambda(x,y).\langle\bar{u}, \langle \mathit{vars}(f) \mid \mathit{field} \in \text{f of } P\rangle\rangle)$
18   **case** `throw`:
19   **return** *Undef*

Figure 5: Translation of a CFG into a rule

CFG. In each recursive call φ_{path} is the conjunction of branch constraints for the corresponding execution path. On line 1 satisfiability of φ_{path} is checked using an SMT solver to prune paths from the rule being constructed.

For obtaining final rules $\textsc{ToRule}_F$, which is not shown, is used. The only difference to $\textsc{ToRule}_R$ is that on line 17 the returned base rule does not specify a state update.

The basic block B consists of a list of non-branching statements followed by a *terminator*. $\textsc{ToRule}_R$ executes the statements in the basic block by calling $\textsc{EvalStmt}$, which returns an updated variable assignment and a list of yields. The code on lines 6–19 that pattern matches on the terminator of B handles the different types of control flow:

`if else` causes the exploration to branch into two recursive ToRule_R calls. The path constraint of the recursive calls to ToRule_R may end up being unsatisfiable, in which case the rule simplifies to the one from the other branch instead of an *Ite*-rule.

`goto` has one target and as such the recursive call is a tail call, i.e., for efficiency this call could just set the parameters in the current call and jump to the beginning of the procedure.

`yield break` terminates the current execution path. A *Base*-rule is constructed from the list of yields along the path and the state update as defined by the values in *vars* for the fields of P.

`throw` results in an *Undef*-rule, indicating that the input was rejected.

This process of exploring an execution tree of the CFG while pruning unsatisfiable branches also supports looping constructs in C#, since these translate to a CFG with `if else` and `goto` terminators. As long as the loops terminate for all inputs and states (also unreachable ones), ToRule_R will also terminate. However, it is easy to use loops to specify very large rules, in which case ToRule_R may run out of memory or appear to hang.

Example 3.1. The following transducer formats unsigned 32-bit integers in decimal notation.

```
partial class FormatInt : Transducer<uint,char> {
    public override IEnumerable<char> Update(uint x) {
        int digits = 10;
        int divisor = 1000000000;
        while (divisor > 1 && divisor > x) {
            divisor /= 10;
            --digits;
        }
        for (int i = 0; i < digits; ++i) {
            yield return (char)((x / divisor) % 10 + '0');
            divisor /= 10;
        }
        yield return '\n';
    }
}
```

The input rule for this transducer has ten different *Base*-rules (one for each number of digits). A transducer written in C# without loops would be larger and the code would have more repetition. ⊠

The procedure for translating C# statements is EvalStmt in Figure 6, which directly handles:

- `yield return` statements by returning the expression evaluated with Eval in the list of yields, and
- local variable definitions by returning an updated *vars*.

For statements which consist of just a C# expressions it calls Eval, which interprets the expression in the context of the current *vars* and returns an equivalent expression in the background logic. Since expressions may have side effects, Eval also returns an updated version of *vars*.

The handling of function calls on lines 6–13 of Figure 6 calls for further explanation. To evaluate the function f its CFG is created and interpreted by a call to the procedure ToExpr

```
EvalStmt(stmt, vars)
match stmt
    case yield return a:
        (vars, _, v_a) := Eval(a, vars)
        return (vars, [v_a])
    case var a = b:
        (vars, result) := Eval(b, vars)
        return (vars{a ↦ result}, ū)
    default:
        // The statement is an expression
        (vars, _) := Eval(stmt, vars)
        return (vars, [])
```

```
Eval(expr, vars)
result = ⊥
match expr
    case a = b:
        (vars, result) := Eval(b, vars)
        vars := vars{a ↦ result}
    case f(a_1,...,a_n):
        B_f := GetCFG(f)
        vars_f := {}
        for i = 1...n
            (vars, v) := Eval(a_i, vars)
            vars_f := vars_f{ith parameter of f ↦ v}
        result := ToExpr(B_f, vars_f, true)
        vars := vars_n
    case var if var is a variable:
        result := vars(var)
    case l if l is a literal:
        result := l lifted into the background logic
    case a + b:
        plus := + lifted into the background logic
        (vars, v_a) := Eval(a, vars)
        (vars, v_b) := Eval(b, vars)
        result := plus(v_a, v_b)
    // Cases for other binary and unary operations omitted
return (vars, result)
```

Figure 6: Translation of C# expressions into a background logic

in Figure 7. The arguments are interpreted left-to-right (applying any side effects in *vars*) and an initial variable assignment $vars_f$ mapping the parameters of f to the arguments is created.

ToExpr is largely similar to ToRule_R in Figure 5, except that it constructs a formula in the background logic instead of rules. The main differences are in the terminators supported, ToExpr($B, vars, \varphi_{path}$)

1 **if** $\neg$IsSat(φ_{path})
2   **return** $\bot$
3 **for** each *stmt* in the statements of B
4     $(vars, _) := $ Eval($stmt, vars$)
5 **match** the terminator of B
6   **case** `if (`*cond*`)` B_{true} `else` B_{false}:
7         $(vars, _, \varphi_{cond}) := $ Eval($cond, vars$)
8         $e_{true} := $ ToExpr($B, true, vars, \varphi_{path} \wedge \varphi_{cond}$)
9         $e_{false} := $ ToExpr($B, false, vars, \varphi_{path} \wedge \neg\varphi_{cond}$)
10   **if** $e_{true} = \bot$ **return** e_{false}
11   **elseif** $e_{false} = \bot$ **return** e_{true}
12   **else return** `ite`($\varphi_{cond}, e_{true}, e_{false}$)
13   **case** `goto` B_{target}:
14   **return** ToExpr($B_{target}, vars, \varphi_{path}$)
15   **case** `return` a:
16         $(_, result) := $ Eval($a, vars$)
17   **return** *result*

Figure 7: Translation of a CFG for a pure function into an expression in a background logic

with functions passed to ToExpr having to end in a `return` statement and `yield` statements not being supported. Also note that on line 12 the `ite` being returned is a term in the background logic instead of an *Ite*-rule. As the initial $vars_f$ constructed by Eval includes only the parameters of f, only pure functions are supported by ToExpr.

While the Eval presented here uses ToExpr to inline function calls, ToExpr can also be used to provide background definitions for functions. This would result in more compact terms being created, but would prevent functions from being simplified to the context they are called from.

4 Register to Control State Exploration

In this section we develop an algorithm that allows us to eliminate either all or some of the state registers used in a deterministic *BST* A. In particular, we focus on two, most prominent cases:

- *full* exploration, and
- *Boolean* exploration.

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

Explore($A^{\iota/o;\tau_1\times\tau_2}$)

$p_0 := \pi_1(q_A^0)$
$q_0 := \pi_2(q_A^0)$
$S := \mathit{stack}(q_0)$
$P := \{p_0\}$
$\mathit{Add} \stackrel{\text{def}}{=} \boldsymbol{\lambda} p.$ **if** $p \notin P$ **then** $P := P \cup \{p\};\ \mathit{Push}(S, p)$
$R := \{\mapsto\}$
$F := \{\mapsto\}$
while $S \neq \emptyset$
  $p := \mathit{Pop}(S)$
  $R(p) := \mathit{Expl}(\lambda y\!:\!\tau_2.\mathtt{true}, \mathit{Inst}(\lambda y\!:\!\tau_2.\mathtt{true}, R_A, p), \mathit{Add})$
  $F(p) := \mathit{Expl}(\lambda y\!:\!\tau_2.\mathtt{true}, \mathit{Inst}(\lambda y\!:\!\tau_2.\mathtt{true}, F_A, p), \mathit{Add})$
return (P, p_0, q_0, R, F)

Inst(φ, R, p)

match R
  case *Undef*: **return** *Undef*
  case $\mathit{Base}(f, g)$: **return** $\mathit{Base}(\lambda y.f(p, y), \lambda y.g(p, y))$
  case $\mathit{Ite}(\psi, t, f)$:
    $\varphi_t := \lambda y.\varphi(y) \wedge \psi(p, y)$
    $\varphi_f := \lambda y.\varphi(y) \wedge \neg\psi(p, y)$
    if IsSat(φ_t) **return** Inst(φ_f, f, p)
    elseif IsSat(φ_f) **return** Inst(φ_t, t, p)
    else return $\mathit{Ite}(\lambda y.\psi(p, y), \text{Inst}(\varphi_t, t, p), \text{Inst}(\varphi_f, f, p))$

Expl(φ, R, Add)

match R
  case *Undef*: **return** *Undef*
  case $\mathit{Ite}(\psi, t, f)$: **return** $\mathit{Ite}(\varphi, \text{Expl}(\varphi \wedge \psi, t, \mathit{Add}), \text{Expl}(\varphi \wedge \neg\psi, f, \mathit{Add}))$
  case $\mathit{Base}(f, g)$:
    $\psi := \lambda y\, z\, .\varphi(y) \wedge (z = \pi_1(g(y)))$
    $r := \mathit{Undef}$
    while $\exists M \models \psi$
      $r := \mathit{Ite}(\lambda y.p = \pi_1(g(y)), \mathit{Base}(f, \lambda y.\pi_2(g(y)), p), r)$
      $\psi := \lambda y\, z\, .\psi(y, z) \wedge z \neq z^M$
      $\mathit{Add}(z^M)$
    return r

Figure 8: Exploration algorithm of *BST*s.

equivalent form

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

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

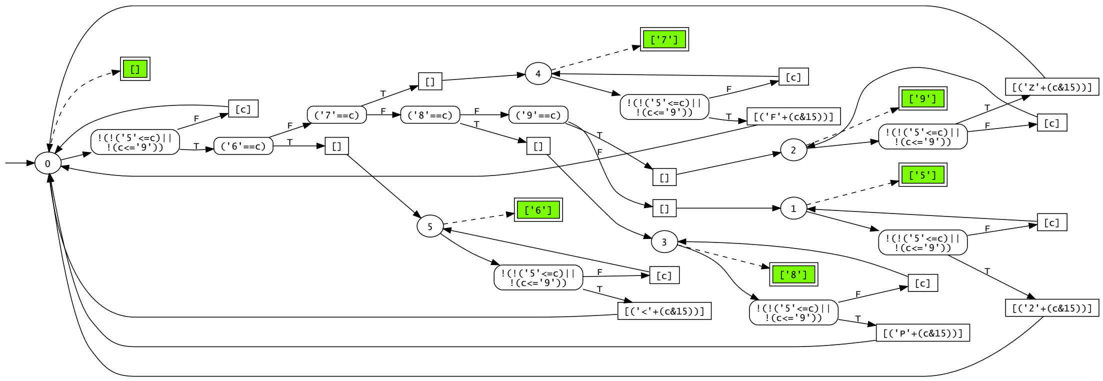

Figure 9: *BST* after full exploration of `DecodeDigitPairs` in Figure 1.

separate from the data state. For example, the *BST* is Example 2.1 is not stateless because the rules depend on the register y.

By *full exploration* of A, we mean a construction of a stateless BST $A^{\mathbf{f}}$ such that $\mathscr{T}_A = \mathscr{T}_{A^{\mathbf{f}}}$, i.e., A and $A^{\mathbf{f}}$ are equivalent. Full exploration is not always possible, because equivalence of stateless *BST*s reduces to equivalence of symbolic finite transducers (SFTs), and equivalence of SFTs is decidable [16] modulo a decidable label theory, while equivalence of *BST*s is undecidable already for very restricted decidable label theories. Even when full exploration is possible, $A^{\mathbf{f}}$ may still be exponentially larger than A.

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

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

Algorithm. The generic exploration algorithm of *BST*s is described in figure 8. The algorithm takes as its input a BST A, and assumes a projection of the state type τ of A into two parts τ_1 and τ_2. We assume, without loss of generality, that $\tau = \tau_1 \times \tau_2$. The algorithm uses an SMT solver to solve satisfiability and to generate models for formulas.

The algorithm generates a new BST by exploring the rules with respect to τ_1, effectively eliminating τ_1, i.e. turning it into an explicit state. In order to avoid special cases, we may always assume that either τ_1 or τ_2 can be unit types $\mathsf{T0}$ ($\mathbb{U}_{\mathtt{T0}} = \{\langle\rangle\}$). Now, full exploration of A corresponds to the case when τ_2 is unit type, and Boolean exploration corresponds to the case when τ_1 is a Cartesian combination of Boolean registers and τ_2 is a Cartesian combination of all the non Boolean registers.

$\textsc{Inst}(\varphi, r, p)$ creates an instance of the rule r with the path condition φ with respect to the fixed register values given by p. For the exception rule this is a no-op. For a basic rule this is a partial instantiation of the yield and update with respect to p, where $\lambda y.f(p,y)$ instantiates the first projection of the state register with the value p. An important point for the rules is that unreachable rule instances are incrementally eliminated by deciding satisfiability of corresponding accumulated path conditions.

$\textsc{Expl}(\varphi, r, \mathit{add})$ is a form of partial exploration of r the with respect to τ_1 or the projection projection function. For the exception rule the operation is a no-op. For an if-then-else rule, the step is a direct propagation of the concretizations of the branches. The core of the computation takes place during the concretization of basic rules.

Theorem 4.1. Let A be a deterministic BST with state type $\tau_1 \times \tau_2$. If $\textsc{Explore}(A)$ terminates then the result is a BST that is equivalent to A and whose state type is τ_2.

We omit the formal proof of the theorem but note that termination of the algorithm depends on two factors: decidability of the background theory, and finiteness of the reachable subset of $\mathbb{U}_{\tau_1}$. The first point is already needed in the $\textsc{Inst}$ procedure that eliminates unsatisfiable branches. The second point is needed both, for termination of construction of r in $\textsc{Expl}$, as well as for guaranteeing that the search stack is bounded in size. A sufficient condition for the second point is when the functions used for computing the first state projection have the *finite-range* property, i.e., when $\mathbb{U}_{\tau_1}$ can be assumed to be finite.

Example 4.1. The BST after full exploration of `DecodeDigitPairs` from Figure 1, is illustrated in Figure 9. The unexplored BST (in Figure 2) has a single control state 0, while the fully explored BST has 6 control states. ⊠

5 Implementation

We have implemented the C# to BSTs and the register exploration algorithms in the Automata library, which is available under the MIT license. A version with our changes can be found at:

https://github.com/OlliSaarikivi/Automata/

The C# frontend is implemented in the `CSharpFrontend` project. A `string` containing a class extending `Transducer<I,O>` can be turned into an instance of `STb<FuncDecl,Expr,Sort>` (a BST with Z3 formulas as its background logic) by calling:

```
Microsoft.Automata.CSharpFrontend.CSharpParser.FromString(Z3Context,string)
```

For Boolean exploration `STb` has an `ExploreBools()` method.

6 Related Work

Symbolic transducers were introduced in flat form in [16] for analysis of string sanitizers with the main focus on *symbolic finite transducers* or SFTs. The paper [14] develops composition algorithms for BSTs. Further work on symbolic transducers has focused on *register exploration* and *input grouping*. 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 [7] and parallelization [17].

Stream processing area has a large body of work [9, 10, 11, 12, 15]. Some libraries for streams provide APIs for expressing stateful operations. The Apache Flink [5] and Spark Streaming [4] 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 [2] are traditional stream libraries, which both provide a way to express stateful operations.

7 Conclusion

The translation of C# into BSTs in Section 3 allows a natural and compact way to specify effectful comprehensions as imperative code. Using a fragment of the host language for specification ensures a seamless integration by obviating impedance mismatches arising from differences in type systems.

The register exploration algorithm in Section 4 exposes control states in the BST, thus allowing the programmer to freely use C#'s native types for state while still permitting efficient application of BST algorithms that leverage control state, such as fusion [14].

References

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

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

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

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

[5] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. The Stratosphere platform for big data analytics. *The VLDB Journal*, 23(6):939–964, December 2014.

[6] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970.

[7] Loris D'antoni and Margus Veanes. Extended symbolic finite automata and transducers. *Formal Methods in System Design*, 47(1):93–119, August 2015.

[8] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[9] Denis Debarbieux, Olivier Gauwin, Joachim Niehren, Tom Sebastian, and Mohamed Zergaoui. Early nested word automata for XPath query answering on XML streams. *Theoretical Computer Science*, 578:100–125, May 2015.

[10] Andreas Maletti, Jonathan Graehl, Mark Hopkins, and Kevin Knight. The power of extended top-down tree transducers. *SIAM J. Comput.*, 39(2):410–430, June 2009.

[11] Tova Milo, Dan Suciu, and Victor 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.

[12] Barzan Mozafari, Kai Zeng, Loris D'antoni, and Carlo Zaniolo. High-performance complex event processing over hierarchical data. *ACM Trans. Database Syst.*, 38(4):21:1–21:39, December 2013.

[13] Todd A. Proebsting and Scott 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.

[14] Olli Saarikivi, Margus Veanes, Todd Mytkowicz, and Madan Musuvathi. Fusing effectful comprehensions. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'17)*. ACM, 2017.

[15] Jesper H. Spring, Jean Privat, Rachid Guerraoui, and Jan 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.

[16] Margus Veanes, Pieter Hooimeijer, Benjamin Livshits, David Molnar, and Nikolaj 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.

[17] Margus Veanes, Todd Mytkowicz, David Molnar, and Benjamin 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.

[18] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, January 1988.

[19] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion 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.