Automatic Program Inversion using Symbolic Transducers

Qinheping Hu
University of Wisconsin-Madison, USA
qhu28@wisc.edu

Loris D'Antoni
University of Wisconsin-Madison, USA
loris@cs.wisc.edu

Abstract

We propose a fully-automated technique for inverting functional programs that operate over lists such as string encoders and decoders. We consider programs that can be modeled using symbolic extended finite transducers (s-EFTs), an expressive model that can describe complex list-manipulating programs while retaining several decidable properties. Concretely, given a program P expressed as an s-EFT, we propose techniques for: 1) checking whether P is injective and, if that is the case, 2) building an s-EFT P^{-1} describing its inverse. We first show that it is undecidable to check whether an s-EFT is injective and propose an algorithm for checking injectivity for a restricted, but a practical class of s-EFTs. We then propose an algorithm for inverting s-EFTs based on the following idea: if an s-EFT is injective, inverting it amounts to inverting all its individual transitions. We leverage recent advances in program synthesis and show that the transition inversion problem can be expressed as an instance of the syntax-guided synthesis framework. Finally, we implement the proposed techniques in a tool called GENIC and show that GENIC can invert 13 out of 14 real complex string encoders and decoders, producing inverse programs that are almost identical to manually written ones.

CCS Concepts **• Theory of computation → Formal languages and automata theory**

Keywords Extended Symbolic Transducers, Program Inversion, GENIC

1. Introduction

Program inversion is an old but everlasting topic in computer science. Already in 1978, Dijkstra was investigating domain-specific techniques for manually inverting simple array-manipulating programs [11]. Besides being intriguing and foundational, the problem of efficiently and automatically producing correct inverse programs has practical applications in many fields of computer science, such as data extraction, transformation, compression, and encryption.

PLDI'17, June 18–23, 2017, Barcelona, Spain
ACM. 978-1-4503-4988-8/17/06...$15.00
http://dx.doi.org/10.1145/3062341.3062345

Consider the task of writing a string encoder such as BASE64 and the corresponding decoder—i.e., two programs that transform a plain text from one format to another. After successfully writing the encoder, the programmer also has to write a decoder and test whether the two programs invert each other. Since these programs are very similar, the programmer is repeating the same work twice and is more likely to introduce mistakes. In fact, mistakes in this type of programs are common and, in the past, buggy string encoders have caused large-scale security vulnerabilities [19].

Inspired by such scenarios, we investigate the problem of completely automated program inversion. Since the first Dijkstra's effort, this problem has been investigated in several areas and many techniques have been proposed. However, these techniques are only effective on simple programs [5, 12, 14, 16–18] or are only *semi*-automated [22]. Moreover, not all such techniques guarantee the soundness of the generated inverse program. We propose a *sound* and *fully automated* approach for inverting programs that manipulate lists, such as string encoders and CSV file transformations.

Programs as transducers Our approach is formal and lays its foundations in the theory of finite state automata and finite state transducers—i.e., automata with outputs. In this paper, we focus on programs that can be modeled using symbolic extended finite transducers (s-EFTs), a transducer model that can capture complex manipulating programs, such as real encoders, while retaining several decidable properties. However, the paradigm we propose could be extended to more powerful models such as tree transformations using trees and visibly push-down transducers [4, 21]. s-EFTs extend traditional transducer models in two ways. First, s-EFT transitions are labeled with predicates and functions drawn from decidable first-order theories (e.g. Presburger arithmetic over the integers). Thanks to this feature, s-EFTs can model transformations over lists that contain elements belonging to complex and potentially infinite domains. Second, s-EFT transitions can read multiple adjacent input symbols and use the inputs to produce multiple adjacent output symbols. For example, consider the following transition t of an s-EFT A operating over lists of integers.

$$p \xrightarrow[2]{x_1 \geq 0 \wedge x_2 \geq 0 / [x_2, x_1 - 1]} q$$

When in state p and reading a list (e.g., $[1, 4, 6, 10]$), the s-EFT A checks whether the first two elements of the list (e.g., 1 and 4) are greater than 0, then appends two numbers to the output list (e.g., 5 and 0), moves the control to the state q, and continues reading the rest of the list (e.g., $[6, 10]$). Despite the generality of this model, if the predicates appearing in the transitions are conjunctions of unary predicates, it is decidable to check whether two s-EFTs describe the same list transformation. This property was used by D'Antoni and Veanes to prove the correctness of complex implementations of encoders and decoders such as BASE64 [7]. In this paper, we move a step further and ask the following question. *Given an s-EFT A, can we construct an s-EFT A^{-1} that computes the inverse transformations of A?* While this problem has been studied for simple transducer models [4, 25], automatic inversion of s-EFTs is a much more challenging problem due to the use of arbitrary first order theories.

Proving transducer injectivity When trying to automatically invert a transducer A, a natural question arises: is the transducer A actually invertible? A necessary condition for invertibility is that A computes an injective function. We study the problem of checking whether an s-EFT is injective and show that, in general, this problem is undecidable. We then propose an algorithm for checking injectivity of a restricted but practical class of s-EFTs. To check injectivity of an s-EFT A, we compute predicates describing the output of each transition and use them to construct an automaton A_O describing the output language of A. Our restriction requires all the predicates in A_O to be Cartesian—i.e., the predicates are expressible as conjunction of unary predicates—and we provide an algorithm for checking whether a predicate is Cartesian. We then prove that an s-EFT is injective iff, for each possible list, there exists at most one accepting path in its output automaton A_O; we provide an algorithm for checking this property. We show that Cartesian s-EFTs can still describe complex programs.

Inverting transducers We propose an algorithm for inverting s-EFTs based on the following idea: if an s-EFT T is injective, inverting T amounts to inverting all its individual transitions. For example, the transition t' that inverts the transition t from the earlier example is the following.

$$p \xrightarrow{y_1 \geq 0 \wedge y_2 \geq -1 / [y_2 + 1, y_1]} q$$

To invert a transition we need to compute the predicate appearing in the guard and the functions generating the output. The predicate for the guard can be computed symbolically using quantifier elimination. For example, the guard of t' is the quantifier-free formula equivalent to

$$\exists x_1, x_2 . x_1 \geq 0 \wedge x_2 \geq 0 \wedge y_1 = x_2 \wedge y_2 = x_1 - 1.$$

text	M	a	n
ASCII	77 (#x4D)	97 (#x61)	110 (#x6E)
bit pattern	0 1 0 0 1 1 0 1	0 1 1 0 0 0 0 1	0 1 1 0 1 1 1 0

index	19	22	5	46
BASE64	T	W	F	u

Figure 1: Example BASE64 encoding from Wikipedia.

Computing the output components requires inverting functions expressed in an arbitrary first order theory. We formalize this problem and show that it can be naturally encoded in the framework of syntax-guided synthesis (SyGuS). For example, the output functions $g_1(y_1, y_2) = y_2 + 1$ and $g_2(y_1, y_2) = y_1$ of the inverted transition t' are solutions of the SyGuS specification

$$\begin{aligned} \varphi(g_1, g_2) \quad &= \forall x_1, x_2 . x_1 \geq 0 \wedge x_2 \geq 0 \\ &\longrightarrow g_1(x_2, x_1 - 1) = x_1 \wedge g_2(x_2, x_1 - 1) = x_2. \end{aligned}$$

***The tool* GENIC** We implemented our techniques in a programming language, GENIC, for which the semantics is given using s-EFTs. We exploit some of the domain knowledge and language features of GENIC to propose synthesis techniques that improve the performance SyGuS instances generated when inverting transitions—e.g., we synthesize auxiliary functions to speed up the synthesis. We evaluate the effectiveness of GENIC on 14 real encoders and decoders, and 40 synthetic programs, with size ranging from 6 to 131 line of code. GENIC is able to automatically invert 13 of the real programs and produces small inverses, which are substantially identical to the manually written ones.

Contributions In summary, our contributions are:

- GENIC, a language for the problem of automatically inverting list manipulating programs that are expressible as s-EFTs (§ 3).
- A formal study of the problem of checking injectivity of s-EFTs, including a proof of undecidability for the general case and an algorithm for checking injectivity for a practical subclass of s-EFTs (§ 4).
- A formal study of the problem of inverting s-EFTs, including an algorithm for inverting injective s-EFTs (§ 5), and a concrete instantiation of the algorithm in the framework of Syntax Guided Synthesis (§ 6).
- A comprehensive evaluation of GENIC on both real and synthetic benchmarks (§ 7).

A long version of this paper containing all the proofs has been submitted as supplementary material.

2. Motivating Example

We use the BASE64 encoder and its variants to illustrate how GENIC works. The standard encoding BASE64 is used to translate binary data in textual format. A BASE64 encoder transforms sequences of bytes, also called octets, into sequences of 6-bit characters. This is done by reading 3 characters at a time, splitting the resulting sequence of 24 bits into

```
1   // Axuiliary functions
2   fun E (x: x <= #x40) :=
3      (ite (x <= #x19) (x + #x41)
4         (ite (x <= #x33) (x + #x47)
5           (ite (x <= #x3d) (x - #x04)
6             (ite (x == #x3e) #x2b #x2f))))
7   fun B h l x := (x << (7 - h)) >> (7 - h + l)
8   // List transformations
9   trans B64E (l: (BitVec 8) list) : (BitVec 8) :=
10     match l with
11     // (input list) when (predicate) -> (output list)
12     | x::y::z::tail when true ->
13       E (B 7 2 x) ::
14       E (((B 1 0 x) << #x04) | (B 7 4 y)) ::
15       E (((B 4 0 y) << 2) | (B 7 6 z)) ::
16       E (B 5 0 z) :: B64E(tail)
17     | x::y::[] when true ->
18       E (B 7 2 x) ::
19       E (((B 1 0 x) << 4) | (B 7 4 y)) ::
20       E ((B 4 0 y) << 2) :: #x3d :: []
21     | x::[] when true ->
22       E (B 7 2 x)) ::
23       E ((B 1 0 x) << 4) :: #x3d :: #x3d :: []
24     | [] when true -> []
25  // Operations on the list transformation
26  isInjective B64E
27  invert B64E
```

Figure 2: GENIC program for the BASE64 encoder. #x3d is the ASCII code of the symbol '='.

groups of 6 bits, and applying a character mapping to each 6-bit number. The character mapping receives as input values between 0 and 63 to produce safe ASCII characters—e.g., letters, numbers, '+', '/' and '='. If the source stream is not divisible by 3 then the last four output characters will contain padding characters '=' to make it cleanly divisible. An example of this encoding is given in Figure 1.

The GENIC program showed in Figure 2 implements the standard BASE64 encoder. The program starts by defining two auxiliary functions (lines 2-7): the function `E` performs the character mapping illustrated in the last two lines of Figure 1 and the function `B` extracts the bits between positions `l` and `h` in a bit-vector `x`. The function `B64E` performs the actual list transformation: it takes as input a list of bytes `l` and uses pattern-matching to decide which rule to fire. The first rule (lines 12-16) fires when the input list `l` contains at least 3 elements. This rule reads the first three characters of `l`; it outputs the four ASCII characters corresponding to the BASE64 encoding of the consumed characters, and it appends to the output the result of recursively invoking `B64E` on the remainder of the list. The other three rules are fired when `l` has fewer than 3 elements. Notice that the second and third rules add the padding character '='—i.e., `#x3d`—to make the length of the output divisible by 4.

In the last two lines of the program, GENIC checks whether the list transformation `B64E` is injective and, if that is the case, GENIC computes the inverse of `B64E`. A skeleton of the GENIC program `B64D` produced by **`invert`** `B64E` is depicted in Figure 3. In this case, GENIC also synthesizes an auxiliary function `D` that inverts our original auxiliary function `E` and uses it in the body of the program `B64D`, making the inverted

```
1   // We omit the detail in auxiliary functions
2   fun D (x:...) := ... // Synthesized by genic
3   fun B h l x := ...
4   fun pred x := ... // Synthesized by genic
5   ===
6   // Synthesized by genic
7   trans B64D (l: (BitVec 8) list) : (BitVec 8) :=
8      match l with
9      | ...
10     | x::y::z::w::[] when (and (pred x) (pred y)
11             (z == #x3d) (w == #x3d)) ->
12       ((D x) << #x02) | (B #x05 #x04 (D y)) :: []
13     | ...
```

Figure 3: Skeleton of the implementation of the BASE64 decoder synthesized by GENIC.

program more natural to read. GENIC takes approximately 2 seconds to prove that `B64E` is injective and approximately 10 seconds to compute its inverse.

We now introduce a variant of the standard BASE64 and show how a small change in the encoder can trigger non-trivial changes in the corresponding decoder. The modified BASE64 for XML tokens differ from the standard BASE64 because it uses a different mapping function `E`, which maps values 62 and 63 to characters '.' and '-' respectively, and it does not use padding characters. It is easy to modify the program in Figure 2 to reflect these changes. However, this small change in the encoder triggers non-trivial changes in the corresponding decoder. For example, for the skeleton of the decoder presented in Figure 3, we need to modify the function `D` and the predicate `pred` to operate over a different set of symbols, and we need to change the pattern of the rule in line 10 to read only two symbols. Despite this non-trivial change, GENIC can prove the injectivity of the modified encoder and compute its inverse in approximately 10 seconds, relieving the programmer from the burden of manually writing a correct modified decoder.[1]

3. The GENIC Language

The language GENIC is designed with the following goals in mind. First, GENIC should be expressive enough to model useful and practical list manipulating programs. Second, it should be possible to automatically check whether GENIC programs are injective and to invert them. We introduce the language constructs of GENIC and simultaneously define their formal semantics using s-EFTs.

3.1 Alphabet Theories

GENIC programs and s-EFTs manipulate lists that contain elements from complex domains. We assume a *background*

[1] To better appreciate the complexity of the problem the reader can look at the Javascript implementation of BASE64 at `http://bit.ly/2eIJeSe`. To adopt the change proposed in our motivating example, one will need to 1) modify lines 67-69 and 83-85 since now there are no = characters, 2) add a mechanism to detect the early end of input and react appropriately (e.g., change lines 79 and 92), 3) modify the decoding table in line 16. We believe that these changes are non-trivial and error-prone.

universe $\mathfrak{D}$ with built-in function and relation symbols. The universe $\mathfrak{D}$ is multi-typed with $\mathfrak{D}_\tau$ denoting the sub-universe of elements of type τ. We use λ-expressions for representing anonymous functions that we call *λ-terms*. A λ-term $\lambda x.\varphi(x)$ of type $\sigma \to \textsc{Bool}$ is called a σ-predicate or a predicate over σ. We use σ^i to denote the Cartesian product of σ i times—i.e., $\sigma^0 = \emptyset$, $\sigma^1 = \sigma$ and $\sigma^{i+1} = \sigma \times \sigma^i$.

An *alphabet theory* is given by a set Ψ of terms that is closed under Boolean operations (i.e., the theory forms a Boolean algebra), substitution, equality, and if-then-else terms. Unless stated differently, we assume that the alphabet theory Ψ is *decidable*—i.e., checking satisfiability of formulas $\varphi \in \Psi$, $\mathit{IsSat}(\varphi)$, is decidable. The alphabet theory is recursively enumerable if the set Ψ is recursively enumerable. We use $[\![\varphi]\!] \subseteq \mathfrak{D}_\sigma$ to denote the set of all values that satisfy φ; φ is *valid*, $\mathit{IsValid}(\varphi)$, when $[\![\varphi]\!] = \mathfrak{D}_\sigma$. GENIC currently supports the theories of bit-vector arithmetic and linear integer arithmetic, which are ones supported by SYGUS solvers.

3.2 Auxiliary Functions

The first part of a GENIC program contains auxiliary partial functions defined using terms in the alphabet theory. Auxiliary functions can be used as within the list transformations and we will discuss later how they can speed up synthesis.

Example 3.1. *The* GENIC *program presented in Figure 2 contains two auxiliary functions (lines 2-7). The partial function* `E` *maps characters in the range* `[#00-#3f]` *to the corresponding* BASE64 *symbols and is undefined on characters outside the range* `[#00-#3f]`*. The function* `B` *extracts the bits between position* `h` *and* `l` *in a bit-vector* `x`*. In the following examples we use $b^h_l(x)$ and $\lceil x \rceil$ to refer to the terms in the alphabet theory $\mathfrak{D}$ corresponding to the* GENIC *functions* `B h l x` *and* `E x`*, respectively.*

3.3 List Transformations

The core constructs of GENIC are list-to-list transformations, for which the semantics is given in terms of Symbolic Extended Finite Transducers (s-EFTs).

Definition 3.2. *An* Extended Symbolic Finite Transducer (s-EFT) *with* input type σ *and* output type γ *is a tuple $A = (Q, q^0, \Delta)$ where:*

- *Q is a finite set of* states *and $q^0 \in Q$ is the* initial state.
- *Δ is a set of* transitions *of the form $r = (p, \ell, \varphi, \hat{f}, q)$ (denoted $p \xrightarrow[\ell]{\varphi/\hat{f}} q$) such that $p \in Q$, $q \in Q \cup \{\bullet\}$, $\ell \geq 1$ is the lookahead of r, and*
  - *φ, the* guard *of r, is a predicate over σ^ℓ;*
  - *$\hat{f}$, the* output *of r, is a list of functions $[f_0, \ldots, f_n]$ such that $n \geq 0$ and for every i, $f_i : (\sigma^\ell \to \gamma)$.*

The lookahead *of A is the maximum of all lookaheads of rules in Δ. An s-EFT where all the rules output the empty list is an* Extended Symbolic Finite Automaton *(*s-EFA*). For s-EFAs, we omit the output component from the transitions.*

A finalizer is a rule with target state $q = \bullet$ and is a generalization of the notion of final state. A finalizer with lookahead ℓ is used when the end of the input sequence has been reached with *exactly* ℓ input elements remaining. We use the following abbreviated notation for rules, by omitting explicit λ's. We write

$$p \xrightarrow[\ell]{\varphi(\bar{x})/[f_0(\bar{x}),\ldots,f_k(\bar{x})]} q \text{ for } p \xrightarrow[\ell]{\lambda\bar{x}.\varphi(\bar{x})/\lambda\bar{x}.[f_0(\bar{x}),\ldots,f_k(\bar{x})]} q,$$

where φ and f_i are terms whose free variables are among $\bar{x} = (x_0, \ldots, x_{\ell-1})$. We often abbreviate $[f_0(\bar{x}), \ldots, f_k(\bar{x})]$ with $\bar{f}(\bar{x})$.

In the following we explain how GENIC programs are translated into s-EFTs. In GENIC a list transformation is declared using the signature **trans** p and it represents a state p of an s-EFT together with the corresponding transitions out of p. Transformations contain two kinds of matching rules. The first kind is of the form

$$\begin{aligned}&x_0 :: \ldots :: x_n :: \mathit{tail} \ \textbf{when}\ \varphi(x_0, \ldots, x_n) \texttt{ ->}\\ &\qquad f_1(x_0, \ldots, x_n) :: \ldots :: f_k(x_0, \ldots, x_n) :: p'(\mathit{tail})\end{aligned}$$

and it corresponds to an s-EFTs transition of the form

$$p \xrightarrow[n+1]{\varphi(x_0,\ldots,x_n)/[f_0(x_0,\ldots,x_n),\ldots,f_k(x_0,\ldots,x_n)]} p'.$$

The second kind of matching rules is of the form

$$\begin{aligned}&x_0 :: \ldots :: x_n :: \texttt{[]} \ \textbf{when}\ \varphi(x_0, \ldots, x_n) \texttt{ ->}\\ &\qquad f_0(x_0, \ldots, x_n) :: \ldots :: f_k(x_0, \ldots, x_n) :: \texttt{[]}\end{aligned}$$

and it corresponds to an s-EFTs finalizer transition

$$p \xrightarrow[n+1]{\varphi(x_0,\ldots,x_n)/[f_0(x_0,\ldots,x_n),\ldots,f_k(x_0,\ldots,x_n)]} \bullet$$

Intuitively, rules that pattern-match against a list of fixed length correspond to finalizer transitions. The predicates and functions appearing in GENIC rules are required to be well-typed and to be elements of the alphabet theory. The initial state of the s-EFT is the one corresponding to the transformation for which the user wants to compute the inverted program—i.e., the name appearing after the keyword **invert**.

Example 3.3. *The s-EFT $T_{B64E} = (\{p\}, p, \Delta_{B64E})$ describes the* BASE64 *encoder presented in Figure 2. T_{B64E} operates over the theory of bit-vectors and has* BITVEC 8 *as both its input and output type. The set Δ_{B64E} contains the following transitions. $\top$ is the true predicate.*

$$p \xrightarrow[3]{\top/[\lceil b^7_2(x_0)\rceil, \lceil(b^1_0(x_0){\ll}4)|b^7_4(x_1)\rceil, \lceil(b^3_0(x_1){\ll}2)|b^7_6(x_2)\rceil, \lceil b^5_0(x_2)\rceil]} p$$

$$p \xrightarrow[0]{\top/[\,]} \bullet \qquad p \xrightarrow[1]{\top/[\lceil b^7_2(x_0)\rceil,\ \lceil b^1_0(x_0){\ll}4\rceil,\ \text{‘=’},\ \text{‘=’}]} \bullet$$

$$p \xrightarrow[2]{\top/[\lceil b^7_2(x_0)\rceil,\ \lceil(b^1_0(x_0){\ll}4)|b^7_4(x_1)\rceil,\ \lceil b^3_0(x_1){\ll}2\rceil,\ \text{‘=’}]} \bullet$$

The state p corresponds to the list transformation function `B64E` *and, since the program asks to invert* `B64E`*, the state p is also the initial state.*

We now define the semantics of s-EFTs and therefore of GENIC programs. In the remainder of the section, let $A = (Q, q^0, \Delta)$ be a fixed s-EFT with input type σ and output type γ. For each rule in Δ we define the set of corresponding non-symbolic rules as follows.

$$[\![p \xrightarrow[\ell]{\varphi(x_0,\dots,x_{\ell-1})/[f_0(x_0,\dots,x_{\ell-1}),\dots,f_k(x_0,\dots,x_{\ell-1})]} q]\!] \stackrel{\text{def}}{=}$$
$$\{p \xrightarrow{\bar{a}/[t_0,\dots,t_k]} q \mid |\bar{a}| = \ell \wedge \bar{a} \in [\![\varphi]\!] \wedge t_i = [\![f_i]\!](\bar{a})\}$$

Intuitively, a rule with lookahead ℓ reads ℓ adjacent input symbols $\bar{a}$ and produces a sequence of output symbols $t_0, \dots, t_k$ by applying the output functions in $\bar{f}$ to $\bar{a}$.

In the following, let $[\![\Delta]\!] \stackrel{\text{def}}{=} \bigcup_{r\in\Delta}[\![r]\!]$ and let $s_1 \cdot s_2$ denote the concatenation of two sequences s_1 and s_2.

Definition 3.4. *For $u \in \Sigma^*, v \in \Gamma^*, q \in Q, q' \in Q \cup \{\bullet\}$, define $q \overset{u/v}{\twoheadrightarrow}_A q'$ as follows: there exists $n \geq 0$ and $\{r_i \mid r_i = p_i \xrightarrow{u_i/v_i} p_{i+1} \wedge i \leq n\} \subseteq [\![\Delta]\!]$ such that $u = u_0 \cdot u_1 \cdots u_n$, $v = v_0 \cdot v_1 \cdots v_n$, $q = p_0$, and $q' = p_{n+1}$. Let also $q \overset{\varepsilon/\varepsilon}{\twoheadrightarrow}_A q$ for all $q \in Q$.*

Definition 3.4 describes a path between two states in an s-EFT where transitions are traversed by reading symbols in the input list. Transduction paths are required to start at the initial state and end in the finalizer state $\bullet$. We use $Paths_A(u, \bullet) = \{(p_0, r_0)\cdots(p_n, r_n)\}$ to denote the set of paths with which u can reach a finalizer—i.e., the transition r_n is of the form $p_n \xrightarrow[\ell]{u_n/v_n} \bullet$.

Definition 3.5. *The* transduction *of A is defined as*

$$\mathbb{T}_A(u) \stackrel{\text{def}}{=} \{v \mid q^0 \overset{u/v}{\twoheadrightarrow} \bullet\}.$$

s-EFTs have nondeterministic semantics and transductions typically represent relations rather than functions. Unambiguous s-EFTs are an interesting subclass of s-EFTs that compute functions.

Definition 3.6. *An s-EFT A is* unambiguous *if for every list u, $|Paths_A(u, \bullet)| \leq 1$.*

When an s-EFT A is unambiguous, $\mathbb{T}_A(u)$ is either a singleton set or the empty set. We write $\mathbb{T}_A(u) = t$ when $\mathbb{T}_A(u) = \{t\}$ and $\mathbb{T}_A(u) = \perp$ when $\mathbb{T}_A(u) = \{\}$. When $\mathbb{T}_A(u) \neq \perp$, we write $Path_A(u, \bullet) = (p_0, r_0)\cdots(p_n, r_n)$ to denote the unique run of A on u.

Unfortunately, it is undecidable to check whether an s-EFT is ambiguous [8], which means that in general we cannot check whether a GENIC program is ambiguous or not. Instead, we require GENIC programs to be deterministic because this property can be automatically checked. We define $\varphi \curlywedge \psi$, where φ is a σ^m-predicate and ψ a σ^n-predicate, as the $\sigma^{\max(m,n)}$-predicate $\lambda(x_1, \dots, x_{\max(m,n)}).\varphi(x_1, \dots, x_m) \wedge \psi(x_1, \dots, x_n)$. We define *equivalence of f and g modulo φ*, $f \equiv_\varphi g$, as: *IsValid*$(\lambda\bar{x}.(\varphi(\bar{x}) \Rightarrow f(\bar{x}) = g(\bar{x})))$.

Definition 3.7. *An s-EFA A is* deterministic *if for all transitions $p \xrightarrow[\ell]{\varphi/f} q$ and $p \xrightarrow[\ell']{\varphi'/f'} q' \in \Delta$ one of the following properties hold.*

(a) If $q, q' \in Q$ and $IsSat(\varphi \curlywedge \varphi')$, then $q = q'$, $\ell = \ell'$ and $f \equiv_{\varphi\curlywedge\varphi'} f'$.
(b) If $q = q' = \bullet$, $IsSat(\varphi \curlywedge \varphi')$, and $\ell = \ell'$, then $f \equiv_{\varphi\curlywedge\varphi'} f'$.
(c) If $q \in Q$, $q' = \bullet$, and $IsSat(\varphi \curlywedge \varphi')$, then $\ell > \ell'$.

Clearly, a deterministic s-EFA is also unambiguous. Intuitively, determinism means that no two rules may overlap. To check whether a GENIC program is deterministic we check whether the induced s-EFT is deterministic.

Example 3.8. *The* GENIC *program in Figure 2 is deterministic because at most one rule can be triggered for any input.*

Without loss of generality, we assume that all states are reachable from the initial state and can reach the finalizer $\bullet$.

3.4 Operations

GENIC supports the following operations over list transformations. **isInjective** f checks whether the transformation f is injective and returns two different input lists that produce the same output otherwise (Section 4). **invert** f constructs a program f^{-1}, which is the inverse of the injective transformation f—i.e., for every input x, $\mathrm{f}^{-1}(\mathrm{f}(x)) = x$ (Section 5).

4. Checking s-EFT Injectivity

To invert a GENIC program we first need to check whether the program is injective—i.e., whether for any two distinct inputs the program produces distinct outputs. This property is necessary for invertibility, but not sufficient (We will expand on this aspect in Section 5). In this section, we formally study the problem of checking whether an s-EFT is injective. We first show that the problem is undecidable and then present a technique for checking injectivity for a restricted but practical class of s-EFTs. For the sake of generality, we state our theorems for unambiguous s-EFTs, but all theorems also hold for deterministic s-EFTs.

4.1 A Theory of Injectivity for s-EFTs

We first define what it means for an s-EFT to be injective and then show that this property is equivalent to the conjunction of two simpler properties called transition-injectivity and path-injectivity.

Definition 4.1 (Injectivity)**.** *An s-EFT A is* injective *iff for every two lists u and v, if $\mathbb{T}_A(u) \neq \perp$, $\mathbb{T}_A(v) \neq \perp$, and $u \neq v$, then $\mathbb{T}_A(u) \neq \mathbb{T}_A(v)$.*

The following definition captures the case in which an s-EFT is not injective because an individual transition can produce the same output when provided with different input.

Definition 4.2 (Transition-injective s-EFT)**.** *A transition $p \xrightarrow[\ell]{\varphi(\bar{x})/\bar{f}} q$ is* injective *iff every two lists $\bar{a}$ and $\bar{b}$ of size*

ℓ, *if* $\bar{a} \neq \bar{b}$, $\varphi(\bar{a})$ *and* $\varphi(\bar{b})$, *then* $[\![\bar{f}]\!](\bar{a}) \neq [\![\bar{f}]\!](\bar{b})$. *An s-EFT* A *is* transition-injective *iff all its transitions are injective.*

Example 4.3. *The transition* $p \xrightarrow[2]{\top/[x_0+1,x_1]} q$ *is injective because both its output functions are injective. The transition* $p \xrightarrow[1]{\top/[x_0^2]} q$ *is not injective because the function* x^2 *is not injective. If we restrict the domain of the transition to the set of positive numbers we obtain an injective transition* $p \xrightarrow[1]{x_0>0/[x_0^2]} q$.

The following notion captures the case in which an s-EFT is not injective because two different input lists with different accepting paths produce the same output.

Definition 4.4 (Path-injective s-EFT)**.** *An unambiguous s-EFT* A *is* path-injective *iff for every two lists* u *and* v*, if* $\mathbb{T}_A(u) \neq \bot$, $\mathbb{T}_A(v) \neq \bot$, *and* $Path_A(u, \bullet) \neq Path_A(v, \bullet)$, *then* $\mathbb{T}_A(u) \neq \mathbb{T}_A(v)$.

Example 4.5. *Consider the s-EFT* $P = (\{p, q\}, p, \Delta_P)$ *with the following transitions.*

$$p \xrightarrow[1]{x_0>0/[x_0-5]} q, \quad q \xrightarrow[1]{x_0>0/[x_0-5]} \bullet,$$
$$p \xrightarrow[2]{x_0<0 \wedge x_1<0/[x_0+5,x_1+5]} \bullet$$

This s-EFT P *is transition-injective, but not path-injective, because on both the inputs* $[5, 5]$ *and* $[-5, -5]$ *it outputs the list* $[0, 0]$.

We now show the relations between s-EFT injectivity and the two properties we just defined.

Theorem 4.6. *An unambiguous s-EFT* A *is injective iff* A *is both transition-injective and path-injective.*

Our characterization of injectivity in terms of transition- and path-injectivity is unique to symbolic transducers. In finite transducers, which operate over finite alphabets, all transitions are trivially injective

4.2 Checking Transition-Injectivity

We provide an algorithm for checking whether an s-EFT is transition-injective.

Lemma 4.7. *It is decidable to check whether an s-EFT is transition-injective.*

Proof. Checking whether an s-EFT A is transition-injective amounts to checking whether all its transitions are injective. To see whether a transition $p \xrightarrow[\ell]{\varphi(\bar{x})/\bar{f}(\bar{x})} q$ is injective we can test whether the following Boolean formula is satisfiable.

$$(\bar{x}_1 \neq \bar{x}_2) \wedge \varphi(\bar{x}_1) \wedge \varphi(\bar{x}_2) \wedge ([\![f]\!](\bar{x}_1) = [\![f]\!](\bar{x}_2))$$

This test is decidable because the alphabet theory of A forms a decidable Boolean algebra. □

4.3 Checking Path-Injectivity

In general, checking path injectivity is undecidable.

Theorem 4.8. *It is undecidable to check whether a deterministic s-EFT is path-injective or injective.*

Proof. Recall that a Minsky machine is a program with a finite sequence of instructions that has two registers r_1 and r_2 that can hold natural numbers. Each instruction is one of the following: INC_i (increment r_i and continue with the next instruction); DEC_i (decrement r_i if $r_i > 0$ and continue with the next instruction); $JZ_i(j)$ (if $r_i = 0$ then jump to the j'th instruction else continue with the next instruction). The machine halts when the end of the program is reached. Let M be a Minsky machine with program P. We use a tuple (*program counter*, r_1, r_2) of type $\sigma = \gamma = \mathbb{N}^3$ to represent a configuration of M. Let $\pi_j : \sigma \to \mathbb{N}$ be the function that projects the j'th element of a k-tuple where $0 \leq j < k$.

We construct an s-EFT A, which is ambiguous iff the Minsky machine M halts on input $(0, 0)$ with a non-zero output in r_1. Let φ^{000} be the predicate $\lambda x.x = (0, 0, 0)$ and let φ^{111} be the predicate $\lambda x.x = (1, 1, 1)$. Let φ^{fin} be the final predicate $\lambda x.\pi_0(x) = |P| \wedge \pi_1(x) \neq 0$. The predicates φ^{000} and φ^{fin} describe the initial and final configurations of M, respectively.

We define a binary predicate φ^{step} that describe pairs of correct adjacent configurations in M. The predicate φ^{step} is of the form $\lambda(x, x'). \bigvee_{i<|P|} \varphi_i^{\text{step}}$, where φ_i^{step} is the formula describing the valid step relation of M from a configuration x to the next configuration x', when executing the i-th instruction. For example, if the i'th instruction is INC_1, then φ_i^{step} is

$$\pi_0(x) = i \wedge \pi_0(x') = i+1 \wedge \pi_1(x') = \pi_1(x)+1 \wedge \pi_2(x') = \pi_2(x).$$

The encoding is similar for the other instructions. The s-EFT A is the tuple $(\{q_0, q_1, q_2, q_3\}, q_0, \Delta)$ where Δ contains the following transitions.

$$q_0 \xrightarrow[1]{\varphi^{000}(x_1)/[\,]} q_1, \quad q_1 \xrightarrow[2]{\varphi^{\text{step}}(x_1,x_2)/[x_1,x_2]} q_1,$$
$$q_1 \xrightarrow[0]{\top/[\,]} \bullet \quad , q_0 \xrightarrow[1]{\varphi^{111}(x_1)/[\,]} q_2, \quad q_1 \xrightarrow[1]{\varphi^{000}(x_1)/[x_1]} q_2$$
$$q_2 \xrightarrow[2]{\varphi^{\text{step}}(x_1,x_2)/[x_1,x_2]} q_2, \quad q_2 \xrightarrow[1]{\varphi^{\text{fin}}(x_1)/[x_1]} \bullet$$

The s-EFT A is deterministic and transition-injective. Given a list of the form $(0, 0, 0)a_1 \ldots a_{2n}$, A outputs $a_1 \ldots a_{2n}$ iff, for every $1 \leq i \leq n$, the symbols a_{2i-1} and a_{2i} are correct adjacent configurations of the machine M. The transduction is undefined on all other list starting with $(0, 0, 0)$. Given a list of the form $(1, 1, 1)a_1 \ldots a_{2n}$, A outputs $a_1 \ldots a_{2n}$ iff, for every $1 \leq i < n$, the symbols a_{2i} and a_{2i+1} are correct adjacent configurations of the machine M, a_1 is the initial configuration of M, and a_{2n} is the final configuration of M. The transduction is undefined on all other lists starting with $(1, 1, 1)$.

Notice that each individual transition computes an injective function. The s-EFT A is not injective iff there exists a sequence $a_1 \dots a_{2n}$ for which A is defined on both $(0,0,0)a_1 \dots a_{2n}$ and $(1,1,1)a_1 \dots a_{2n}$, as these two inputs would produce the same output $a_1 \dots a_{2n}$. The existence of such a sequence implies that $a_1 \dots a_{2n}$ is an accepting run of the Minsky machine M as it satisfies the step relations at all positions and it has the correct initial and final configuration.

It follows that A is not injective iff M halts on input $(0,0)$ with a non-zero output in r_1. The latter is an undecidable problem as an instance of Rice's theorem. This concludes our proof. $\square$

The proof relies on the use of symbolic alphabets; in fact, checking injectivity of finite transducers, which operate over finite alphabets, is decidable [13].

In the following, we provide a technique for checking path-injectivity for a restricted but practical class of s-EFTs. Given an s-EFT A, we show how to construct a non-deterministic s-EFA A_O that accepts the set of all output lists produced by A and show that A is path-injective iff A_O is ambiguous—i.e., there exists an input with two different accepting paths. Finally, if all the predicates appearing in the transitions of A_O are expressible as conjunctions of unary predicates, we provide an algorithm for checking whether A_O is ambiguous.

Definition 4.9 (Output s-EFA)**.** *Given an s-EFT $A = (Q, q_0, \Delta)$, the* output automaton *of A is the s-EFA $A_O = (Q \cup \Delta, q_0, \Delta_O)$ such that for every transition $t = p \xrightarrow[\ell]{\varphi(\bar{x})/\bar{f}(\bar{x})} q \in \Delta$, Δ_O contains two transitions $t^\epsilon = p \xrightarrow[0]{true} q_t$ and $t^{out} = q_t \xrightarrow[|\bar{f}|]{\exists \bar{a}.\varphi(\bar{a}) \wedge \bar{x} = [\![\bar{f}]\!](\bar{a})} q$ where q_t denotes the state representing t.*

For every list v accepted by A_O, there exists a list u such that $\mathbb{T}_A(u) = v$. We can now define the relation between path-injectivity of A and ambiguity of A_O .

Lemma 4.10. *An unambiguous s-EFT A is path-injective iff its output automaton A_O is unambiguous.*

Intuitively, this lemma follows from the fact that for each input of A there exists a path in A_O accepting the corresponding output. Therefore, if A on two different inputs produces the same output l, there will be two distinct paths in A_O accepting the list l.

Example 4.11. *Consider the s-EFT P from Example 4.5. The output automaton of P_O has the following transitions.*

$$p \xrightarrow[0]{true} pt_1, \quad pt_1 \xrightarrow[1]{\exists y. y>0 \wedge x_0 = y-5} q,$$
$$q \xrightarrow[0]{true} qt_2, \quad qt_2 \xrightarrow[1]{\exists y. y>0 \wedge x_0 = y-5} \bullet,$$
$$p \xrightarrow[0]{true} pt_3, \quad pt_3 \xrightarrow[2]{\exists y_0, y_1 . y_0 < 0 \wedge y_1 < 0 \wedge x_0 = y_0 + 5 \wedge x_1 = y_1 + 5} \bullet$$

The s-EFA P_O is ambiguous because the list $[0,0,0]$ is accepted by the two paths containing the sequences of states $p, pt_1, q, qt_2, \bullet$ and $p, pt_3, \bullet$.

Using Theorem 4.8 we have that it is undecidable to check whether an s-EFA is unambiguous. We introduce a subclass of s-EFAs for which checking ambiguity is decidable. A binary relation R over X is *Cartesian over X* if R is the Cartesian product $R_1 \times R_2$ of some $R_1, R_2 \subseteq X$. The definition is lifted to n-ary relations and σ^n-predicates for $n \geq 2$ in an obvious way. To decide if a satisfiable predicate $\varphi(\bar{x})$ of arity n is Cartesian over a type σ (denoted $IsCartesian(\varphi)$), given a model $(a_0, \dots, a_{n-1})$ of φ we can check whether the following formula is valid.

$$\forall \bar{x}\, (\varphi(\bar{x}) \Leftrightarrow \textstyle\bigwedge_{i<n} \varphi(a_0, \dots, a_{i-1}, x_i, a_{i+1}, \dots, a_{n-1}))$$

In other words, a predicate $\varphi(\bar{x})$ is Cartesian over σ iff φ can be rewritten equivalently as a conjunction of n independent unary predicates. Notice that we can use the validity decision procedure of the alphabet theory to decide whether a predicate is cartesian.

Definition 4.12 (Cartesian s-EFA [8])**.** *An s-EFA is* Cartesian *if all its guards are Cartesian.*

Example 4.13. *The s-EFA P_O in Example 4.11 is Cartesian. The only transition containing a binary predicate is the last one, which contains the predicate $\exists y_0, y_2 . y_0 < 0 \wedge y_1 < 0 \wedge x_0 = y_0 + 5 \wedge x_1 = y_1 + 5$. This predicate is equivalent to $x_0 < 5 \wedge x_1 < 5$. An example of non-Cartesian predicates is the predicate $x_0 = x_1$ over the type $\mathbb{N}$.*

Cartesian s-EFAs can be transformed into equivalent s-FAs by splitting each transition of lookahead k into k transitions of lookahead 1. Since it is decidable to check whether an s-FA is ambiguous, we have that checking ambiguity is also decidable for Cartesian s-EFAs. The algorithm is based on the product construction used for checking NFA ambiguity.

Lemma 4.14 (s-EFA ambiguity)**.** *Checking whether a Cartesian s-EFAs is unambiguous is decidable.*

Example 4.15. *Recall the s-EFT for* BASE64 *in Example 3.3. The corresponding output automaton is a Cartesian s-EFA and has the following transitions*

$$p \xrightarrow[4]{\beta_{64}(x_0) \wedge \beta_{64}(x_1) \wedge \beta''_{64}(x_2) \wedge x_3 = \text{‘=’}} \bullet \qquad p \xrightarrow[4]{\bigwedge_{i=0}^{3} \beta_{64}(x_i)} p$$
$$p \xrightarrow[0]{\top} \bullet \qquad p \xrightarrow[4]{\beta_{64}(x_0) \wedge \beta'_{64}(x_1) \wedge x_2 = \text{‘=’} \wedge x_3 = \text{‘=’}} \bullet$$

where the predicate $\beta_{64}(y)$ is true iff y is a valid BASE64 *digit, i.e., $y = \lceil x \rceil$ for some x, $0 \leq x \leq 64$. The predicates $\beta'_{64}(y)$ and $\beta''_{64}(y)$ are restricted versions of $\beta_{64}(y)$ and the* ASCII *of* ‘=’ *is not a valid* BASE64 *digit. This automaton is deterministic and therefore unambiguous, which implies that the s-EFT in Example 3.3 is path-injective.*

We can now state our main decidability result.

Theorem 4.16 (Cartesian-output s-EFT injectivity)**.** *Given an unambiguous s-EFT $A = (Q, \{q_0\}, \Delta)$, if the output automaton A_O is Cartesian, then it is decidable to check whether A is injective. Moreover, there exists an algorithm for checking injectivity with complexity $O((n + t\ell)^2 + (t\ell)^2 f(2k))$, where $n = |Q|$, $t = |\Delta|$, ℓ is the lookahead of A, k is the size of the largest transition in A, and $f(x)$ is the complexity of checking satisfiability of predicates of size x in the alphabet theory of A.*

Proof. Decidability follows from Theorems 4.6 and 4.7, and Lemmas 4.10 and 4.14. Checking whether A is transition-injective requires one satisfiability check for every transition and has complexity $O(tf(k))$. The s-EFA A_O has the same lookahead of A and the same number of states and transitions. Moreover, each predicate in A_O has size $O(k)$. The Cartesian s-EFA A'_O equivalent to A_O has $O(n + t\ell)$ states, $O(t\ell)$ transitions, the largest predicate has size $O(k)$. Finally, the product automaton B has $O((n + t\ell)^2)$ states, $O((t\ell)^2)$ transitions, and the largest predicate has size $O(2k)$ because of the conjunctions appearing on each transition. The emptiness of B can be checked using a depth-first search. Hence, the complexity $O((n + t\ell)^2 + (t\ell)^2 f(2k))$. □

5. Inverting s-EFTs

In this section, we formalize the theory of s-EFT inversion and describe how to translate it into practice using SyGuS solvers. For the sake of generality, we state our theorems for unambiguous s-EFTs, but all theorems also hold for deterministic s-EFTs. The functions we invert may be partial. We start by defining the inverse of an s-EFT.

Definition 5.1 (Inverse s-EFT)**.** *An unambiguous s-EFT A is an* inverse *of an unambiguous s-EFT B iff, for every list u and v, $\mathbb{T}_A(u) = v$ iff $\mathbb{T}_B(v) = u$.*

Clearly a necessary condition for admitting an inverse is that A is injective. We provide a symbolic s-EFT construction for characterizing the structure of an inverse s-EFT. We first define the notion of inverse transitions and then use it to build inverse s-EFTs.

Definition 5.2 (Transition inverse)**.** *A transition $(p, k, \psi, \bar{g}, q)$* inverts *a transition $(p, \ell, \varphi, \bar{f}, q)$ iff $k = |\bar{f}|$, $\ell = |\bar{g}|$, $\psi(\bar{y}) \equiv \exists\bar{x}.\varphi(\bar{x}) \wedge (\bar{y} = \bar{f}(\bar{x}))$, and $\forall\bar{x}.\varphi(\bar{x}) \to (\bar{g}(\bar{f}(\bar{x})) = \bar{x})$.*

Given a transition r, we use $inv(r)$ to denote the set of all transitions that invert r and r^{-1} to denote some element of $inv(r)$. Intuitively, if a transition r outputs the list v when reading a list u, the inverse transition r^{-1} outputs u when reading v. This definition is symmetric—i.e., t inverts r iff r inverts t. For r to have an inverse, r must be injective.

Example 5.3. *Consider two s-EFT's transitions*

$$p \xrightarrow[2]{x_0<0 \wedge x_1<0/[x_0+5, x_1+5]} q \qquad p \xrightarrow[2]{y_0<5 \wedge y_1<5/[y_0-5, y_1-5]} q$$

The two transition invert each other. The guard of the right transition, $y_0 < 5 \wedge y_1 < 5$, is equivalent to the quantified formula describing the set of possible outputs of the left transition. The composition of their output functions is equivalent to identity function when evaluated over elements satisfying the guard $x_0 < 0 \wedge x_1 < 0$.

If we have an effective way for inverting transitions, we can construct an inverse of an s-EFT $A = (Q, q^0, \Delta)$ as s-EFT $A^{-1} = (Q, q^0, \Delta^{-1})$ where $\Delta^{-1} \overset{\text{def}}{=} \{r^{-1} \mid r \in \Delta\}$.

Theorem 5.4. *Given an unambiguous injective s-EFT A, the s-EFT A^{-1} is an inverse of A and is unambiguous.*

Theorem 5.4 shows that the inverse of a deterministic s-EFT might be nondeterministic but unambiguous.

Example 5.5. *Consider the s-EFT D with the following transitions and initial state q_0.*

$$q_0 \xrightarrow[1]{x_0<0/[x_0]} q_1, \quad q_0 \xrightarrow[1]{x_0>0/[-x_0]} q_2$$
$$q_2 \xrightarrow[0]{true/[x_0]} q_1, \quad q_1 \xrightarrow[0]{true/[\,]} \bullet$$

The s-EFT D is deterministic and injective, and its inverse D^{-1} contains the following transitions.

$$q_0 \xrightarrow[1]{x_0<0/[x_0]} q_1, \quad q_0 \xrightarrow[1]{x_0<0/[-x_0]} q_2$$
$$q_2 \xrightarrow[1]{true/[x_0]} q_1, \quad q_1 \xrightarrow[0]{true/[\,]} \bullet$$

The s-EFT D^{-1} is non-deterministic and unambiguous.

Theorem 5.4 shows that, if we have an algorithm for inverting transitions, we can directly use it to compute inverse s-EFTs. We say that an alphabet theory *admits inverse functions* if for every two types σ and γ, for every predicate $\varphi(\bar{x})$ of type $\sigma \to$ Bool, and for every function $f(\bar{x})$ of type $\sigma \to \gamma$, there exists a function g of type $\sigma \to \gamma$ in $\mathfrak{D}$ such that $\forall\bar{x}.\varphi(\bar{x}) \to (g(f(\bar{x})) = \bar{x})$. This class of alphabet theories guarantees the existence of the inverse of a transition (Definition 5.2). The following corollary immediately follows from Theorem 5.4.

Corollary 5.6. *Given an unambiguous injective s-EFT A over a recursively enumerable alphabet theory that admits inverse functions, there exists an s-EFT B that is the inverse of A. Moreover, constructing B is decidable.*

To compute B, we can simply invert each transition by enumerating all functions in the alphabet theory. The existence of such functions is guaranteed because the alphabet theory admits inverse functions. In the concrete implementation of Genic, we use more effective synthesis techniques to produce inverse functions. An interesting aspect of this algorithm is that all the transitions can be inverted independently and the computation of B is amenable for parallelization.

Example 5.7. *The theory of bit-vector arithmetic admits inverse functions because it operates over a finite domain.*

The theory of arithmetic with transcendental functions is not recursively enumerable and it does not admit inverse functions. For example, the inverse of a function $f(x) = x - sin\ x$ cannot be expressed using a term of finite size.

6. Inverting s-EFTs in GENIC

The previous section presented a framework for building the inverse of an s-EFT by computing the inverse of all its transitions. However, the proposed techniques are far from being practical. First, if we use Definition 5.2 directly to produce the guard of the inverse transition, we get a predicate containing quantifiers. Predicates containing quantifiers are not practical because they cannot be programmed efficiently and they are not directly expressible in GENIC. To address the first issue, GENIC uses quantifier elimination to produce a guard that does not contain quantifiers. As we discussed in Section 3, GENIC supports the theories of linear integer arithmetic and bit-vector arithmetic, and these theories both admit decidable quantifier elimination.

Second, if we look at Corollary 5.6, the algorithm proposed for generating the output term in the inverse transition requires an exhaustive enumeration of all the functions in the alphabet theory. In this section, we show how the problem of computing inverse functions of a transition can be encoded in the framework of Syntax-Guided Synthesis [2].

Background A Syntax-Guided Synthesis (SYGUS) problem is specified with respect to a background theory T that fixes the type of variables, operations on types, and their interpretation. The goal of a SYGUS problem is to synthesize a function f of a given type which satisfies two constraints provided by users. The first constraint describes a semantic property that f should satisfy and is given as a predicate $\psi(e) \overset{\text{def}}{=} \forall x.\phi(e, x)$ with a free variable e—i.e., the unknown function. The second constraint limits the syntactic structure f is allowed to have and is given as a set E of expressions specified by a context-free grammar defining a subset of all the terms in T. A solution to the SYGUS problem is an expression f in E such that the formula $\psi(f)$ is valid.

***Inverting functions with* SYGUS** Definition 5.2 gives us a natural way to encode the problem of computing the function $\bar{g}$ as a SYGUS problem. Assume that we have a transition $t = (p, \ell, \varphi, \bar{f}, q)$ and want to generate one inverted transition $t^{-1} = (p, k, \psi, \bar{g}, q)$. We showed how to compute the guard ψ, but we need a list of functions $\bar{g} \overset{\text{def}}{=} [g_1, g_2, ..., g_l]$ such that $\forall \bar{x}.\varphi(\bar{x}) \rightarrow (\bar{g}(\bar{f}(\bar{x})) = \bar{x})$. We observe that the output functions g_i are independent from each other and refine the specification for each function g_i as $\forall \bar{x}.\varphi(\bar{x}) \rightarrow (g_i(\bar{f}(\bar{x})) = x_i)$. We now have a separate SYGUS semantic constraint for each function g_i. Since we do not know what terms the function g_i might need, the SYGUS syntactic constraint for g_i is the context-free grammar describing all the terms in the alphabet theory.[2]

Example 6.1. *The following* SYGUS *problem asks to synthesize the function $g_0(y_0, y_1) : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$, which represents the first output component of the transition r in Example 5.3. The semantic constraint ensures that r inverts the transition t.*

$$\psi(g_0) \overset{\text{def}}{=} \forall(x_0, x_1).x_0 > 0 \wedge x_1 > 0 \rightarrow g_0(x_0{+}5, x_1{+}5) = x_0$$

A possible syntactic constraint for g_0 is the following context-free grammar.

$$\begin{array}{ll} \textit{Start} ::= \textit{(Start + Start)} \mid \textit{(Start − Start)} \mid \textit{Var} \mid \textit{Const} & \\ \textit{Var} ::= y_0 \mid y_1 \qquad \textit{Const} ::= 0 \mid 1 \mid 2 \mid 5 \mid 6 & \end{array}$$

A solution to this problem is $g_0(y_0, y_1) = y_0 - 5$.

GENIC *optimizations* We design two techniques that are specific to the problem of inverting functions in GENIC and use them to improve the performance and the result quality of GENIC. We evaluate both these techniques in Section 7.

The goal of our first technique is to use auxiliary functions to generate GENIC programs that are small in size and natural to read. We use SYGUS to invert the auxiliary functions that are defined in the preamble of the GENIC program and then add these functions to the SYGUS syntactic constraint when trying to invert transitions. For example, the GENIC program in Figure 2 has two auxiliary functions (lines 2-6). GENIC checks which of these functions are injective and it computes their inverses using SYGUS. After synthesizing the inverses of the auxiliary functions, GENIC produces new SYGUS instances for synthesizing the transitions. This time, the SYGUS solver is provided with an enriched grammar that allows the synthesizer to produce terms containing the synthesized auxiliary functions and the auxiliary functions of the original program. This optimization also allows GENIC to produce more succinct programs.

Our second technique aims at reducing the size of the SYGUS grammar and therefore the search space. Consider the following transition and its inverse:

$$t = p \xrightarrow[2]{x_1 \geq 0, x_2 \geq 0 / [x_1 + x_2, x_1]} q$$
$$t^{-1} = p \xrightarrow[2]{y_1 \geq y_2, y_2 \geq 0 / [y_2, y_1 - y_2]} q.$$

First, since the functions appearing in t only use the plus operator, their inverses only require the minus operator. We can use this observation to simplify what operators we allow in the SYGUS grammar. Second, we observe that the first function appearing in t^{-1} only uses the variable y_2 as input. Intuitively, the variables y_1 is *enough* to recover the original value of x_1 because the function $f_2(x_1, x_2) = x_1$ in the output of t is injective on x_1 and constant—i.e., does

[2] Even though all numerical constants can be derived using the terms +, -, 0, and 1, to simplify the search, we also add all the constants appearing in the input program to the grammar given a syntactic constraint.

family	program	states	trans	auxFun	max ℓ	size (bytes)	isDet (secs)	isInj (secs)	inversion total	inversion max-tr	res	theory
BASE64	encoder	2	4	2	3	971	0.05s	2.20s	9.32s	5.18s	✓	BitVec 8
	decoder	2	4	3	4	1454	0.14s	2.92s	33.66	19.24s	✓	BitVec 8
mod BASE64	encoder	2	4	2	3	954	0.03s	2.28s	10.30s	6.06s	✓	BitVec 8
	decoder	2	4	3	4	1396	0.08s	2.73s	34.43s	21.64s	✓	BitVec 8
BASE32	encoder	2	6	2	5	1735	0.19s	6.45s	20.55s	9.06s	✓	BitVec 8
	decoder	2	6	3	8	1570	0.18s	4.66s	138.46	53.05s	✓	BitVec 8
BASE16	encoder	2	2	2	1	473	0.03s	0.30s	2.10s	2.10s	✓	BitVec 8
	decoder	2	2	3	2	732	0.03s	0.15s	1.92s	1.13s	✓	BitVec 8
UTF-8	encoder	1	4	1	1	1974	0.17s	1.05s	80.17s	69.20s	3/4	BitVec 32
	decoder	1	4	1	4	1864	0.19s	0.86s	8.13s	3.57s	✓	BitVec 32
UTF-16	encoder	1	2	0	1	1436	0.06s	0.64s	31.19s	30.56s	✓	BitVec 32
	decoder	1	2	0	2	1279	0.12s	0.87s	3.17s	2.72s	✓	BitVec 32
UU	encoder	2	4	2	4	862	0.03s	2.85s	6.14s	4.06s	✓	BitVec 8
	decoder	2	4	3	3	1258	0.07s	2.95s	24.16s	18.56s	✓	BitVec 8

Table 1: Performance and effectiveness of GENIC on 14 encoders and decoders. The column **total** shows the total time to invert the program, while the column **max-tr** shows the maximum time of inverting a single transition. In the column res, the symbol ✓denotes that all transitions were inverted. For the UTF-8 encoder, GENIC could only invert 3 transitions out of 4 and we report the total time taken to invert the 3 transitions.

not depend—on x_2. In general, when inverting a transition $p \xrightarrow[\ell]{\varphi(\bar{x})/[f_1(\bar{x}),\ldots,f_k(\bar{x})]} q$, each function in p^{-1} might only require a subset of the variables $\{y_1, ..., y_k\}$. Intuitively, a subset of the variables $\bar{y}^* = \{y_{i_1}, \ldots, y_{i_j}\} \subseteq \{y_1, ..., y_k\}$ is *enough* for recovering x_i if $\bar{f}^* = [f_{i_1}, \ldots, f_{i_j}]$ is injective with respect to $\bar{y}^*$ and constant with respect to $Y \setminus \bar{y}^*$. With abuse of notation we use $\bar{a}^*$ to denote an assignment to the variables in $\bar{y}^*$, $\bar{c}^{\#}$ to denote an assignment to the variables in $Y \setminus \bar{y}^*$, and $f(\bar{a}^*, \bar{c}^{\#})$ (resp. $\varphi(\bar{a}^*, \bar{c}^{\#})$) to denote the result of substituting the variables in $\bar{y}^*$ with $\bar{a}^*$ and the variables in $Y \setminus \bar{y}^*$ with $\bar{c}^{\#}$ in f (resp. φ). Formally, the set of variables $\bar{y}^*$ is enough for recovering x_i iff for every $\bar{a}^*, \bar{b}^*, \bar{c}^{\#}, \bar{d}^{\#}$, such that $\bar{a}^* \neq \bar{b}^*$, then the following implications hold:

$$\varphi(\bar{a}^*, \bar{c}^{\#}) \wedge \varphi(\bar{b}^*, \bar{c}^{\#}) \to \bar{f}^*(\bar{a}^*, \bar{c}^{\#}) \neq \bar{f}^*(\bar{b}^*, \bar{c}^{\#}) \quad (1)$$

$$\varphi(\bar{a}^*, \bar{c}^{\#}) \wedge \varphi(\bar{a}^*, \bar{d}^{\#}) \to \bar{f}^*(\bar{a}^*, \bar{c}^{\#}) = \bar{f}^*(\bar{a}^*, \bar{d}^{\#}) \quad (2)$$

Equation 1 guarantees that $\bar{f}^*$ is injective with respect to $\bar{y}^*$, while equation 2 guarantees that $\bar{f}^*$ is constant with respect to $Y \setminus \bar{y}^*$. Using this definition, we can exhaustively search for a minimum set of variables $\bar{y}^*$ needed to synthesize each output function. This set might not be unique. The variable reduction procedure does not sacrifice completeness, but, in general, reducing the SYGUS grammar may prevent the existence of inverse functions. In practice, we run the synthesis algorithm in parallel with and without optimization.

7. Evaluation

We now describe the implementation details of GENIC and evaluate its effectiveness and performance on a comprehensive set of benchmarks. The experiments were run on an Intel Core i7 4.00GHz CPU with 32 GB of RAM.

Implementation GENIC is written in JAVA, and uses the symbolic automata library SVPAlib [6] for the required automata operations. The alphabet theories of bit-vector and linear integer arithmetic are implemented using the SMT solver Z3 [10] and are also the only theories supported by existing SYGUS solvers. We experimented with all the SYGUS solvers from SYGUS-comp 2014, 2015, and 2016 [3] and chose the Enumerative CEGIS solver[3], the winner of the 2014 competition, as our SYGUS solver.[4] The other solvers were either slower or did not support the full SYGUS syntax.

Benchmarks We evaluate our technique on both real programs and artificial benchmarks. All programs are deterministic. First, we assess whether GENIC can invert 14 efficient bit-vector implementations of string encoders and decoders that are commonly used in networking. The set of considered coders can be found in Table 1. All these programs operate over the theory of bit-vectors and represent characters as groups of bytes. The BASEX encodings are binary-to-text encoding schemes that represent binary data in ASCII string format. The modified BASE64 encoder is the special version of the BASE64 encoding mentioned in Section 2. The UTFX encodings translate Unicode characters int groups of bytes of fixed length. The UU encoding is similar to BASE64. The programs have 0 to 3 auxiliary functions, 1 to 2 states, 2 to 6 transitions, and sizes varying between 473 and 1974 bytes.

Second, we use an artificial set of 17 programs of varying size to evaluate how the number of states and transitions in the program affects the performance of GENIC. These programs operate over the theory of linear integer arithmetic and are described in Section 7.2.

[3] Available at `https://github.com/abhishekudupa/sygus-comp14`.

[4] The solver CVC4, which won the 2015 competition, was also able to handle several of our benchmarks. However, it was slower than Enumerative CEGIS and produced very large and unreadable results, especially for the theory of bit-vectors.

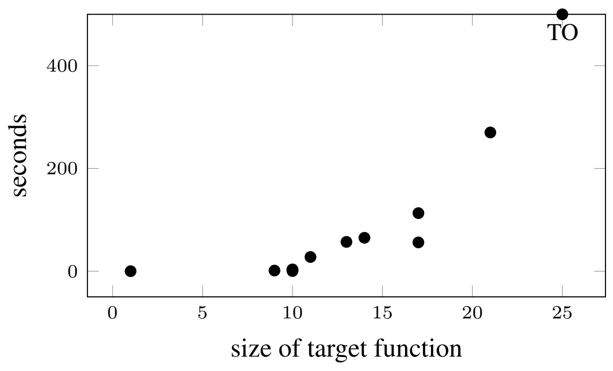

Figure 4: Synthesis time vs size of target functions.

7.1 Effectiveness of GENIC

In this experiment we evaluate how effective GENIC is at inverting real string coders. The results are shown in Table 1. The timeout for inverting a transition was set to 20 minutes.

GENIC successfully inverted 13 programs out of 14 and could prove injectivity and determinism for all the 14 programs. Checking determinism took less than 0.2 seconds per program (avg: 0.1s) and each injectivity check took less than 10 seconds (avg: 2.2s). Inverting programs is the most costly operation. GENIC took between 2 and 138 seconds (avg: 25s) to invert each program and failed on one program. For the UTF-8 encoder, GENIC was only able to invert 3 transitions out of 4 because the failing transitions required synthesising an inverse functions that contained 25 operators and terms. Currently, this size is beyond the reach of existing SYGUS solvers.

Although Theorem 5.4 shows that the output of GENIC can be a nondeterministic unambiguous program, in this experiment, GENIC always produced a deterministic program.

As we observed in Section 5, our inversion algorithm is amenable for parallelization because each transition can be inverted independently. For each program we report the maximum time it took to invert a single transition (max-tr) and observe that, in many cases, a single transition dominates the total runtime. Despite this fact, inverting each transition in parallel yields a 1.69x average speedup.

7.2 Detailed Evaluation

We now discuss and evaluate quantitative aspects of our algorithms.

Size of inverted functions In Section 7.1 we mentioned that the SYGUS solver could not synthesize functions with more than 25 operators and terms. In this experiment we report the runtime of each call to the SYGUS solver performed during the experiments showed in Table 1 and measure how the performance of SYGUS degrades when the size of the inverse function increases. The results are shown in Figure 4. In general, we can observe an exponential trend in the size of the target function. This fact further shows how GENIC

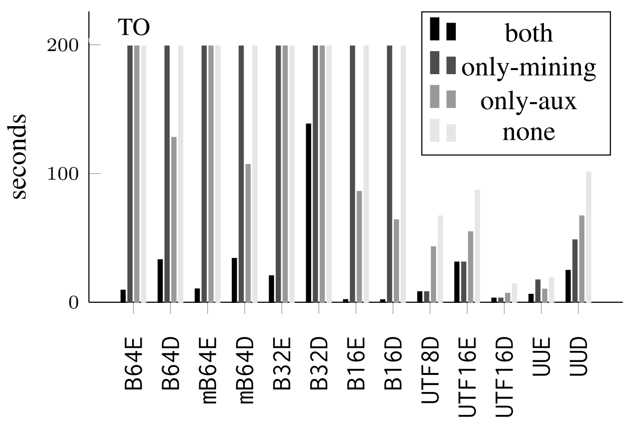

Figure 5: Inversion time with and without optimizations. only-aux indicates that only the optimization that synthesizes inverse of auxiliary functions is used, only-mining indicates that only the optimization that simplifies the grammar given to the SYGUS solver was used, both indicates that both optimizations were used, and none that no optimization was used. A line reaching 200 indicates a timeout.

is able to break the large synthesis problem of inverting a program into small synthesis problems that are within reach of existing SYGUS solvers.

Impact of optimizations We evaluate the impact of the optimizations we discussed at the end of Section 6. Figure 5 shows the time taken to invert the programs in Table 1 when using all optimizations (all), only the technique for synthesizing auxiliary functions (only-aux), only the technique for reducing the grammar given to the SYGUS solver (only-mining), and no optimization (none). GENIC can invert 13 programs when all optimizations are used. When both optimizations are turned off, GENIC can only invert 5 programs and this number does not change when we do not synthesize auxiliary functions. When we only enable the technique for synthesizing auxiliary functions, GENIC can invert 9 programs. This experiment motivates the need for both our optimizations and shows how the design of GENIC is beneficial in the synthesis process. In particular, thanks to the structure of GENIC programs, we are able to synthesize auxiliary functions which are shown to be crucial for performance.

Size of inverted programs Figure 6 compares the sizes of the programs generated by GENIC against the sizes of programs written by us. The sizes of the generated programs are, on average, 1.7 times larger than those written by us. There are two factors leading to the increased sizes of the generated programs. First, the predicates generated by the SMT solver and the functions generated by the SYGUS solver are typically not minimal. Second, sometimes the synthesized programs do not use the auxiliary functions in places where these functions could help reducing the size of the programs. Despite this fact, the programs generated by

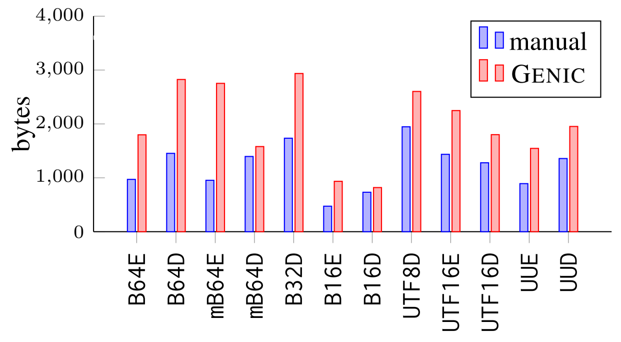

Figure 6: Sizes of manually written programs and programs produced by GENIC with and without using auxiliary functions. ⁑ indicates a timeout.

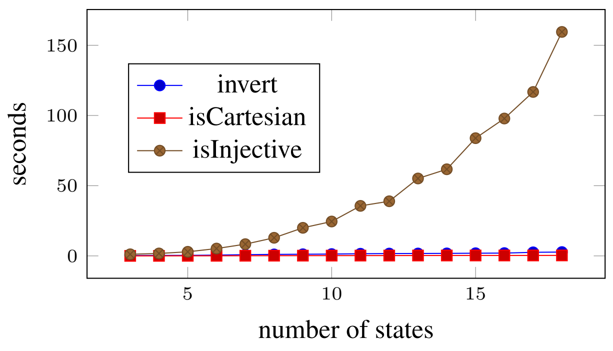

Figure 7: Running time for checking injectivity, computing the inverse, and performing the Cartesian check over the output language for the functions in ST.

GENIC are comparable in size to those we manually wrote. In addition, we found that the generated programs were easy to understand.

Number of states and transitions We consider a set of synthetic benchmarks to illustrate how the different components of our inversion procedure scale when varying the number of states and transitions in the program. The GENIC programs in the set $ST = \{S_2, \dots, S_{18}\}$ operate over lists of integers and are described as follows. Each program S_k, contains $k+1$ states $\{q_0, \dots, q_k\}$. and $2k$ transitions with lookahead 3. For every state q_i, such that $0 \le i < k$, the program contains two transitions of the form:

$$q_i \xrightarrow[3]{\lambda x_1,x_2,x_3.x_1=0/[x_1,x_2+c_i,x_3+d_i]} q_i, \text{ and}$$

$$q_i \xrightarrow[3]{\lambda x_1,x_2,x_3.x_1=1/[x_1,x_2+c_i,x_3+d_i]} q_{i+1}$$

where c_i, d_i are some constants in $\mathbb{Z}$.

Figure 7 shows the running times for checking injectivity and for computing the inverse for the programs in ST. Note that, in this experiment, the number of states is proportional to the number of transitions. The time taken to check injectivity aligns with the quadratic complexity we proved in Corollary 4.16. Figure 7 also shows that the fraction of the time spent computing the Cartesian predicates corresponding to the outputs of each transition is negligible and is proportional to the number of transitions. Since all transitions have comparable complexity, the time taken to invert each program scales linearly with the number of transitions. These trends align with the theoretical complexities of our decision procedures.

7.3 Limitations

We showed that the tool GENIC is able to automatically and efficiently invert 13 out of 14 real world encoders and certain classes of programs operating over lists of integers. We now describe some limitations of our tool, in particular with respect to the programs we could not model or invert.

Functions with large arity We showed how existing SYGUS solvers can only synthesize functions of limited sizes and complexity. In particular, functions of size greater than 25 are beyond the capabilities of existing solvers. However, given the continuous advances in SYGUS solvers [3], we believe that soon GENIC will be able to synthesize more programs, and invert the UTF-8 encoder transition we currently cannot invert.

Expressiveness The main structural limitation of GENIC is that it can only invert programs that are expressible as s-EFTs. While this class captures many interesting programs such as string encoders, CSV file transformations, and certain functional programs that operate over lists, many other programs we would like to invert cannot be modeled as s-EFTs. For example, XML and JSON transformations operate over hierarchical structures, while compression algorithms like LZ77 transform the input in two passes: the first pass produces a dictionary of all the words in the input, and the second pass uses the dictionary rewrites the input. Similarly, several network format transformations use checksums and fields to declare the length of the payload. These transfomations require the ability to model trees and an infinite number of states—e.g., registers. Defining transducer models that can capture these types of behavior and use them to extend GENIC is an interesting and challenging research direction. We will expand on these extensions in related work.

Limited theories GENIC's currently supports only theories that admit quantifier elimination. This operation is used to efficiently compute the output predicates of each transition, predicates that are then used to check injectivity and to produce the inverted transitions. Although our current implementation is based on quantifier elimination, it is possible to use SYGUS to synthesize not only the functions, but also the predicates appearing on each transition. This technique is likely to have performance limitations, but it could be used to extend GENIC to decidable first-order theories that do not admit quantifier elimination.

8. Related Work

Symbolic automata and transducers Symbolic finite automata (s-FAs) and transducers (s-FTs) were formalized in [15] with a focus on analysis of sanitizers. The extended model we studied, s-EFT, was designed to analyze string encoders [8]. The main result in [8] is that equivalence is decidable for Cartesian s-EFTs and undecidable in the general case. An algorithm for checking whether a predicate is Cartesian is also proposed in [8]. The same paper presents monadic s-EFTs, which are equivalent in expressiveness to Cartesian s-EFTs and allow transition guards to be finite disjunction of Cartesian predicates. In general, it is undecidable to check whether a predicate is monadic [23]. Our paper builds on some of these results, but, to our knowledge, the questions of checking s-EFT injectivity and automatically inverting s-EFTs have not been studied before. Veanes et al. also proposed a variant of s-EFTs called k-s-FTs. This model reflects the semantics of ML-style pattern-matching [24]—i.e., programs are deterministic because rules are evaluated in a given order. k-s-FTs do not enjoy good closure properties and are not a good target model for GENIC.

Transducer inversion Finite state transducers (FST), which operate over *finite alphabets*, enjoy closure properties that s-EFTs do not enjoy. In fact, the problems of checking injectivity and producing an inverse are trivial for FSTs. Injectivity has also been studied for more complex transducer models, such as deterministic tree transducers [4, 13].

To our knowledge, we are the first to study injectivity checking and automatic inversion of symbolic transducers. Our approaches are novel in two ways. First, we show that checking injectivity for deterministic s-EFTs is harder—in fact undecidable—than checking injectivity for deterministic FSTs. We then present a decidable fragment of s-EFTs for which injectivity is decidable. Second, while the transduction relation computed by an FST can always be inverted, this is not the case for s-EFTs because the functions in the alphabet theory might not admit a closed form inverse.

Moreover, our proposed paradigm is foundational, general, and it can be applied to other transducers models. For example, applying our paradigm to Symbolic Tree Transducers [9] will allow us to invert programs that operate over trees and general recursive data-types, instead of just lists—e.g., XML and JSON transformations. Similarly, models like streaming transducers [1] will enable inversion of programs for which the transformations depends on the input data—e.g., translations of network packets for which the length is a function of an element of the input packet.

Automatic program inversion Dijkstra was the first one to investigate the problem of automatically inverting programs and manually inverted simple programs operating over arrays [11]. Since then, new inversion techniques have been proposed, but they are either only effective on simple programs [5, 12, 14, 16–18, 20] or are only *semi*-automated [22]. Moreover, some of these techniques do not provide guarantees on the correctness of their results. We detail our comparison in the following paragraphs.

Logic programming has been used to encode the semantics of simple programs and compute their inverse [18]. Relational calculus and deductive reasoning were used to derive inverses of simple tree traversals [5, 20]. Unlike these approaches, we focus on different, well-defined, classes of programs and on providing algorithmic foundations for such classes of programs. Formal techniques such as LR parsing and context-free grammars have been used to invert very small programs [14, 17]. In these cases, the formalisms were used as algorithmic tools while we take a foundational approach and ask whether programs represented using s-EFTs, a well defined class, can be automatically inverted. Alur et al. used testing techniques to synthesize inverse of a restricted class of programs mapping arrays to arrays in an iterative manner [17]. We tackle a different class of programs and propose algorithms with well-defined formal guarantees.

Srivastava et al. combine templates and inductive synthesis to design a *semi-automated* tool, PINS, for automatically inverting imperative programs [22]. PINS requires the user to provide templates and sometimes modify expressions in the inverted programs and is therefore a fairly general tool that is able to invert programs that are not expressible in GENIC. Our approach is different in the following aspects. First, our method is *completely* automatic and does not require the user to provide any information other than the input program. Second, our approach guarantees that the resulting program is indeed the inverse of the input one. Instead, PINS only tests the produced result and ultimately requires the programmer to manually inspect multiple inverted solution. Third, GENIC is grounded in formal methods and it can, for example, prove injectivity of the input program while PINS cannot. Fourth, on the programs both PINS and our tool can invert, GENIC is 10 to 100 times faster—e.g, 9 vs 1,300 sec for BASE64.

9. Conclusion

We presented the tool GENIC that automatically inverts functional programs operating over lists containing elements from complex domains-e.g., integers and bit-vectors. GENIC is grounded in the theory of extended symbolic finite transducers, s-EFT, an expressive formal model that can capture complex list-to-list transformations. We provided algorithms for checking injectivity of s-EFTs and for automatically inverting them. Using these algorithms, GENIC can automatically produces *correct* inverses for complex programs such as BASE64 and UTF-8 encoders and decoders.

Acknowledgements

Research funded by the Univ. of Wisconsin, Fall Research Competition 2015. We thank Margus Veanes for the insightful conversations that lead to this paper and the anonymous reviewers for their feedback.

References

[1] R. Alur. *Streaming String Transducers*, pages 1–1. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[2] R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design (FMCAD), 2013*, pages 1–8. IEEE, 2013.

[3] R. Alur, D. Fisman, R. Singh, and A. Solar-Lezama. Results and analysis of SyGuS-comp'15. *arXiv preprint arXiv:1602.01170*, 2016.

[4] M. Benedikt, J. Engelfriet, and S. Maneth. Determinacy and rewriting of top-down and mso tree transformations. In *International Symposium on Mathematical Foundations of Computer Science*, pages 146–158. Springer, 2013.

[5] W. Chen and J. T. Udding. Program inversion: More than fun! *Science of Computer Programming*, 15(1):1–13, 1990.

[6] L. D'antoni and R. Alur. Symbolic visibly pushdown automata. In *International Conference on Computer Aided Verification*, pages 209–225. Springer, 2014.

[7] L. D'Antoni and M. Veanes. Equivalence of extended symbolic finite transducers. In *International Conference on Computer Aided Verification*, pages 624–639. Springer, 2013.

[8] L. D'Antoni and M. Veanes. Extended symbolic finite automata and transducers. *Formal Methods in System Design*, 47 (1):93–119, 2015.

[9] L. D'Antoni, M. Veanes, B. Livshits, and D. Molnar. Fast: A transducer-based language for tree manipulation. In *ACM SIGPLAN Notices*, volume 49, pages 384–394. ACM, 2014.

[10] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[11] E. Dijkstra. Program inversion. *Program Construction*, pages 54–57, 1979.

[12] D. Eppstein. A heuristic approach to program inversion. In *IJCAI*, volume 85, pages 219–221, 1985.

[13] Z. Fueloep and P. Gyenizse. On injectivity of deterministic top–down tree transducers. *Information processing letters*, 48 (4):183–188, 1993.

[14] R. Glück and M. Kawabe. A method for automatic program inversion based on lr (0) parsing. *Fundamenta Informaticae*, 66(4):367–395, 2005.

[15] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and precise sanitizer analysis with BEK. In *Proceedings of the 20th USENIX conference on Security*, pages 1–1. USENIX Association, 2011.

[16] A. Kanade, R. Alur, S. Rajamani, and G. Ramanlingam. Representation dependence testing using program inversion. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 277–286. ACM, 2010.

[17] M. Kawabe and R. Glück. The program inverter LRinv and its structure. In *International Workshop on Practical Aspects of Declarative Languages*, pages 219–234. Springer, 2005.

[18] B. J. Ross. Running programs backwards: the logical inversion of imperative computation. *Formal Aspects of Computing*, 9 (3):331–348, 1997.

[19] SANS. Malware faq. `https://www.sans.org/security-resources/malwarefaq/wnt-unicode`.

[20] B. Schoenmakers. Inorder traversal of a binary heap and its inversion in optimal time and space. In *Mathematics of Program Construction*, pages 291–301. Springer, 1993.

[21] F. Servais and J.-F. Raskin. Visibly pushdown transducers. *ULB, Belgique*, 2011.

[22] S. Srivastava, S. Gulwani, S. Chaudhuri, and J. S. Foster. Path-based inductive synthesis for program inversion. In *ACM SIGPLAN Notices*, volume 46, pages 492–503. ACM, 2011.

[23] M. Veanes, N. BjØrner, L. Nachmanson, and S. Bereg. Monadic decomposition. In *International Conference on Computer Aided Verification*, pages 628–645. Springer, 2014.

[24] M. Veanes, T. Mytkowicz, D. Molnar, and B. Livshits. Data-parallel string-manipulating programs. In *ACM SIGPLAN Notices*, volume 50, pages 139–152. ACM, 2015.

[25] D. M. Yellin. *Attribute Grammar Inversion and Source-to-source Translation*. Springer-Verlag New York, Inc., New York, NY, USA, 1988. ISBN 0-387-19072-4.