Reverse Template Processing using Abstract Interpretation

Matthieu Lemerre[0000−0002−1081−0467]

Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France
matthieu.lemerre@cea.fr

Abstract. Template languages transform tree-structured data into text. We study the reverse problem, transforming the template into a parser that returns all the tree-structured data that can produce a given text. Programs written in template languages are generally not injective (they have multiple preimages), not affine (some input variables can appear at several locations in the output), and erasing (they provide only a partial view of the source), which makes the problem challenging. We propose to solve this problem using concepts from abstract interpretation, like the denotational style of abstract semantics, soundness, exactness, or reduction, to reason about the precision and the recovery of all the preimages. This work shows that Abstract Interpretation is a very useful theory when reasoning about the reversal of non-injective programs.

1 Introduction

One of the most ubiquitous ways to format data into text is through the use of a *template engine,* or *template processor*. They interpret a program in a *template language*, which consists in a fixed text intertwined with specific *instructions* that produces an *output text*, given tree-structured data as an *input*. The input data often comes from XML, JSON, relational databases, or records from a language providing the data. Examples of such template engines include Apache Freemarker, Mustache, ERB, Jinja, Liquid, XSLT, or StringTemplate [38].

We study the reverse transformation, i.e. the problem of retrieving the input data from the output text. In general, the problem of finding if a program p admits an input that produces a given output w is undecidable. This is the case for the template language that we want to address, as e.g. inversing a template can be used to find the solutions of Diophantine equations, an undecidable problem [28]. Thus, we cannot have an algorithm which is simultaneously *sound* (i.e. does not forget about an input), *complete*, (i.e. does not return incorrect inputs), *terminating*, and working on an *expressive* template language. Another problem is that of *finite representation* of these inputs: if the transformation is not injective (which is rarely the case, as the output is often only a partial view of the input), the set of corresponding inputs can be large or even infinite; we thus want a *finite* representation of this infinite set.

Our solution uses abstract interpretation [8] to solve the reverse transformation of template processing of tree-structured data. Using abstract domains to finitely

represent an infinite set of JSON-like databases, our template reversing algorithm is terminating (Section 4), sound (Section 5), and works on an expressive template language (which can be easily extended, as shown in Sections 7 and 8). What we have to give up is completeness; however, most of our operations are complete, and our algorithm can detect if the set of values that it returns is exact or is an over-approximation (Section 6). Our experiments (Section 8) show that, on practical examples, it is effectively able to retrieve the part of the database that was used to produce a given output.

Specifically, our main contribution is a technique to invert a function by

1. deriving a backward denotational semantics from the forward big-step semantics of the language, with modifications such that the reverse algorithm will need to explore only a finite number of evaluation trees (Section 4);
2. using abstract interpretation [8] on this denotational semantics to derive a sound algorithm (Section 5), but also to reason about the precision (Section 6), simplification by constraint propagation (Section 7), and extensions (Section 8) of the algorithm.

While we show that template languages are a good fit for applying these techniques, they are quite general and could be reused in other areas of reverse computation[1].

2 Motivation and example

Our original motivation comes from the following practical problem. In embedded systems, automated generation of code and data in source code is very common, as this minimizes the storage and execution costs in the runtime. Template languages are used to simplify the formatting of this source code.

If it is part of a safety-critical application, this generated source code must be certified. One strategy for this is to certify the code generator. Because an error in this generator can lead to an error when the system runs, such a generator is considered to be critical component which must be qualified at the highest assurance level, which is very costly.

An alternative certification strategy [33] is to develop an *independent checker* that verifies that the generated source code corresponds to safety requirements, independently of how the source was generated (this is similar to the translation validation [40] approach found e.g. in CompCert [26]).

Because an error in such a checker cannot introduce new errors (it can only omit to see existing errors), such a checker is less critical than a code generator and needs only to be qualified at a low assurance level, which is much less costly. The code generator may not even have to be qualified at all, as the certification works on the generated code, independently of its provenance.

Figure 1 represents all the steps of this use-case: on the left, a team of developpers create (manually or automatically) a JSON database containing system parameters from a high-level description of the system. Template processing is

[1] In particular, reverse computation normally focus on injective function, while our technique does require this limitation.

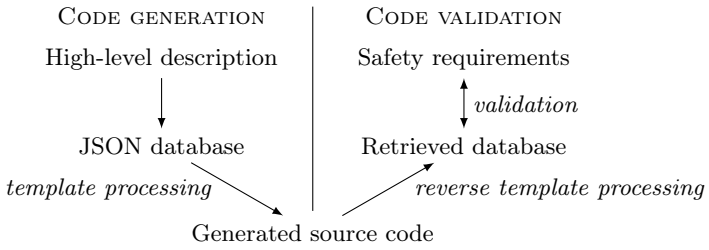

Fig. 1. Verifying generated source code, featuring reverse template processing.

then used to generate source code using these parameters. On the right, the verification team uses the checker first to retrieve a database by parsing the generated source code; this retrieved database can be used to check that the parameters comply with safety requirements. One may wonder why we cannot just reuse the generated JSON database directly: the reason is that this would be considered a common cause of failure by certification authorities, as the verification process must be independent and cannot rely on intermediate artefacts produced by the code generation. On the other hand, the templates can be reused, because they can be viewed as a specification on how the source code has to be generated.

Since such a checker must work from the source code of the safety-critical application, writing a parser extracting the data to validate from the source code is a cumbersome task. When the source code has been produced using templates, our idea is that the parsing can be done by *reversing the template processing* – i.e. evaluating the template backwards, to retrieve the input data from the output text. We can then validate that this retrieved data matches the safety requirements of the system. Figure 1 represents all the steps of this use-case.

Let us examine a concrete example inspired by the actual industrial problem that motivated this work. Consider this example of drones monitoring a sea area from a location in Bermuda. Each drone is assigned a flight plan, which is a sequence of coordinates. Each drone has limited memory, thus the flight plan is generated as a static linked list in source code which is compiled and put inside each drone. To make the flight plan more maintainable, the generation is split in two phases: first, *parameters* are computed and stored in a JSON database; second, source code is generated using a template processor. Figure 2 represents the template generating this linked list, the input parameters, and the resulting code, which can be compiled and linked to the drone autopilot.

Now imagine that the drone must be certified according to safety-critical aeronautics standards. The computation of the flight plan parameters, having to satisfy many antagonistic constraints, is a hard problem (encompassing the traveling salesman), and certifying the code generator producing this flight plan would be hard. Luckily, certifying the code generator is not mandatory, as only the generated code must be certified (independently of how it was generated). The safety requirements on the flight plan code are actually quite simple: (1) the code must provide this data in a format that is suitable for the drone runtime,

struct coord { **float** lat; **float** lon; **struct** coord ∗next;};
⟨**for** *c* **in** *seq*⟩**struct** coord ⟨= *c.name*:*symbol* ⟩;⟨**end**⟩
struct coord ∗start = &⟨= *first*:*symbol* ⟩};

⟨**for** *c* **in** *seq*⟩
  struct coord ⟨= *c.name*:*symbol* ⟩ =
  { ⟨= *c.lat*:*float* ⟩, ⟨= *c.lon*:*float* ⟩, ⟨**if** *c.last*⟩ 0 ⟨**else**⟩ &⟨= *c.next*:*symbol* ⟩ ⟨**end**⟩ };
⟨**end**⟩

$$\left[\begin{array}{ll} first \mapsto hamilton & \\ seq \mapsto & \left[\begin{array}{ll} name \mapsto hamilton \\ lat \mapsto 32.36 \\ lon \mapsto -64.67 \\ last \mapsto false \\ next \mapsto san\_juan \end{array}\right] :: \left[\begin{array}{ll} name \mapsto san\_juan \\ lat \mapsto 18.46 \\ lon \mapsto -66.10 \\ last \mapsto false \\ next \mapsto miami \end{array}\right] :: \left[\begin{array}{ll} name \mapsto miami \\ lat \mapsto 25.76 \\ lon \mapsto -80.19 \\ last \mapsto true \end{array}\right] \end{array}\right]$$

---

struct coord { **float** lat; **float** lon; **struct** coord ∗next;};
struct coord hamilton; **struct** coord san_juan; **struct** coord miami;
struct coord ∗start = &hamilton;

struct coord hamilton = { 32.31, −64.76, &san_juan };
struct coord san_juan = { 18.46, −66.10, &miami };
struct coord miami = { 25.76, −80.19, 0; };

Fig. 2. Forward evaluation of the template (top) on the input environment (middle) yields the output text (bottom).

something which can be specified using the template; (2) the length of the route must be small enough so that the drone does not run out of battery and fall.

To validate the parameters, we cannot just reuse the generated JSON database directly, as this would be considered a common cause of failure by certification authorities. The templates can be reused, as they can be considered as a specification on how the source code has to be formatted. Thus, a suitable strategy to certify compliance of the flight plan source code is to start from the code, and to: 1. Verify that this source code complies with the specified template (this checks safety requirement (1)), and retrieve all the possible input parameters (the JSON databases) that can produce the source code using the template; 2. Verify that each possible input parameters satisfies the safety requirement (2).

Step 2. is application-dependent (Methni et al. [33] provides several examples, such as verifying that communication buffers have sufficient size, that a static schedule plan allows meeting every deadline, etc.), but we propose to automatize step 1. using *reverse template processing*.

3 Background: the RTL template language

We study the problem of reversing templates on a simple functional template language, whose syntax and semantics is given here. While simple, this language is quite representative of how typical templates are developed; moreover, we propose extensions to this language in Section 8.

$$
\begin{array}{rll}
\mathbb{P} \ni p, q & \triangleq x \mid p.f & \text{path} \\
\mathbb{X} \ni x & & \text{variable names} \\
\mathbb{F} \ni f & & \text{field names} \\
\mathbb{W} \ni u, v, w & & \text{word (string)} \\
\mathbb{T} \ni t & & \text{scalar types}
\end{array}
$$

$$
\begin{array}{rll}
\mathbb{I} \ni i, j & \triangleq w & \text{fixed word} \\
& \mid \langle = p{:}t \rangle & \text{replacement} \\
& \mid i\ i & \text{template concatenation} \\
& \mid \langle \texttt{if}\ p \rangle\, i\, \langle \texttt{else} \rangle\, i\, \langle \texttt{end} \rangle & \text{conditionals} \\
& \mid \langle \texttt{for}\ x\ \texttt{in}\ p \rangle\, i\, \langle \texttt{end} \rangle & \text{iterations} \\
& \mid \langle \texttt{apply}\ \phi\ \texttt{with}\ x = p \rangle & \text{function call}
\end{array}
$$

Fig. 3. Syntax for the RTL template language

Notations We write equality by definition as $\triangleq$, word concatenation as $\cdot$, and the empty word as ε. The power set of a set $\mathbb{S}$ is written as $\mathcal{P}(\mathbb{S})$, and the empty set is represented by $\{\}$. The domain of a function φ is represented by $\mathrm{dom}(\varphi)$. We represent by $Seq(\mathbb{S})$ the set of finite sequences of elements in $\mathbb{S}$, and represent a sequence of n elements by $e_0 :: \ldots :: e_{n-1}$. A sequence of n elements in $\mathbb{S}$ can also be considered as a function $\in [0..n-1] \to \mathbb{S}$.

We use $\mathbb{Y} \rightharpoonup \mathbb{Z}$ to represent the set of partial functions from $\mathbb{Y}$ to $\mathbb{Z}$. If $\Gamma \in \mathbb{Y} \rightharpoonup \mathbb{Z}$, $y \in \mathrm{dom}(\Gamma)$, and $z \in \mathbb{Z}$, we note by $\Gamma[x] \in \mathbb{Z}$ the value bound to x in Γ, and by $\Gamma[y \mapsto z] \in \mathbb{Y} \rightharpoonup \mathbb{Z}$ the replacement of the binding from y to z in Γ. We note by $[]$ a partial function with an empty domain, and by $[y \mapsto z]$ the partial function which binds only y to z (i.e. $[y \mapsto z] = [][y \mapsto z]$). Finally, we note by $\mathit{unbind}(y, \Gamma)$ the restriction of Γ where y is removed from $\mathrm{dom}(\Gamma)$. We represent anonymous functions and partial functions using λ-calculus notation.

Syntax The template language, whose syntax is given in Figure 3, manipulates a tree-structured data as its input, and values are referred by paths in that tree. A *path* $p \in \mathbb{P}$ is either a *variable* $x \in \mathbb{X}$, or access to a *field* $f \in \mathbb{F}$ of a record. A template is decomposed into *instructions* $i \in \mathbb{I}$. An instruction is either a fixed word (reproduced as is), a replacement (replaced by the printed representation of a value in a database), a concatenation of two instructions, a conditional, a loop, or a function call. For the sake of simplicity the functions here only take one argument, and we omit the syntax for the definition of functions, which is standard; in particular, recursive calls are allowed.

Databases A *database* $d \in \mathbb{D}$ is a *tree* defined inductively as either a *scalar* of type t, a *record* from field names to databases, or a *finite sequence* of databases.

$$\mathbb{D} \triangleq \mathbb{D}_s \cup \mathbb{D}_r \cup \mathbb{D}_* \qquad \mathbb{D}_s \triangleq \bigcup_{t \in \mathbb{T}} \mathbb{D}_t \qquad \mathbb{D}_r \triangleq \mathbb{F} \rightharpoonup \mathbb{D} \qquad \mathbb{D}_* \triangleq Seq(\mathbb{D})$$

Scalar types must include integer and boolean types. As we focus on source-code generation, our practical examples also use floating point numbers, symbols (sequence of alphanumeric letters), and quoted strings (delimited by `""`).

Environments An *environment* $\Gamma \in \mathbb{E} = \mathbb{X} \rightharpoonup \mathbb{D}$ is a mapping from variables to databases. For instance, the mapping in the middle of Figure 2 is an environment, mapping the variables *first* and *seq* to databases; *first* is a scalar (a *symbol*), and *seq* is a sequence of 3 records, each containing only scalar values.

$$\text{FIXED}\quad (\!|w|\!)(\Gamma) = w$$

$$\text{REPLACE}\quad \frac{\mathit{show}_t(\Gamma[p]) = w}{(\!|\langle{=}\ p{:}t\ \rangle|\!)(\Gamma) = w}$$

$$\text{CONCAT}\quad \frac{(\!|i|\!)(\Gamma) = u \qquad (\!|j|\!)(\Gamma) = v}{(\!|i\ j|\!)(\Gamma) = u \cdot v}$$

$$\text{IF-TRUE}\quad \frac{\Gamma[p] = \mathit{true} \qquad (\!|i|\!)(\Gamma) = w}{(\!|\langle\texttt{if}\ p\rangle\ i\ \langle\texttt{else}\rangle\ j\ \langle\texttt{end}\rangle|\!)(\Gamma) = w}$$

$$\text{IF-FALSE}\quad \frac{\Gamma[p] = \mathit{false} \qquad (\!|j|\!)(\Gamma) = w}{(\!|\langle\texttt{if}\ p\rangle\ i\ \langle\texttt{else}\rangle\ j\ \langle\texttt{end}\rangle|\!)(\Gamma) = w}$$

$$\text{APPLY}\quad \frac{i = \Delta(\phi) \qquad (\!|i|\!)(\Gamma[x \mapsto \Gamma[p]]) = w}{(\!|\langle\texttt{apply}\ \phi\ \texttt{with}\ x = p\rangle|\!)(\Gamma) = w}$$

$$\text{FOR}\quad \frac{\Gamma[p] = d_1 :: \ldots :: d_n \qquad \forall k \in 1..n : (\!|i|\!)(\Gamma[x \mapsto d_k]) = w_k}{(\!|\langle\texttt{for}\ x\ \texttt{in}\ p\rangle\ i\ \langle\texttt{end}\rangle|\!)(\Gamma) = w_1 \cdot\ \ldots\ \cdot w_n}$$

Fig. 4. Big-step operational semantics for RTL.

We extend our notation for variable access in environment, and field access in databases, to handle paths:

$$\Gamma[p.f] \triangleq \Gamma[p][f] \qquad \Gamma[p.f \mapsto v] \triangleq \Gamma[p \mapsto \Gamma[p][f \mapsto v]] \qquad [p.f \mapsto v] \triangleq [p \mapsto [f \mapsto v]]$$

Functions on scalars For every scalar type $t \in \mathbb{T}$, we suppose that there exists an injective function $\mathit{show}_t \in \mathbb{D}_t \to \mathbb{W}$ to convert the value into a string, and a partial function $\mathit{read}_t \in \mathbb{W} \rightharpoonup \mathbb{D}_t$ that does the reverse, i.e. their composition is the identity: $\forall s \in \mathbb{D}_t : \mathit{read}_t(\mathit{show}_t(s)) = s$. Note that read_t may fail (e.g. when attemptying to parse "foo" as a number).

Forward semantics We now give the normal (forward) semantics for our template language (Figure 4), in the big-step structural operational style of [22].

$(\!|\cdot|\!)$ is a partial function that takes a template instruction $i \in \mathbb{I}$ and an environment $\Gamma \in \mathbb{E}$, and either produces a word, or fails (when the template tries to access a path not bound in the environment Γ). The evaluation is deterministic.

The evaluation goes as follows: fixed words are copied as is (FIXED). Replacements are done using the value taken from the corresponding field in the database (REPLACE). Concatenated instructions produce a concatenated word (CONCAT). Conditionals depend on the boolean value of a path in the database (IF-TRUE and IF-FALSE); templates which use as a condition a value which is not a boolean fail with an error. The `for` instruction iterates over a sequence, binds each element of the sequence to a variable, evaluates the child instruction with this new environment, and concatenates the results (FOR). The `apply` instruction (APPLY) introduces a new binding which is used in the called function; we assume the existence of a global mapping of definitions Δ from function names to instructions created by parsing the function definitions (the "main" template is attached to the name `main`).

In general the evaluation will start with one or several variables bound, which are the "roots" of the database.

4 Semantics for reversing template

We define the *reverse template problem* for i and w as finding the set of all the environments that can produce a word w given a template i. Formally:

Definition 1 (Backward interpretation function). *Given a template i, the* backward interpretation function $[\![i]\!](\cdot)$ *is defined as follows:*

$$[\![i]\!](\cdot) : \mathbb{W} \to \mathcal{P}(\mathbb{E}) \qquad [\![i]\!](w) \triangleq \{\Gamma \in \mathbb{E} : (\!|i|\!)(\Gamma) = w\}$$

We derive a sound algorithm for the reverse template problem in 3 steps:

1. we observe that we can reverse the forward evaluation as a parsing algorithm, but this requires fixing an issue on some `for` template instructions;
2. we define a denotational (i.e. compositional) semantics for the backward interpretation function, that we modify to work around the parsing issue;
3. we define abstract domains that provide finite, computer-manipulable representations of infinite sets of environments, and we use them to derive an algorithm that computes a representation of a superset of $[\![i]\!](w)$.

This section describes the steps 1. and 2., while Section 5 presents step 3.

4.1 Parsing as natural deduction

To compute $[\![i]\!](\cdot)$, a key observation is that the operational semantics given in Fig 4 can be read backwards, as a proof search that parses the text according to the template, and use this to retrieve a Γ (this makes use of the correspondance between parse trees and deduction trees observed by Shieber et al. [46]).

This suggests an algorithm that would first use a parser to enumerate all the suitable deduction/parse trees, and solve the associated constraints for each tree (it suffices to translate the template into a context-free grammar that over-approximates the set of words that a template can generate, see Appendix A.1 for details).

Unfortunately, this algorithm fails when the number of parse trees is infinite, which can happen in our language. In Appendix A.1 we show that this happens when the context-free grammar corresponding to the template is cyclic, i.e. a non-terminal can derive into itself without outputing anything else. This can happen notably when a function ϕ may recursively call itself without outputing anything after and before the recursive call, e.g. when ϕ is defined as follows: "$\langle$`if` *p.cond*$\rangle$ $\langle =$ *p.data:int* $\rangle$ $\langle$`else`$\rangle$ $\langle$`apply` ϕ `with` $p = p.next\rangle$ $\langle$`end`$\rangle$". Note that non-cyclic recursive calls to template functions (e.g. if ϕ is defined as "$\langle$`if` *p.cond*$\rangle$ $\langle =$ *p.data:int* $\rangle$ $\langle$`else`$\rangle$ ($\langle$`apply` ϕ `with` $p = p.next\rangle$) $\langle$`end`$\rangle$") can be handled as in this case, there would still be a finite number of parse trees; further, we explain how to extend our technique to handle cyclic recursive calls to templates in Section 8.4). We never encountered such cyclic recursive calls in our real examples (and it is easy to add text in the output to make the recursive call non-cyclic), so in the following we assume that the template has no cyclic recursive calls.

On the contrary, the following pattern is very common. Consider, in Fig 4, the FOR rule: if the i instruction can produce the empty word ε, then this rule may have an arbitrary large number of antecedents. For instance, in the template:

$$\langle \texttt{for}\ n\ \texttt{in}\ nums\rangle\ \langle \texttt{if}\ n.odd\rangle\ \langle = n.id{:}int\ \rangle\ \langle \texttt{else}\rangle\ \langle \texttt{end}\rangle;\ \langle \texttt{end}\rangle$$

$$\textsc{For-alt}\quad \frac{\begin{array}{ll} & d_1{::}\ldots{::}d_n \text{ subsequence of } \Gamma[p] \\ \forall k \in 1..n : w_k \neq \varepsilon & \forall d : d \in \Gamma[p] \wedge d \notin d_1{::}\ldots{::}d_n \Rightarrow (\!|i|\!)(\Gamma[x \mapsto d]) = \varepsilon \\ & \forall k \in 1..n : (\!|i|\!)(\Gamma[x \mapsto d_k]) = w_k \end{array}}{(\!|\langle \texttt{for}\ x\ \texttt{in}\ p\rangle\ i\ \langle \texttt{end}\rangle|\!)(\Gamma) = w_1 \cdot\ \ldots\ \cdot w_n}$$

Fig. 5. Alternative rule for the `for` instruction

applied to the string "11;17;", there can be arbitrarily many n in *nums* which are not *odd*, each of them corresponding to a different parse tree. As in this case the number of parse trees is infinite, the proof search cannot terminate.

To work around this problem, we provide an alternative FOR-ALT rule (Fig 5, where *subsequences* are not necessarily consecutive). This alternative rule just separates the elements in the sequence that produce an empty string from those that do not, and it is thus easy to prove that both rules are equivalent.

Theorem 1. *The* FOR *and* FOR-ALT *rules are equivalent.*

Using this new rule, the children of a `for` node in a parse tree is now a finite sequence of subtrees each corresponding to a non-empty word, instead of an arbitrary long sequence of subtrees corresponding to either empty or non-empty words (which could thus not be enumerated). The combination of the FOR-ALT rule and restriction on non-cyclic recursive calls makes sure that the number of possible parse trees for any given word is finite, and can thus be enumerated (see Appendix A.1 for a proof).

4.2 A backward denotational semantics

Using Definition 1, we can easily prove that the backward interpretation function $[\![]\!]()$ has the properties given in Fig. 6. But these equations can also act as a denotational semantics for $[\![]\!]()$, i.e. they define the behaviour of $[\![]\!]()$ constructively. To summarize, outputing a fixed word is possible only when the word matches its output; outputing a path constrains this path in the environment; when two templates are concatenated, the environment must fullfill the constraints of both templates; in conditionals the environments must fulfill one of the two cases; as applying a template creates a binding from the formal to actual argument, the reverse evaluation has to perform the opposite operation of unbinding the constraints; finally, handling `for` loops is a combination between concatenation of a finite sequence and unbinding of the loop iterator.

As we explained in Section 4.1, this semantics cannot be used "as-is" as the basis for an algorithm, because the `for` rule can perform an infinite number of decompositions. To solve this problem, we have to ignore the elements in sequences that produce an empty string. To do this, we first define *unseq*:

$$\begin{array}{l} \mathit{unseq} : \mathbb{X} \times \mathbb{P} \times \mathit{Seq}(\mathbb{E}) \to \mathcal{P}(\mathbb{E}) \\ \mathit{unseq}(x, p, \Gamma_1{::}\ldots{::}\Gamma_n) \triangleq \{\, \Gamma : \quad \Gamma = \mathit{unbind}(x, \Gamma_1) = \ldots = \mathit{unbind}(x, \Gamma_n) \\ \qquad\qquad\qquad\qquad\qquad \wedge\ \Gamma_1[x]{::}\ldots{::}\Gamma_n[x] \text{ subsequence of } \Gamma[p] \quad \} \end{array}$$

$$
\begin{aligned}
[\![v]\!](w) &= \begin{cases} \mathbb{E} & \text{if } v = w \\ \emptyset & \text{otherwise} \end{cases} \\
[\![\langle = p{:}t \rangle]\!](w) &= \{\Gamma \in \mathbb{E} : \Gamma[p] = \mathit{read}_t(u)\} \\
[\![i\ j]\!](w) &= \bigcup_{u \cdot v = w} [\![i]\!](u) \cap [\![j]\!](v) \\
[\![\langle \texttt{if } p\rangle\, i\, \langle \texttt{else}\rangle\, j\, \langle \texttt{end}\rangle]\!](w) &= \{\Gamma \in [\![i]\!](w) : \Gamma[p]\} \\
&\quad \cup \{\Gamma \in [\![j]\!](w) : \neg\, \Gamma[p]\} \\
[\![\langle \texttt{apply } f \texttt{ with } x = p\rangle]\!](w) &= \{\mathit{unbind}(x, \Gamma) : \ \Gamma \in [\![\Delta(f)]\!](w) \\
&\quad \wedge \Gamma[p] = \Gamma[x]\} \\
[\![\langle \texttt{for } x \texttt{ in } p\rangle\, i\, \langle \texttt{end}\rangle]\!](w) &= \\
\bigcup_{\substack{n \in \mathbb{N} \\ w_1 \cdot w_2 \cdots w_n = u}} & \{\ \Gamma : \exists (\Gamma_1, \ldots, \Gamma_n) \in [\![i]\!](w_1) \times \ldots \times [\![i]\!](w_n), \\
& \quad \Gamma[p] = \Gamma_1[x] :: \ldots :: \Gamma_n[x] \\
& \quad \wedge \mathit{unbind}(x, \Gamma_1) = \ldots = \mathit{unbind}(x, \Gamma_n) \quad \}
\end{aligned}
$$

Fig. 6. A denotational semantics for reversing templates.

The idea is that *unseq* takes a variable x, a path p, and a sequence of environments $\Gamma_1 :: \ldots :: \Gamma_n$ where x is bound, and it retrieves all the Γ where $\Gamma_1 :: \ldots :: \Gamma_n$ can correspond to a subsequence of the environments that are created in the forward evaluation of a `for` x `in` p instruction applied to an environment Γ.

We also define *emptyseq* as follows:

$$
\begin{aligned}
&\mathit{emptyseq} : \mathbb{I} \times \mathbb{X} \times \mathbb{P} \times \mathit{Seq}(\mathbb{E}) \to \mathcal{P}(\mathbb{E}) \\
&\mathit{emptyseq}(i, x, p, \Gamma_1 :: \ldots :: \Gamma_n) \triangleq \{\, \Gamma : \ \forall d \in \Gamma[p] : \ d \neq \Gamma_1[x] \wedge \ldots \wedge d \neq \Gamma_n[x] \\
&\qquad\qquad \Leftrightarrow \Gamma[x \mapsto d] \in [\![i]\!](\varepsilon) \qquad \}
\end{aligned}
$$

Intuitively, the function *emptyseq* constrains the databases d in $\Gamma[p]$, that produce an empty string, to be distinct from those that are in the subsequence $\Gamma_1 :: \ldots :: \Gamma_n$. Finally, we can modify the rule for `for` templates using these definitions:

$$
[\![\langle \texttt{for } x \texttt{ in } p\rangle\, i\, \langle \texttt{end}\rangle]\!](w) = \bigcup_{\substack{n \in \mathbb{N} \\ w_1 \cdot w_2 \cdots w_n = u \\ \forall w_i : w_i \neq \varepsilon}} \ \bigcup_{\substack{\Gamma_1 \in [\![i]\!](w_1) \\ \ldots \\ \Gamma_n \in [\![i]\!](w_n)}} \mathit{emptyseq}(i, x, p, \Gamma_1 :: \ldots :: \Gamma_n) \cap \mathit{unseq}(x, p, \Gamma_1 :: \ldots :: \Gamma_n)
$$

This denotational semantics for reverse template, with the above modified rule for `for` templates, is the basis for our algorithm for sound reversal of templates, provided in the next section.

5 Sound reversal of templates

In Section 4 we developed a denotational semantics for reversing templates that is compositional and requires only a finite number of parse trees (for a given text). This semantics is not computable because it manipulates infinite sets of databases and environments: for instance, every environment in $\mathbb{E}$ can render the text `"foo"` from the template `"foo"`, and we cannot enumerate $\mathbb{E}$.

A similar problem exists in the setting of sound static analysis of programs, that relies on a program semantics which is not computable due to the handling of infinite (or very large) sets of states $\mathbb{S}$. The solution to this problem is abstract interpretation [8] : the program semantics is approximated by a computation over an *abstract domain* $\mathbb{S}^\sharp$, which is a computer-representable lattice, such that $\mathbb{S}$ and $\mathbb{S}^\sharp$ are (typically) related by a Galois connection. Abstract interpretation is a *sound* method, i.e. it is guaranteed to compute a sound over-approximation of the set of all reachable states of the program; here we propose to use this method to soundly over-approximate the set of all the environments that can produce a given text with a given template.

We first present our *abstract databases* and *abstract environments*, respectively used to represent sets of concrete databases and environments; then our *abstract backward semantics*, which allows computing a sound overapproximation of the backward denotational semantics $[\![]\!]()$. We give the concretisation function γ that provides the formal meaning of the abstract domain by relating abstract elements to the set of concrete elements that it represents, and soundness theorems of all the main operations (we generally omit the proof, as the proof method is standard).

5.1 Abstract databases

Intuitively, we can see our abstract domains as representing databases as a collection of *constraints*, such that they describe the set of databases that obey these constraints. As there are three different kinds of databases (scalars, records, and sequences), we need three different kinds of constraints to represent them. The domains are presented formally in Figure 7. We define an *abstract database* $d^\sharp \in \mathbb{D}^\sharp$ to be either a *scalar value* $s \in \mathbb{D}_s$, an *abstract record* $d_r^\sharp \in \mathbb{D}_r^\sharp$ mapping fields to *abstract databases*, an *abstract sequence* $d_*^\sharp \in \mathbb{D}_*^\sharp$, or $\bot$, which represents an unsatisfiable constraint. Its concretisation is defined recursively as representing either a single scalar, the empty set, or a set of abstract databases or abstract sequences.

Remark 1. Our abstract elements cannot represent a set of databases that would contain both a scalar and a record. Thus, the abstraction function α cannot be defined on our domains, and we use a γ-only style of abstract interpretation [10]. Representing such a heterogeneous set is not needed, as abstract databases do not feature a join operator (instead, we perform a *disjunctive completion* [9] by *powerset completion* [2, 14] on abstract environments, see Section 5.2).

An *abstract record* $d_r^\sharp \in \mathbb{D}_r^\sharp$ is a partial function from fields to abstract databases. Intuitively, each field in the abstract record constrains the corresponding field in the concrete records (fields that are not in the abstract record are unconstrained).

Example 1. Let $d_r^\sharp = [\text{"}a\text{"} \mapsto [\text{"}b\text{"} \mapsto 1]]$. Then:

$$[\text{"}a\text{"} \mapsto 3] \notin \gamma_{\mathbb{D}_r^\sharp}(d_r^\sharp) \qquad [\text{"}d\text{"} \mapsto 8; \text{"}a\text{"} \mapsto [\text{"}b\text{"} \mapsto 1; \text{"}c\text{"} \mapsto 3]] \in \gamma_{\mathbb{D}_r^\sharp}(d_r^\sharp)$$

An *abstract sequence* is defined as a set of sequences of abstract databases. Its intuitive meaning is that the sequences in the set constrains the possible subsequences of the concrete elements.

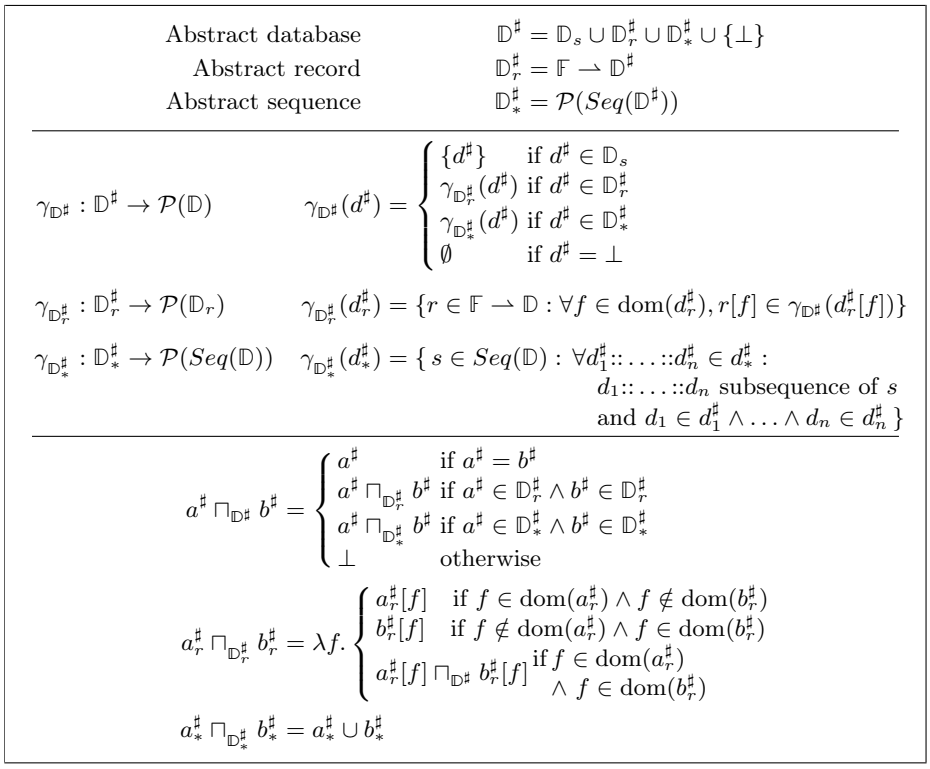

Fig. 7. Abstract databases: definitions, concretisations, and abstract intersections.

Intuitively, this set of constraints appears because we do not have another mean to merge constraints over different subsequences of the same sequence. This problem, which is similar to the alignment problem in bidirectional languages [4, 24] is studied more in depth in Section 7.

Example 2. Let $d_*^\sharp = \{1::2, 1::3\}$. Then:

$$\{1::2::4::3, 1::5::3::2, 2::1::2::3\} \subseteq \gamma_{\mathbb{D}_*^\sharp}(d_*^\sharp) \qquad \{2::1::3, 1::3\} \cap \gamma_{\mathbb{D}_*^\sharp}(d_*^\sharp) = \{\}$$

As shown by this example, some intuitively important relations on the sequences (like the fact that the length of the sequence is known), are not captured by our abstraction; this is explored in Section 6.3.

The only operator that we need over abstract databases is *abstract intersection*, defined in Fig 7, and whose meaning is given by the following theorem:

Theorem 2 (Soundness and exactness of abstract intersection). *Abstract intersection operators* $\sqcap_{\mathbb{D}^\sharp}$, $\sqcap_{\mathbb{D}^\sharp_r}$, *and* $\sqcap_{\mathbb{D}^\sharp_*}$ *are sound and exact:*

$$\begin{aligned}
\forall (a^\sharp, b^\sharp) \in \mathbb{D}^\sharp : &\qquad \gamma_{\mathbb{D}^\sharp}(a^\sharp \sqcap_{\mathbb{D}^\sharp} b^\sharp) = \gamma_{\mathbb{D}^\sharp}(a^\sharp) \cap \gamma_{\mathbb{D}^\sharp}(b^\sharp) \\
\forall (a^\sharp_r, b^\sharp_r) \in \mathbb{D}^\sharp_r : &\qquad \gamma_{\mathbb{D}^\sharp_r}(a^\sharp_r \sqcap_{\mathbb{D}^\sharp_r} b^\sharp_r) = \gamma_{\mathbb{D}^\sharp_r}(a^\sharp_r) \cap \gamma_{\mathbb{D}^\sharp_r}(b^\sharp_r) \\
\forall (a^\sharp_*, b^\sharp_*) \in \mathbb{D}^\sharp_* : &\qquad \gamma_{\mathbb{D}^\sharp_*}(a^\sharp_* \sqcap_{\mathbb{D}^\sharp_*} b^\sharp_*) = \gamma_{\mathbb{D}^\sharp_*}(a^\sharp_*) \cap \gamma_{\mathbb{D}^\sharp_*}(b^\sharp_*)
\end{aligned}$$

5.2 Abstract environments

Abstract environment $\quad \mathbb{E}^\sharp = \mathbb{X} \rightharpoonup \mathbb{D}^\sharp$

$$\gamma_{\mathbb{E}^\sharp} : \mathbb{E}^\sharp \to \mathcal{P}(\mathbb{E}) \quad \gamma_{\mathbb{E}^\sharp}(\Gamma^\sharp) = \{\Gamma \in \mathbb{X} \rightharpoonup \mathbb{D} : \forall x \in \mathrm{dom}(\Gamma^\sharp), \Gamma[x] \in \gamma_{\mathbb{D}^\sharp}(\Gamma^\sharp[x])\}$$

$$\mathit{unseq}^\sharp(x, p, \Gamma^\sharp_1 :: \ldots :: \Gamma^\sharp_n) = \left(\prod_{k \in 1..n}{}_{\mathbb{E}^\sharp} \mathit{unbind}(x, \Gamma^\sharp_k)\right) \sqcap_{\mathbb{E}^\sharp} [p \mapsto \{\Gamma^\sharp_1[x] :: \ldots :: \Gamma^\sharp_n[x]\}]$$

$$\Gamma^\sharp_1 \sqcap_{\mathbb{E}^\sharp} \Gamma^\sharp_2 = \lambda x. \begin{cases} \Gamma^\sharp_1[x] & \text{if } x \in \mathrm{dom}(\Gamma^\sharp_1) \wedge x \notin \mathrm{dom}(\Gamma^\sharp_2) \\ \Gamma^\sharp_2[x] & \text{if } x \notin \mathrm{dom}(\Gamma^\sharp_1) \wedge x \in \mathrm{dom}(\Gamma^\sharp_2) \\ \Gamma^\sharp_1[x] \sqcap_{\mathbb{D}^\sharp} \Gamma^\sharp_2[x] & \text{if } x \in \mathrm{dom}(\Gamma^\sharp_1) \\ & \wedge\, x \in \mathrm{dom}(\Gamma^\sharp_2) \end{cases}$$

Fig. 8. The *abstract environment* $\mathbb{E}^\sharp$ abstract domain.

Abstract environments $\quad \mathbb{E}^\sharp_\vee = \mathcal{P}(\mathbb{E}^\sharp)$

$$\gamma_{\mathbb{E}^\sharp_\vee} : \mathbb{E}^\sharp_\vee \to \mathcal{P}(\mathbb{E}) \qquad \gamma_{\mathbb{E}^\sharp_\vee}(e^\sharp) = \bigcup_{\Gamma^\sharp \in e^\sharp} \gamma_{\mathbb{E}^\sharp}(\Gamma^\sharp)$$

$$\top_{\mathbb{E}^\sharp_\vee} = \{[]\} \qquad \bot_{\mathbb{E}^\sharp_\vee} = \{\} \qquad e^\sharp_1 \sqcup_{\mathbb{E}^\sharp_\vee} e^\sharp_2 = e^\sharp_1 \cup e^\sharp_2$$

$$e^\sharp_1 \sqcap_{\mathbb{E}^\sharp_\vee} e^\sharp_2 = \{\, \Gamma^\sharp_1 \sqcap_{\mathbb{E}^\sharp} \Gamma^\sharp_2 : \Gamma^\sharp_1 \in \gamma_{\mathbb{E}^\sharp_\vee}(e^\sharp_1) \wedge \Gamma^\sharp_2 \in \gamma_{\mathbb{E}^\sharp_\vee}(e^\sharp_2) \,\}$$

Fig. 9. The domain of *abstract environments* $\mathbb{E}^\sharp_\vee$ is the powerset completion of $\mathbb{E}^\sharp$.

Similarly to abstract databases, we define *abstract environments* to represent a set of environments. An *abstract environment* $\Gamma^\sharp \in \mathbb{E}^\sharp$ is a mapping from variables to abstract databases. It is very similar to abstract records (except that it is indexed by variables instead of fields); in particular, the concretisation $\gamma_{\mathbb{E}^\sharp}$ and abstract intersection $\sqcap_{\mathbb{E}^\sharp}$ are very similar to the concretisation $\gamma_{\mathbb{D}^\sharp_r}$ and intersection $\sqcap_{\mathbb{D}^\sharp_r}$ of abstract databases.

Theorem 3. $\sqcap_{\mathbb{E}^\sharp}$ *is sound and exact:*

$$\forall(\Gamma_a^\sharp, \Gamma_b^\sharp) \in \mathbb{E}^\sharp : \quad \gamma_{\mathbb{E}^\sharp}(\Gamma_a^\sharp \sqcap_{\mathbb{E}^\sharp} \Gamma_b^\sharp) = \gamma_{\mathbb{E}^\sharp}(\Gamma_a^\sharp) \cap \gamma_{\mathbb{E}^\sharp}(\Gamma_b^\sharp)$$

We define another transfer function on abstract environments, $\mathit{unseq}^\sharp$, that is used to revert a ⟨`for` x `in` p⟩ ⟨`end`⟩ instruction: it receives a variable $x \in \mathbb{X}$, a path $p \in \mathbb{P}$, and a sequence of abstract environments in which x is bound; and use this to constrain the path p. Specifically:

Theorem 4 (Soundness and exactness of $\mathit{unseq}^\sharp$).
Let $x \in \mathbb{X}$, $p \in \mathbb{P}$, and $\Gamma_1^\sharp :: \ldots :: \Gamma_n^\sharp \in Seq(\mathbb{E}^\sharp)$. Then:

$$\gamma_{\mathbb{E}^\sharp}(\mathit{unseq}^\sharp(x, p, \Gamma_1^\sharp :: \ldots :: \Gamma_n^\sharp)) = \bigcup_{\Gamma_1 \in \Gamma_1^\sharp,\ \ldots,\ \Gamma_n \in \Gamma_n^\sharp} \mathit{unseq}(x, p, \Gamma_1 :: \ldots :: \Gamma_n)$$

We define the lattice of *abstract environments* $\mathbb{E}_\vee^\sharp$ to be the standard *powerset completion* [2, 9, 14] of $\mathbb{E}^\sharp$.

5.3 Abstract semantics

Equipped with these abstract domains, we can now propose an algorithm for reversing templates (Fig 10). This algorithm derives from the backward denotational semantics of Section 4.2, using the denotational style of abstract interpretations [44]. This allows, using the soundness theorems on the abstract domains, to prove the soundness of our main algorithm:

Theorem 5 (Soundness of abstract semantics). *Given any template i and word w:*

$$\gamma_{\mathbb{E}_\vee^\sharp}([\![i]\!]^\sharp(w)) \ \supseteq \ [\![i]\!](w)$$

Thanks to the compositional nature of both the abstract and backward denotational semantics, the proof is easily done by induction. It is worth noting that the part where we lose exactness (replacing equality by $\subseteq$) is the abstraction of `for` instructions, as we do not retain the information expressed by *emptyseq* in the backward semantics.

6 Precision and exactness

We now have a sound algorithm that retrieves a superset of the input environments that can produce a template for a given text.

As shown in Section 2, retrieving too many inputs can later lead to false alarms when trying to validate the inputs against requirements. Thus, we want to study whether this set can be sufficiently precise to be of practical use, and when would the algorithm have optimal precision.

$$[\![i]\!]^\sharp(\cdot) : \mathbb{W} \to \mathbb{E}^\sharp_\vee \qquad [\![v]\!]^\sharp(w) = \begin{cases} \top_{\mathbb{E}^\sharp_\vee} & \text{if } v = w \\ \bot_{\mathbb{E}^\sharp_\vee} & \text{otherwise} \end{cases}$$

$$[\![\langle = p{:}t \rangle]\!]^\sharp(w) = \{[p \mapsto \mathit{read}_t(u)]\} \qquad [\![i\ j]\!]^\sharp(w) = \bigsqcup_{u \cdot v = w}{}_{\mathbb{E}^\sharp_\vee} \ [\![i]\!]^\sharp(u) \sqcap_{\mathbb{E}^\sharp_\vee} [\![j]\!]^\sharp(w)$$

$$[\![\langle \texttt{if}\ p \rangle\ i\ \langle \texttt{else} \rangle\ j\ \langle \texttt{end} \rangle]\!]^\sharp(w) = [\![i]\!]^\sharp(w) \sqcap_{\mathbb{E}^\sharp_\vee} \{[p \mapsto \mathit{true}]\} \sqcup_{\mathbb{E}^\sharp_\vee} [\![j]\!]^\sharp(w) \sqcap_{\mathbb{E}^\sharp_\vee} \{[p \mapsto \mathit{false}]\}$$

$$[\![\langle \texttt{apply}\ f\ \texttt{with}\ x = p \rangle]\!]^\sharp(w) = \{\ \mathit{unbind}(x, \Gamma^\sharp[p \mapsto \Gamma^\sharp[p] \sqcap_{\mathbb{D}^\sharp} \Gamma^\sharp[x]]) : \Gamma^\sharp \in [\![\Delta(f)]\!]^\sharp(w)\ \}$$

$$[\![\langle \texttt{for}\ x\ \texttt{in}\ p \rangle\ i\ \langle \texttt{end} \rangle]\!]^\sharp(w) = \bigsqcup_{\substack{n \in \mathbb{N} \\ w_1 \cdot w_2 \cdots w_n = u \\ \forall w_i : w_i \neq \varepsilon}}{}_{\mathbb{E}^\sharp_\vee} \ \{\ \mathit{unseq}^\sharp(x, p, \Gamma^\sharp_1 :: \ldots :: \Gamma^\sharp_n) : \Gamma^\sharp_1 \in [\![i]\!]^\sharp(w_1) \wedge \ldots \wedge \Gamma^\sharp_n \in [\![i]\!]^\sharp(w_n)\ \}$$

Fig. 10. A sound algorithm for reversing templates.

6.1 Optimal precision

Intuitively, the result of an algorithm for reversing templates would be optimally precise if it returned a single environment, which would be the one used to generate the text. Unfortunately, this is impossible:

Theorem 6. *If some environment Γ exists that generates an output w from a template instruction i, then $[\![i]\!](w)$ is an infinite set.*

Proof. We can create arbitrarily many suitable inputs by adding arbitrary bindings to Γ.

Intuitively, all the bindings that are not used during the evaluation of the template cannot be recovered by the reverse processing, and can have arbitrary values. We thus adopt a more restricted definition of optimal precision by considering only the values bound to paths that are used in the forward evaluation. We first define an order relation $\preccurlyeq$ on databases such that $d_a \preccurlyeq d_b$ means that d_a has fewer bindings than d_b:

Definition 2. *We define a relation $\preccurlyeq \in \mathbb{D} \times \mathbb{D}$ on databases as the smallest relation satisfying:*

- *If $d \in \mathbb{D}_s$ is a scalar, then $d \preccurlyeq d$.*
- *If $d_a, d_b \in \mathbb{D}_r$ are records, $\mathrm{dom}(d_a) \subseteq \mathrm{dom}(d_b)$, and $\forall f \in \mathrm{dom}(d_a) : d_a(f) \preccurlyeq d_b(f)$, then $d_a \preccurlyeq d_b$.*
- *If $d_a, d_b \in \mathbb{D}_*$ are sequences and there exists a subsequence d'_b of d_b such that d_a and d'_b have the same length n, and $\forall k : 0 \leq k \leq (n-1) \Rightarrow d_a(k) \preccurlyeq d'_b(k)$, then $d_a \preccurlyeq d_b$.*

The definition is naturally extended to environments by considering that environments are similar to records (and are the same if field names and variable names are the same set).

Theorem 7. *$\preccurlyeq$ is a partial order.*

Example 3. $[\text{"}a\text{"} \mapsto [\text{"}b\text{"} \mapsto 1::3]] \preccurlyeq [\text{"}d\text{"} \mapsto 8; \text{"}a\text{"} \mapsto [\text{"}b\text{"} \mapsto 1::2::3::4; \text{"}c\text{"} \mapsto 3]]$

This relation allows to "minimize" a database to consider only the parts that are relevant for a given template:

Theorem 8. *Let $i \in \mathbb{I}$ be a template instruction, $w \in \mathbb{W}$ be a word, $\Gamma \in \mathbb{E}$ be an environment, such that $(\!|i|\!)(\Gamma) = w$. The set $\{\Gamma' \mid \Gamma' \preccurlyeq \Gamma \wedge (\!|i|\!)(\Gamma') = w\}$ has a minimum, i.e. a unique minimal element. If this minimum is Γ, we say that Γ is* minimal for i.

Proof sketch. One can collect all the paths that are used during the evaluation of Γ by i, which we call the *footprint* (see Appendix for a formal definition) All the elements corresponding to a path in the footprint must be preserved, otherwise the forward evaluation fails. On the contrary, all the elements that are not in the footprint can be removed without affecting the evaluation. The minimal database is the one that only contains the elements in the footprint.

Thanks to this theorem, we can now partition $[\![i]\!](w)$ into equivalent classes, where Γ and Γ' are equivalent if they can be minimized to the same Γ''; these minimal elements are the representative of the equivalence class. In other word, we can now consider only minimal elements. This allows to define precision as follows:

Definition 3. *Template reversal for a template instruction i and word w is precise if $[\![i]\!](w)$ has only one equivalence class; or equivalently, if only one environment Γ exists such that Γ is minimal for i and $(\!|i|\!)(\Gamma) = w$.*

This definition thus means that there is a unique preimage Γ of w through i, up to minimality. This notion is important for our motivating example: if we can recover this Γ, it means that we have retrieved all the relevant parts of the database that have produced the output.

Remark 2. Another immediate application of program inversion is bidirectional programming [29]: if we first evaluate $(\!|i|\!)(\Gamma) = w$, that w is manually modified, and $[\![i]\!](w')$ returns a single minimal Γ'. Then, using the principle of constant complementation [3, 17, 29], i.e. adding to Γ' the values of Γ that are not bound to any path in Γ', we can update the database to soundly reflect the edits of w. If the template reversal for i and w' is not precise, then we would have to disambiguate the solution, either asking the user for help, or using hints in the template language.

6.2 Inherent imprecisions

There are only two reasons why the backward evaluation of a template instruction i for an output w is imprecise (i.e. the computed abstract environments do not form a single equivalence class):

- Either $[\![i]\!](w)$ is imprecise, and the imprecision comes from the template or particular output: there are several minimal preimages of w for the template i.

– Or $[\![i]\!](w)$ is precise, but the imprecision comes from the fact that $[\![i]\!]^\sharp(w)$ is not *exact* [41], meaning that the abstraction introduces further imprecisions.

In this section we study the first source of imprecision, that we call *inherent* imprecisions. An interesting result if template reversal is imprecise for a template i and output w, this means that w is syntactically ambiguous for the grammar corresponding to the template i. More precisely:

Theorem 9. *If $[\![i]\!](w)$ is imprecise, there exists two environments Γ_1 and Γ_2 in $[\![i]\!](w)$ such that the forward evaluation of Γ_1 and Γ_2 (with the original* FOR *rule) produce deduction/parse trees with different shapes.*

Proof sketch. Suppose that we have Γ_1 and Γ_2 that belong to $[\![i]\!](w)$, are minimal and distinct, and that their evaluation corresponds to deduction trees with the same shape (i.e. the evaluation takes the same branch on conditional tests, and sequences have the same number of elements). The sequence of leaves in a deduction tree is equivalent to a sequence of either fixed strings or calls to $\mathit{show}_t(\Gamma[p])$ for some p, and both sequence must be equal to w. If Γ_1 and Γ_2 have the same parse tree, then this sequence is the same, which means that every $\Gamma[p]$ in the sequence must have the same printed representation. This means that Γ_1 and Γ_2 have equal values for every p in their footprint—which contradicts the fact that Γ_1 and Γ_2 are minimal and distinct.

This theorem is useful, because detecting parsing ambiguity at runtime is relatively easy – for instance the algorithm of $[\![i]\!]^\sharp(w)$, the number of parse trees is the cardinal of the element in $\mathbb{E}^\sharp_\vee$ which is returned (without counting $\bot_{\mathbb{E}^\sharp}$ elements).

Another application of this theorem is that we can enumerate the causes of imprecision in the semantics; it can help the template developer to change the template to fix any ambiguity (e.g. by adding fixed text, e.g. in comments when the output is source code). Imprecisions are due to:

– Ambiguities in concatenation, e.g.
$[\![\langle = x{:}int\,\rangle\ \langle = y{:}int\,\rangle]\!](123) = \{\,[x \mapsto 1, y \mapsto 23], [x \mapsto 12, y \mapsto 3]\}$
– Ambiguities in loops, e.g.
$[\![\langle \texttt{for}\ x\ \texttt{in}\ s\rangle\,\langle = x{:}int\,\rangle\,\langle \texttt{end}\rangle]\!](12) = \{[s \mapsto 1{::}2], [s \mapsto 12]\}$
– Ambiguities in conditionals, e.g.
$[\![\langle \texttt{if}\ c\rangle\,0\,\langle \texttt{else}\rangle\,\langle = x{:}int\,\rangle\,\langle \texttt{end}\rangle]\!](0) = \{\,[c \mapsto \mathit{true}], [c \mapsto \mathit{false}, x \mapsto 0]\}$
– Loops containing an empty production, e.g.
$[\![\langle \texttt{for}\ x\ \texttt{in}\ s\rangle\,\langle \texttt{if}\ x.c\rangle\,\langle = x.id{:}int\,\rangle\,\langle \texttt{else}\rangle\ \langle \texttt{end}\rangle\,\langle \texttt{end}\rangle]\!](7) =$
$\{\,[s \mapsto [c \mapsto \mathit{false}, id \mapsto 7]],$
$[s \mapsto [c \mapsto \mathit{true}]{::}[c \mapsto \mathit{false}, id \mapsto 7]],$
$[s \mapsto [c \mapsto \mathit{true}]{::}[c \mapsto \mathit{true}]{::}[c \mapsto \mathit{false}, id \mapsto 7]], \ldots\}$

In concatenation and loops, the ambiguity can often be fixed by inserting fixed string that acts as a separator between two replacements. Note that removal of ambiguity is the reason why, in the language, our replacements take a type argument. Indeed, if, like in many template languages, the replacement could be arbitrary, then the parsing of loop sequences would become highly ambiguous—as

it would be possible for separators to be part of the replacement text. For instance, the ambiguity of the following backward evaluation is probably not expected:

$$[\![\langle \texttt{for}\ x\ \texttt{in}\ s\rangle\ \langle = x{:}any\,\rangle;\ \langle \texttt{end}\rangle]\!](\text{"}\texttt{11;22;}\text{"}) = \{\,[s \mapsto \text{"}\texttt{11}\text{"}::\text{"}\texttt{22}\text{"}], [s \mapsto \text{"}\texttt{11;22}\text{"}]\}$$

For conditionals, the template should be written such that the languages corresponding to each branch of a conditional do not overlap. One easy way to do it, if the template produces text for a language that allows comments, is to add different comments in the different branches of the condition:

$$[\![\langle \texttt{if}\ c\rangle\ /{*}c{*}/0\ \langle \texttt{else}\rangle\ /{*}!c{*}/\langle = x{:}int\,\rangle\ \langle \texttt{end}\rangle]\!](\text{"}/{*}!c{*}/0\text{"}) = \{[c \mapsto \mathit{false}; x \mapsto 0]\}$$

6.3 Imprecision coming from the abstraction

We now study the second source of imprecision, i.e. imprecisions coming from the abstraction. As we have seen, many of our transfer functions, like abstract intersection, are *exact*. A transfer function $f^\sharp$ is an *exact* [41] (also called forward-complete [19]) approximation of a function f if $f \circ \gamma = \gamma \circ f^\sharp$ (this extends to binary functions).

If S is a set, and $S^\sharp$ is such that $\gamma(S^\sharp) = S$, then $\gamma(f^\sharp(S^\sharp)) = f(S)$. This means that if you only use exact transfer functions in the algorithm for reversing templates (Figure 10), then the result of the algorithm exactly represents the set of all the inputs of the templates: the algorithm introduces no imprecision.

An important source of inexactness is the join operator. For instance, a join operator on our abstract environment abstract domain $\mathbb{E}^\sharp$, would introduce a catastrophic loss of precision, as when joining two environments $\Gamma_1^\sharp$ and $\Gamma_2^\sharp$, any information known about a variable x bound in only one of the Γ would be lost in the result. This is the reason why we introduced a top-level powerset abstract domain $\mathbb{E}^\sharp_\vee$, which allows performing unions without introducing imprecisions.

There is still one operation in our algorithm which is inexact, which is the handling of sequences. In Section 4.2, we introduced the *emptyseq* function to characterize elements in the sequence that produce an empty word. But this information is completely lost in the abstract semantics of Figure 10.

A simple way to fully recover part of this information, which is very useful in practice, is to use a two-element lattice $\{\mathit{complete}, \mathit{incomplete}\}$ with $\mathit{complete} \sqsubseteq \mathit{incomplete}$, where *complete* means that the abstract element corresponds to the whole sequence, and not to only a subsequence, of the concrete sequence.

We thus change our definition of abstract sequences such that

$$\mathbb{D}^\sharp_* \triangleq \{\mathit{complete}, \mathit{incomplete}\} \times \mathcal{P}(Seq(\mathbb{D}^\sharp))$$

and change the definition of $\gamma_{\mathbb{D}^\sharp_*}$ in Figure 7 such that "subsequence of" is replaced by "is equal to" when the abstract sequence is *complete*. Finally, in the algorithm computing $[\![\langle \texttt{for}\ x\ \texttt{in}\ p\rangle\ i\ \langle \texttt{end}\rangle]\!]^\sharp(w)$ (Figure 10), we should set the sequence type to *complete* if the interpretation of the template i cannot produce the empty word, i.e. if $[\![i]\!]^\sharp(\varepsilon) = \bot$.

With these additions, our abstract algorithm is now exact on the example of Figure 2, where every `for` instruction cannot produce empty words for any element. We can actually prove this theorem:

$$\rho_{\mathbb{D}^\sharp} : \mathbb{D}^\sharp \to \mathbb{D}^\sharp \qquad \rho_{\mathbb{D}^\sharp}(d^\sharp) = \begin{cases} d^\sharp & \text{if } d^\sharp \in \mathbb{D}_s \\ \rho_{\mathbb{D}_r^\sharp}(d^\sharp) & \text{if } d^\sharp \in \mathbb{D}_r^\sharp \\ \rho_{\mathbb{D}_*^\sharp}(d^\sharp) & \text{if } d^\sharp \in \mathbb{D}_*^\sharp \end{cases}$$

$$\rho_{\mathbb{D}_r^\sharp} : \mathbb{D}_r^\sharp \to \mathbb{D}_r^\sharp \qquad \rho_{\mathbb{D}_r^\sharp}(d_r^\sharp) = \begin{cases} \bot & \text{if } \exists f \in \mathrm{dom}(d_r^\sharp) : \rho[d_r^\sharp[f]] = \bot \\ \lambda f.\rho_{\mathbb{D}_r^\sharp}[d_r^\sharp[f]] & \text{otherwise} \end{cases}$$

$$\rho_{\mathbb{E}^\sharp} : \mathbb{E}^\sharp \to \mathbb{E}^\sharp \qquad \rho_{\mathbb{E}^\sharp}(\Gamma^\sharp) = \begin{cases} \bot & \text{if } \exists x \in \mathrm{dom}(\Gamma^\sharp) : \rho_{\mathbb{D}_r^\sharp}[\Gamma^\sharp[x]] = \bot \\ \lambda x.\rho[\Gamma^\sharp[x]] & \text{otherwise} \end{cases}$$

$$\rho_{\mathbb{E}_\vee^\sharp} : \mathbb{E}_\vee^\sharp \to \mathbb{E}_\vee^\sharp \qquad \rho_{\mathbb{E}_\vee^\sharp}(e^\sharp) = \{\rho_{\mathbb{E}^\sharp}(\Gamma^\sharp) : \Gamma^\sharp \in e^\sharp \wedge \rho_{\mathbb{E}^\sharp}(\Gamma^\sharp) \neq \bot\}$$

Fig. 11. Some reduction operators for our abstract domains.

Theorem 10. *Let $e^\sharp = [\![i]\!]^\sharp(w)$. If $e^\sharp$ does not contain any incomplete abstract sequences, then the algorithm is exact: $\gamma_{\mathbb{E}_\vee^\sharp}(e^\sharp) = [\![i]\!](w)$*

Combining this theorem with our analysis on inherent imprecisions yields:

Corollary 1. *Let $e^\sharp = [\![i]\!]^\sharp(w)$. If $e^\sharp$ is a singleton and does not contain an incomplete abstract sequence, then $[\![i]\!]^\sharp(w)$ is precise.*

These theorems allow testing if the result of algorithm of Figure 10 is precise, based only the structure of abstract elements.

Remark 3. If the representation of *complete* abstract sequences is maximally precise, this is not the case for *incomplete* ones. More information could be retained in this case on the elements in the sequence that produce the empty word. This also requires updating operators like the intersection operator, as forgetting about this new information would introduce imprecision.

7 Reduction for constraint propagation

Abstract interpretation provides a mean to propagate constraints using an operator called reduction.

Definition 4. *Given an abstract domain $\mathbb{D}^\sharp$, a partial reduction operator $\rho \in \mathbb{D}^\sharp \to \mathbb{D}^\sharp$ is a function such that:*

1. $\forall d^\sharp \in \mathbb{D}^\sharp : \gamma(\rho(d^\sharp)) = \gamma(d^\sharp)$ *(reduction does not change the concretisation)*
2. $\forall d^\sharp \in \mathbb{D}^\sharp : \rho(d^\sharp) \sqsubseteq^\sharp d^\sharp$ *(reduction produces a smaller, simpler abstraction)*

Note that we did not need to define an ordering $\sqsubseteq^\sharp$ on our abstract values up to now so the second condition is less important; a suitable ordering can be defined as $d^\sharp \sqsubseteq^\sharp d'^\sharp \Leftrightarrow \rho(d^\sharp) = \rho(d^\sharp \sqcap d'^\sharp)$.

Removal of $\bot$ There are several places where this operator is useful in our analysis. Figure 11 presents operators for all domains except abstract sequences, that we detail shortly after. The main goal here is to propagate the reductions across all the domains, to remove $\bot$ elements. Indeed, the abstract intersection ($\sqcap$) operators creates $\bot$ elements when the constraints are irreconcilable; this is used in particular, to disambiguate some cases of ambiguous parsing.

Removing these $\bot$ elements is important mainly for performance reasons. Indeed, the $\sqcap_{\mathbb{E}^\sharp_\vee}$ operator has quadratic complexity due to the powerset completion, so removing useless elements early allows to perform fewer computations. Thus, reductions should be performed just after intersections (in practice, it is sometimes easier to interleave the reduction step directly in the definition of the abstract intersection operations).

Reduction of complete sequences Consider the following example, where a sequence appears twice in the template:

$$\left[\!\!\left[\begin{array}{l} A{:}\langle \texttt{for } c \texttt{ in } seq\rangle\ \langle = c.a\texttt{.} : int\,\rangle;\ \langle \texttt{end}\rangle \\ B{:}\langle \texttt{for } c \texttt{ in } seq\rangle\ \langle = c.b\texttt{.} : int\,\rangle;\ \langle \texttt{end}\rangle \end{array} \right]\!\!\right]^\sharp (\texttt{A:1;2;B:3;4;}) =$$
$$[seq \mapsto \{(\mathit{complete}, [a \mapsto 1]::[a \mapsto 2]), (\mathit{complete}, [b \mapsto 3]::[b \mapsto 4])\}]$$

Following our addition from Section 6.3, we can now find out that we retrieved the complete sequence. But the result is not quite the one we would like: what the domain says here is that it saw two complete sequences, but it is unable to merge the elements of the sequence together. The problem here is not a problem of precision, as the concretisation of this domain is optimally precise; the problem comes from the lack of reduction, that the following operator solves. By simplicity this operator is defined $\rho_{\mathbb{D}^\sharp_*}$ on pairs of two elements; the actual operator recursively applies it to every pair until the resulting set can no longer be reduced.

$$\rho_{\mathbb{D}^\sharp_*} : \mathbb{D}^\sharp_* \to \mathbb{D}^\sharp_*$$
$$\rho_{\mathbb{D}^\sharp_*}(\{(\mathit{complete}, d_1::\ldots::d_n), (\_, d'_1::\ldots::d'_n)\}) = \{(\mathit{complete}, d_1 \sqcap_{\mathbb{D}^\sharp} d'_1::\ldots::d_n \sqcap_{\mathbb{D}^\sharp} d'_n)\}$$
$$\rho_{\mathbb{D}^\sharp_*}(\{(\_, d_1::\ldots::d_n), (\mathit{complete}, d'_1::\ldots::d'_n)\}) = \{(\mathit{complete}, d_1 \sqcap_{\mathbb{D}^\sharp} d'_1::\ldots::d_n \sqcap_{\mathbb{D}^\sharp} d'_n)\}$$
$$\rho_{\mathbb{D}^\sharp_*}(\{d_*, d'*\}) = \{d_*, d'_*\} \qquad \text{otherwise}$$

This operator merges together complete sequences when feasible, merging elements based on their positions. Note that it can also merge a complete sequence with a maybe-incomplete one if they have the same number of elements (meaning that the maybe-incomplete sequence is actually complete). Applying this reduction operator on the example above yields the expected result:

$$\rho_{\mathbb{D}^\sharp_*}([seq \mapsto \{(\mathit{complete}, [a \mapsto 1]::[a \mapsto 2]), (\mathit{complete}, [b \mapsto 3]::[b \mapsto 4])\}]) =$$
$$\left[seq \mapsto \left\{\left(\mathit{complete}, \begin{bmatrix} a \mapsto 1 \\ b \mapsto 3 \end{bmatrix} :: \begin{bmatrix} a \mapsto 2 \\ b \mapsto 4 \end{bmatrix}\right)\right\}\right]$$

Reduction of partial sequences Without any additional information, we cannot learn anything about two partial sequences, because we don't know how to

combine different elements together. However, provided that we could uniquely identify elements in the sequence (e.g. using a unique field, or position in the sequence), we could also reduce partial sequences by merging them into a DAG representing the partial order in which elements have been seen.

Such an extension has similarities with the key-based list alignment strategy in bidirectional programming languages [4, 24] that allows to match elements between the source and the view when the list was modified using some key. In general, the strategies used in bidirectional programming to reconcile elements in sequences between the source and the updated view, can also be useful to merge information about sequences in our template inversion problem.

8 Implementations and evaluation

We have implemented this technique on a custom template language called KIT. KIT is a project developed by a small business that develops tools and kernels for safety-critical real-time systems, used in automotive, aerospace & defense, and industrial automation industries. They have developed a custom source-to-source compiler that produces C files based on high-level descriptions of the timing behaviour of their systems. The compiler proceeds in two passes; first it generates an XML database that contains all the low-level parameters, such as the size of the communication buffers of the applications, the authorized inter-process communication between the tasks, or the configuration options used to compile the kernel and the applications. The KIT template engine then process template files using this database to generate C source code containing the application parameters, that will be compiled and linked with the application and kernel code to create an executable describing the whole system.

We proposed template inversion as a possible solution to their problem of independent validation of their generated source code (note that the templates are shared between the generator and validator, but this is not a problem because the templates can be viewed as specifications of the format of the source code). To test our solution, we were given KIT templates and the corresponding output for a sample application. We redeveloped a parser for the KIT language, and a KIT reverse evaluator, in OCaml.

8.1 Extensions for a full-featured template language

The KIT template language gradually evolved and includes numerous features besides those described in our core language. These features are important for practical usage of the template language. Most of these extensions could be handled by transforming the full KIT language into a "core kit" one:

- `include` directives were handled by inlining (transitively) the included file into the main file;
- Comments are widely used in the KIT files; we just remove them during the lexical analysis of KIT templates;

– There are lexical variations around the instructions which allows adjusting newlines between the source code and the KIT code, also handled during the lexical analysis of KIT templates;
– There are assertions in the code, that are handled using constraints as described below;
– There is a `select` instruction which allows querying for a single element in a sequence. This was handled by updating the definition of path;
– The scalars were untyped, but we created a "catch-all" type consisting in int + symbol + strings + floats, that works in practice for this use-case.
– The template functions are defined in the kit language, and can have more than one argument.

8.2 Arbitrary expressions

An interesting feature (commonly found in other template languages) is that replacement instructions, and conditionals, can contain *expressions* instead of simple paths. This is beneficial for code generation, as code generators can handle more situations without having to change the database. Consider reversing this HTML-producing template:

<**tr**>⟨`for` *c* `in` *seq*⟩
  <**td style**="color:⟨`if` *c.temp* < 0⟩ *blue* ⟨`else`⟩ *red* ⟨`end`⟩;">⟨= *c.temp*:*float* ⟩</**td**>
⟨`end`⟩</**tr**>

applied to the text:

<**tr**><**td style**="color:blue";>−40</**td**><**td style**="color:red";>451</**td**></**tr**>

The need for such an expression could be replaced by a change in the input database and a *c.is_temp_negative* test instead, but expressions remove this need. To handle expressions, we simply add our abstract environments $\mathbb{E}^\sharp$ a set of constraints that are bindings of the form *expression* $\mapsto$ *value*. Applied to the above example, this results in:

$$\left\{ \begin{array}{l} seq[0].temp < 0 \mapsto true, \\ seq[1].temp < 0 \mapsto false \end{array} \right\}, [seq \mapsto (\textit{complete}, [temp \mapsto -40] :: [temp \mapsto 451])]$$

Updating the transfer functions to modify the constraints when bindings change in `apply` and `for` instructions is fairly easy. Finally, we can change the reduction operator to take constraints into account. One simple way to do that is to substitute paths for their variable if their value is available; for instance, on the above example, the values of $seq[0].temp$ and $seq[1].temp$ are known. This allows to check if the constraint holds; if so, the constraint can be removed, otherwise we can reduce the environment to $\bot$. Thus, after reduction, our example becomes just $[seq \mapsto (\textit{complete}, [temp \mapsto -40] :: [temp \mapsto 451])]$. More complex strategies could be used, e.g. we could try to see if there is a unique solution to the set of constraints using an SMT solver. This strategy suffices for our use case, as we can always add text in the template (as comments in the output text) to add the values of all paths, so that all the constraints can be eliminated by the reduction operator.

8.3 Parsing

We have implemented two versions of our template reversing algorithm. In the first one, we relied on DYPGEN [37], a GLR parser generator for OCaml. The idea of this implementation was that a context-free grammar can be used as an overapproximation of the language that can be produced by a template instruction; semantic actions in the parser can then be used to drop the parse trees that cannot be produced by the template (which corresponds to the case where the backward evaluation of the template returns {}).

DYPGEN also implements a lexing pass, but it follows the longest match rule, so using it can lead to missing parse trees. Thus, we used single characters as tokens. The generated parser works well on simple examples, but unfortunately, generating the parser for some simple constant texts take several minutes, for instance in header texts like this one:

```
/********** thread_runtime.c.kit: runtime generation **********/
```

We measured that building the parsing tables for this bottom-up parser requires a time complexity which is exponential in the number of same characters put side by side. We then opted for direct top-down backtracking [5] implementation of the parsing algorithm, with two optimizations:

- When evaluating $[\![i\ j]\!]^\sharp(w)$, we do not attempt every decomposition of w. Instead, we try to parse i on w, and we ask it to return the set of all the decompositions of w for which it may return an environment, together with the abstract environment. This is much more efficient in particular when analyzing fixed strings, which is the most common template instructions.
- Abstract backward evaluation of a template i is thus a function of type $\mathbb{N} \to \mathcal{P}(\mathbb{N} \times \mathbb{E}^\sharp_\vee)$, where the integer represents a position in the string. To avoid reanalyzing the same position several times, we use memoization tables, like top-down chart parser [23] or packrat parsers [15].

8.4 Handling cyclic recursive calls

The most important limitation of our implementation is that we do not handle templates that correspond to left-recursive grammars (i.e. that recursively call a template in left position). We implemented this limitation because it is easy to detect if the non-terminal corresponding to a template function is called twice on the same position, which allows returning an error (instead of entering an infinite loop); because cyclic recursive calls did not appear in our industrial examples or case study (it seems difficult to obtain a singleton abstract environment when there are cyclic recursive calls); and because in many cases, it is easy to make the recursive calls non-cyclic (e.g. by systematically inserting parentheses when needed). Still, it is interesting to see if we could lift these limitations.

First, using grammar analysis, it would be theoretically easy to detect left-recursive calls that are not cyclic (i.e. for which the recursive cycle is followed by a terminal or a non-terminal that cannot produce the empty string), and allow the recursion to continue in this case.

KIT template	LoC	Output file	Lines	Chars	Parse time (s)	Equivalent classes
app.psy.runtime.tak.kit	251	app.psy.runtime.tak	345	10158	0.00	1
hal_error_runtime.c.kit	20	hal_error_runtime.c.kit	46	894	0.00	1
hal_runtime.c.kit	39	hal_runtime.c	34	1292	0.00	1
hal_sources.c.kit	63	hal_sources.c	31	838	0.00	1
runtime.c.kit	31	runtime.c	30	737	0.00	1
thread_runtime.c.kit	14	thread_runtime.c	10	164	0.00	1

Table 1. Evaluation on real-world templates.

Proper handling of truly cyclic recursive calls would require more substantial change, but these changes can be dealt with existing abstract interpretation tools. We can draw ideas from shape analysis to precisely represent the traversal of environments in recursive cycles, such as regular expression on access paths [12] or the inductive predicates of separation logic [6]. We can introduce widening [8] points when analyzing cycling recursive templates to compute an over-approximation of the environments for every possible depth of recursive calls to templates. Thus, abstract interpretation again provides the required tools to implement this extension.

8.5 Evaluation

Research questions Our main goal is to evaluate whether reversing template is feasible on real templates. We evaluated the following research questions:

- **RQ0: Soundness check**: does our algorithm return the expected result?
- **RQ1: Efficiency**: how fast is the algorithm?
- **RQ2: Precision**: is the algorithm precise on usual templates and outputs?

Protocol We focused on 6 representative `.kit` template files given by the industrial company developping its in-house KIT language, together with sample outputs, to evaluate the feasibility of the technique.

The results are given in Table 1, which provides the name of the KIT file, the number of lines (including comments), the name of the output file, its number of lines and character, the time required to parse the file as reported by `time`, and the number of equivalent classes (i.e., number of minimal databases) retrieved by our tool. In all the cases, running the parser on the sample application took less than 0.01s. The computed abstract environment was precise, and a single equivalence class was returned.

To check soundness, we created 7 KIT files corresponding to different ambiguous grammars. In each, our tool could retrieve all the expected abstract environments.

Finally, to evaluate scalability, we created a template with a single `for` template printing numbers separated by ';', and we evaluated it on sequences of different length. The results are reported in Table 2. This table shows that if the execution parsing time is supra-linear in some cases, it is able to handle outputs of fairly large size, sufficient for the generation of code in embedded systems. We could not try larger size has they created a stack overflow within our tool.

Sequence length	10000	20000	30000	40000
Parsing time (s)	18.8	71.65	148.95	344.22

Table 2. Measuring parsing time on large views.

Conclusions The technique had no problem handling existing templates. We feared that reversing the templates on some outputs might be ambiguous and that we would have to perform disambiguation using comments, but it actually never happened. We thus conclude that the technique is viable for reversing templates used to produce embedded systems code.

9 Related work

Inversion of injective programs Many of the work on program inversion is in the setting of injective functions, that only have a single preimage. In this setting, an early technique for inversion of string-to-string programs is the grammar-based approach of Yellin and Mueckstein [48]. Matsuda et al. [31] extends their idea to first-order functional programs transforming abstract datatypes and, like we do, makes use of the correspondance between evaluation trees and parse trees, and use constraints to recover the input. Finally, Matsuda and Wang [32] extends the technique to programs transforming abstract data types into strings, i.e. transforms a pretty-printer in a parser, which is quite similar to our problem.

We build on this work by using set-based compositional reasoning to handle non-injective functions, which may have multiple pre-images, and by proposing abstract interpretation as a framework to finitely represent these sets. This remove important limitations, such as the need for the function that we want to invert to be injective or to obey other sufficient conditions, such as being affine (variables cannot be duplicated, unlike in Figure 2) or nonerasing (the view must tell about every variable in the source).

Other techniques for inversing programs that also focus on injective programs include the use of invertible combinators [35], transformation of term rewriting systems [36], or (non-terminating) universal algorithms [1].

A generic technique for writing invertible programs, which do not need to be injective, is logic programming [25]. But logic programming, or, more generally, generation of constraints to be solved by an external solver, cannot finitely represent the set of all solutions, and can only enumerate solutions.

Finally, a related topic is that of bidirectional transformations [11, 21], that aim at running programs in reverse, but do so to solve the view-update problem [3]. This problem is formalized using the concept of lenses [16]. A lens is a pair containing a function *get* from a *source* to a *view*, and a function *put* that takes the original source, an updated view, and modifies the source based on the modifications on the view. This problem is distinct from our problem of finding all the preimages of a given function, but is related, and there are interesting convergence point between these problems. In particular, using the *constant complement* approach [3], one can derive a *put* function from inversion of the

get one, that preserves the part of input not used by the program, an approach called *syntactic bidirectionalization* [17, 29].

Approaches based on tree transducers An important line of work (e.g. [13, 18, 27, 30, 34, 39, 47]) is concerned with the problem of static type checking of XML transformations. In those, the template language is represented by a model which is generally a tree transducer, which takes and produces a tree. Then, given a transducer and a type R_{out} describing the set of outputs, the inverse type checking problem consists in finding the exact set R_{in} of inputs that may produce an output in R_{out}. Such an approach could be used for template inversion of a word w, using the singleton $\{w\}$ as R_{out}.

An important issue here is that the output in our problem is a string, and not a tree; thus the model of program that we would have to use is that of a tree-to-string transducer. Because strings are less structured than trees (which leads to parsing ambiguities when performing the inversion), this problem is more difficult. As said in Seidl et al. [45], "Amazingly little has been known so far for tree-to-string transducers": although their paper provides an algorithm for deciding the equivalence between tree-to-string transducers (using abstract interpretation), to our knowledge no algorithm exists for inverting the execution of such transducers.

Another issue is that if tree transducers represent a model of a template language, this model can differ from practical languages. Tree transducer replace conditions by non-deterministic choice, cannot compute expressions on the input data, and do not incorporate a notion of sequence. Such extensions could be handled by inventing extensions of existing transducers; for instance macro forest transducers [39] can deal with sequences and can be type-checked, albeit with an algorithm of high complexity. We chose abstract interpretation because it provides a framework to efficiently extend and combine different abstractions [9], some of which could be transducers or regular tree languages; and that it allows to explicitly choose the trade-off between precision and efficiency of inverse computation.

Abstract interpretation Abstract interpretation [8] is a sound method to derive sound static analysis of programs by the systematic construction of abstract domains [9]. In the canonical abstract interpretation framework, the abstract domain is a lattice which is in Galois-connection with a set, which is often the set of reachable states (or the set of traces) in the program. However, a backward analysis [7] computes an over-approximation of a set of states that can reach some condition, which is related to computing the set of preimages to a function. In program analyzes, backward analyses are usually too imprecise if they are not combined with a forward analysis Rival [43]. In our work, the fact that templates can be approximated using a contex-free grammar that generate a finite number of parse trees for a given output word, where parse trees can be viewed as a kind of evaluation trace, allows a very precise backward analysis (with no need for widening or approximation of joins), but abstract interpretation is still needed to handle the remaining imprecision.

The powerset domain that we use was introduced by Cousot and Cousot [9], and studied by Filé and Ranzato [14]. Bagnara et al. [2] noted that computing a widening operation on this domain is difficult; fortunately we do not need to compute a least fixpoint and thus do not need to introduce a widening operator (even with the introduction of functions in the template, the parse tree for a given output text is always finite).

Ranzato [41] defines two notions of completeness in abstract interpretation, which are completeness and exactness (also called forward-completeness). We use the exactness notion in our paper, to characterize the cases when the abstraction exactly represents the set of possible input environments. Exactness is less commonly used than completeness in abstract interpretation because having exact transfer functions yields an exact representation of the greatest fixpoint, and not of the usual least fixpoint. In our case, exactness can be used because the semantics that we analyze do not need to compute a fixpoint. Ranzato and Zanella [42] also uses exactness to verify properties over Support Vector Machines, which also are loop-free programs.

10 Conclusion

We studied the problem of reversing programs written in template languages, that transform tree-structured data into text. The template language that we use is expressive, and programs in this language are generally not injective (they have multiple preimages), not affine (some input variables can appear at several locations in the output), and erasing (they provide only a partial view of the source); they can use arbitrary expressions, which makes the problem undecidable and challenging. We propose a technique based on deriving a "backward" set-based denotational semantics from the forward big-step semantics, with suitable modifications to avoid the need to perform any fixpoint computation; and to use abstract interpretation to over-approximate this set. Using concepts from abstract interpretation, we can then study useful topics like precision or simplification of the abstract representation of the set of preimages. We believe that these concepts are generally useful when reasoning about reversing non-injective programs, and could find good use in other areas of program inversion, such as bidirectional transformations.

A Appendix

A.1 Template viewed as a context-free grammar

This appendix explains how templates in the RTL template language may be approximated using a context-free grammar. Figure 12 describes this translation (where $|$ denotes alternative choices, $\cdot$ concatenation, $*$ repetition). G_t is a grammar rule that corresponds to the printed representations of objects of type t.

This translation is not exact but approximate, in that the language

Theorem 11. *Given a template instruction i and a word w, if there exists Γ such that $(\!|i|\!)(\Gamma)$, then w belongs to the language generated by the context-free grammar $\mathcal{T}(i)$.*

$$
\begin{aligned}
\mathcal{T}(w) &\triangleq w \\
\mathcal{T}(\langle = p{:}t \rangle) &\triangleq G_t \\
\mathcal{T}(i_1\ i_2) &\triangleq \mathcal{T}(i_1) \cdot \mathcal{T}(i_2) \\
\mathcal{T}(\langle \texttt{if}\ p \rangle\ i_1\ \langle \texttt{else} \rangle\ i_2\ \langle \texttt{end} \rangle) &\triangleq \mathcal{T}(i_1) \mid \mathcal{T}(i_2) \\
\mathcal{T}(\langle \texttt{for}\ x\ \texttt{in}\ p \rangle\ i\ \langle \texttt{end} \rangle) &\triangleq (\mathcal{T}(i))* \\
\mathcal{T}(\langle \texttt{apply}\ \phi\ \texttt{with}\ x = p \rangle) &\triangleq \phi \\
\text{productions} &\triangleq \{\phi \to \mathcal{T}(\Delta(\phi)) \mid \phi \in \text{dom}(\Delta)\}
\end{aligned}
$$

Fig. 12. Translation of the RTL template language to a context-free grammar.

Proof. By induction on the instructions of the language.

Given a context-free grammar, we note by $\alpha \Rightarrow \beta$, where α and β are both words over terminals and non-terminals, the fact that β derives from α, meaning that β is obtained from α by substituting a non-terminal by its production [20]. We note by $\Rightarrow^+$ the transitive closure of $\Rightarrow$. We say that a grammar is cyclic if there is a nonterminal ϕ such that $\phi \Rightarrow^+ \phi$.

Theorem 12. *Given a context-free grammar, there exists a word w for which there is an infinite number of parse trees only if the grammar is cyclic.*

Proof. We suppose that the grammar is not cyclic. We consider a non-terminal ϕ in the grammar. If there is a string such that $\phi \Rightarrow^+ \alpha\phi\beta$, then either α or β must derive into a non-empty string (they cannot both derive in the empty string, otherwise the grammar would be cyclic).

This means that given a word w, in any derivation from ϕ to w, ϕ can be replaced only a finite number of time.

As this is true for every non-terminal ϕ and there is a finite number of non-terminals, the derivation from any non-terminal symbol to w must be finite. Because of the correspondence between derivations and parse trees [20], this means that there is a finite number of parse trees.

Thus, if w is a word produced by a template whose translation is a grammar that is not cyclic, then only a finite number of parse trees can parse w. The FOR-ALT rule is a relaxation of this constraint, allowing a finite number of parse tree even in some cyclic grammars (by considering a variation of context-free grammars where the * of repetition is a native construct).

A.2 Definition of footprint

Formally, we first extend the set of paths so that they can represent a specific element in a sequence, by adding the $p[k]$ construct:

$$\mathbb{P} \ni p, q \triangleq x \mid p.f \mid p[k]$$

Then, we define the footprint $\mathcal{F}$ of a template instruction i and environment Γ as the set of paths that is used during the evaluation of $(\!|i|\!)(\Gamma)$. The footprint is computable, and is defined recursively in Figure 13. This function makes use of a substitution function *subst*.

$$
\begin{aligned}
\mathit{subst} &: \mathbb{P} \times \mathbb{X} \times \mathbb{P} \to \mathbb{P} \\
\mathit{subst}(x, x, p) &= p \\
\mathit{subst}(y, x, p) &= p \text{ when } y \neq x \\
\mathit{subst}(q.f, x, p) &= \mathit{subst}(q, x, p).f \\
\mathit{subst}(q[k], x, p) &= \mathit{subst}(q, x, p)[k] \\
\mathcal{F} &: \mathbb{I} \times \mathbb{E} \rightharpoonup \mathcal{P}(\mathbb{P}) \\
\mathcal{F}(w, \Gamma) &= \{\} \\
\mathcal{F}(\langle = p{:}t \rangle, \Gamma) &= \{p\} \\
\mathcal{F}(i\ j, \Gamma) &= \mathcal{F}(i, \Gamma) \cup \mathcal{F}(j, \Gamma) \\
\mathcal{F}(\langle \texttt{if } p \rangle\, i\, \langle \texttt{else} \rangle\, j\, \langle \texttt{end} \rangle, \Gamma) &= \begin{cases} \mathcal{F}(i, \Gamma) \cup \{p\} & \text{if } \Gamma[p] = \mathit{true} \\ \mathcal{F}(j, \Gamma) \cup \{p\} & \text{if } \Gamma[p] = \mathit{false} \end{cases} \\
\mathcal{F}(\langle \texttt{apply } f \texttt{ with } x = p \rangle, \Gamma) &= \\
&\quad \{\, \mathit{subst}(q, x, p) : q \in \mathcal{F}(\Delta(f), \Gamma[x \mapsto \Gamma[p]]) \,\} \\
\mathcal{F}(\langle \texttt{for } x \texttt{ in } p \rangle\, i\, \langle \texttt{end} \rangle, \Gamma) &= \\
&\quad \bigcup_{k \in 1..n} \{\mathit{subst}(q, x, p[k]) : q \in \mathcal{F}(i, \Gamma[x \mapsto \Gamma[p][k]])\}
\end{aligned}
$$

Fig. 13. Definition of the footprint of a template evaluation

Definition 5. *Given a set of paths $P \in \mathcal{P}(\mathbb{P})$, we define the restriction of an environment $\Gamma \in \mathbb{E}$ to P as follows:*

$$
\begin{aligned}
\mathit{restrict} &: \mathbb{E} \times \mathcal{P}(\mathbb{P}) \to \mathbb{E} \\
\mathit{restrict}(\Gamma, P)[p] &= \Gamma[p] \ \textit{if } p \in P \\
\mathit{restrict}(\Gamma, P)[p] &\ \textit{is unbound otherwise.}
\end{aligned}
$$

The fact that the footprint indeed corresponds to the paths that are necessary for the evaluation of the template is given by the following theorems:

Theorem 13. *Let $\mathit{paths}(\Gamma)$ represent all the paths bound in Γ. Then*

1. $(\!|i|\!)(\Gamma) = (\!|i|\!)(\mathit{restrict}(\Gamma, \mathcal{F}(i, \Gamma)))$
2. *If $p \in \mathcal{F}(i, \Gamma)$, then $(\!|i|\!)(\mathit{restrict}(\Gamma, \mathit{paths}(\Gamma) \backslash \{p\})) = \bot$*

Bibliography

[1] Abramov, S.M., Glück, R.: Principles of inverse computation and the universal resolving algorithm. In: The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones [on occasion of his 60th birthday], Lecture Notes in Computer Science, vol. 2566, pp. 269–295, Springer (2002)

[2] Bagnara, R., Hill, P.M., Zaffanella, E.: Widening operators for powerset domains. In: 5th International Conference on Verification, Model Checking, and Abstract Interpretation, (VMCAI), Lecture Notes in Computer Science, vol. 2937, pp. 135–148, Springer (2004)

[3] Bancilhon, F., Spyratos, N.: Update semantics of relational views. ACM Trans. Database Syst. **6**(4), 557–575 (1981)

[4] Barbosa, D.M.J., Cretin, J., Foster, N., Greenberg, M., Pierce, B.C.: Matching lenses: alignment and view update. In: 15th ACM SIGPLAN International Conference on Functional programming (ICFP), pp. 193–204, ACM (2010)

[5] Birman, A., Ullman, J.D.: Parsing algorithms with backtrack. Inf. Control. **23**(1), 1–34 (1973), `https://doi.org/10.1016/S0019-9958(73)90851-6`

[6] Chang, B.Y.E., Rival, X.: Relational inductive shape analysis. In: 35th Symposium on Principles of Programming Languages, pp. 247–260, POPL '08, ACM, New York, NY, USA (2008)

[7] Cousot, P.: Semantic foundations of program analysis. In: Program Flow Analysis: Theory and Applications, chap. 10, pp. 303–342, Prentice-Hall (1981)

[8] Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: 4th Symposium on Principles of Programming Languages (POPL), pp. 238–252 (1977)

[9] Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Sixth Symposium on Principles of Programming Languages (POPL), pp. 269–282, ACM Press, New York, NY, San Antonio, Texas (1979)

[10] Cousot, P., Cousot, R.: Abstract interpretation frameworks. Journal of Logic and Computation **2**(4), 511–547 (auf 1992)

[11] Czarnecki, K., Foster, J.N., Hu, Z., Lämmel, R., Schürr, A., Terwilliger, J.F.: Bidirectional transformations: A cross-discipline perspective. In: 2nd International Conference on Theory and Practice of Model Transformations (ICMT@TOOLS), Lecture Notes in Computer Science, vol. 5563, pp. 260–283, Springer (2009)

[12] Deutsch, A.: A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In: International Conference on Computer Languages (ICCL), pp. 2–13, IEEE Computer Society (1992), `https://doi.org/10.1109/ICCL.1992.185463`

[13] Engelfriet, J., Maneth, S.: A comparison of pebble tree transducers with macro tree transducers. Acta Informatica **39**(9), 613–698 (2003)

[14] Filé, G., Ranzato, F.: The powerset operator on abstract interpretations. Theor. Comput. Sci. **222**(1-2), 77–111 (1999)
[15] Ford, B.: Packrat parsing: : simple, powerful, lazy, linear time, functional pearl. In: Seventh International Conference on Functional Programming (ICFP), pp. 36–47, ACM (2002)
[16] Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In: 32nd Symposium on Principles of Programming Languages (POPL), pp. 233–246, ACM (2005)
[17] Foster, N., Matsuda, K., Voigtländer, J.: Three complementary approaches to bidirectional programming. In: International Spring School on Generic and Indexed Programming (SSGIP), Lecture Notes in Computer Science, vol. 7470, pp. 1–46, Springer (2010)
[18] Frisch, A., Hosoya, H.: Towards practical typechecking for macro tree transducers. In: 11th International Symposium on Database Programming Languages, (DBPL), LNCS, vol. 4797, pp. 246–260, Springer (2007)
[19] Giacobazzi, R., Quintarelli, E.: Incompleteness, counterexamples, and refinements in abstract model-checking. In: 8th Static Analysis Symposium (SAS), LNCS, vol. 2126, pp. 356–373, Springer (2001)
[20] Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation (3rd edition). Pearson (2007)
[21] Hu, Z., Schürr, A., Stevens, P., Terwilliger, J.F.: Dagstuhl seminar on bidirectional transformations (BX). SIGMOD Rec. **40**(1), 35–39 (2011)
[22] Kahn, G.: Natural semantics. In: STACS 87, pp. 22–39, Springer Berlin Heidelberg (1987)
[23] Kay, M.: Algorithm schemata and data structures in syntactic processing. In: Readings in natural language processing, pp. 35–70 (1986)
[24] Ko, H., Hu, Z.: An axiomatic basis for bidirectional programming. Principles of Programming Languages **2**(POPL), 41:1–41:29 (2018)
[25] Kowalski, R.A.: Predicate logic as programming language. In: Rosenfeld, J.L. (ed.) 6th IFIP Congress on Information Processing, pp. 569–574, North-Holland (1974)
[26] Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: 33rd Symposium on Principles of Programming Languages (POPL), pp. 42–54, ACM (2006)
[27] Maneth, S., Berlea, A., Perst, T., Seidl, H.: XML type checking with macro tree transducers. In: 24th Symposium on Principles of Database Systems, pp. 283–294, ACM (2005)
[28] Matiyasevich, Y.: Диофантовость перечислимых множеств [Enumerable sets are Diophantine]. Doklady Akademii Nauk SSSR **191**(2), 279–282 (1970), English translation in Soviet Mathematics 11 (2), pp. 354–357.
[29] Matsuda, K., Hu, Z., Nakano, K., Hamana, M., Takeichi, M.: Bidirectionalization transformation based on automatic derivation of view complement functions. In: 12th International Conference on Functional Programming (ICFP), pp. 47–58, ACM (2007)
[30] Matsuda, K., Inaba, K., Nakano, K.: Polynomial-time inverse computation for accumulative functions with multiple data traversals. High. Order Symb. Comput. **25**(1), 3–38 (2012)

[31] Matsuda, K., Mu, S., Hu, Z., Takeichi, M.: A grammar-based approach to invertible programs. In: 19th European Symposium on Programming Languages and Systems (ESOP), Lecture Notes in Computer Science, vol. 6012, pp. 448–467, Springer (2010)
[32] Matsuda, K., Wang, M.: Flippr: A system for deriving parsers from pretty-printers. New Gener. Comput. **36**(3), 173–202 (2018)
[33] Methni, A., Ohayon, E., Thurieau, F.: ASTERIOS Checker : A Verification Tool for Certifying Airborne Software. In: 10th European Congress on Embedded Real Time Systems (ERTS 2020), Toulouse, France (Jan 2020)
[34] Milo, T., Suciu, D., Vianu, V.: Typechecking for XML transformers. J. Comput. Syst. Sci. **66**(1), 66–97 (2003)
[35] Mu, S., Hu, Z., Takeichi, M.: An injective language for reversible computation. In: 7th International Conference on Mathematics of Program Construction (MPC), LNCS, vol. 3125, pp. 289–313, Springer (2004)
[36] Nishida, N., Sakai, M.: Completion after program inversion of injective functions. Electron. Notes Theor. Comput. Sci. **237**, 39–56 (2009)
[37] Onzon, E.: dypgen: self-extensible parsers and lexers for OCaml (2012), URL `http://dypgen.free.fr`
[38] Parr, T.J.: Enforcing strict model-view separation in template engines. In: 13th international conference on World Wide Web (WWW), pp. 224–233, ACM (2004)
[39] Perst, T., Seidl, H.: Macro forest transducers. Inf. Process. Lett. **89**(3), 141–149 (2004)
[40] Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS), Lecture Notes in Computer Science, vol. 1384, pp. 151–166, Springer (1998)
[41] Ranzato, F.: Complete abstractions everywhere. In: 14th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI), LNCS, vol. 7737, pp. 15–26, Springer (2013)
[42] Ranzato, F., Zanella, M.: Robustness verification of support vector machines. In: 26th International Symposium on Static Analysis (SAS), Lecture Notes in Computer Science, vol. 11822, pp. 271–295, Springer (2019)
[43] Rival, X.: Understanding the origin of alarms in Astrée. In: 12th International Symposium on Static Analysis (SAS), Lecture Notes in Computer Science, vol. 3672, pp. 303–319, Springer (2005)
[44] Schmidt, D.A.: Abstract interpretation from a denotational-semantics perspective. In: 25th Conference on Mathematical Foundations of Programming Semantics (MFPS), Electronic Notes in Theoretical Computer Science, vol. 249, pp. 19–37, Elsevier (2009)
[45] Seidl, H., Maneth, S., Kemper, G.: Equivalence of deterministic top-down tree-to-string transducers is decidable. J. ACM **65**(4), 21:1–21:30 (2018)
[46] Shieber, S.M., Schabes, Y., Pereira, F.C.N.: Principles and implementation of deductive parsing. J. Log. Program. **24**(1&2), 3–36 (1995)
[47] Tozawa, A.: Towards static type checking for XSLT. In: Symposium on Document Engineering, pp. 18–27, ACM (2001)
[48] Yellin, D.M., Mueckstein, E.M.: The automatic inversion of attribute grammars. IEEE Trans. Software Eng. **12**(5), 590–599 (1986)