Fundamenta Informaticae 158 (2018) 297–326
DOI 10.3233/FI-2018-1650
IOS Press

Reference Abstract Domains and Applications to String Analysis

Roberto Amadini
School of Computing and Information Systems
The University of Melbourne, Australia
roberto.amadini@unimelb.edu.au

François Gauthier
Oracle Labs
Brisbane, Australia
francois.gauthier@oracle.com

Peter Schachte
School of Computing and Information Systems
The University of Melbourne, Australia
schachte@unimelb.edu.au

Peter J. Stuckey
School of Computing and Information Systems
The University of Melbourne, Australia
pstuckey@unimelb.edu.au

Graeme Gange
School of Computing and Information Systems
The University of Melbourne, Australia
gkgange@unimelb.edu.au

Alexander Jordan
Oracle Labs
Brisbane, Australia
alexander.jordan@oracle.com

Harald Søndergaard*
School of Computing and Information Systems
The University of Melbourne, Australia
harald@unimelb.edu.au

Chenyi Zhang†
College of Information Science and Technology
Jinan University, Guangzhou, China
chenyi_zhang@jnu.edu.cn

Abstract. Abstract interpretation is a well established theory that supports reasoning about the run-time behaviour of programs. It achieves tractable reasoning by considering abstractions of run-time states, rather than the states themselves. The chosen set of abstractions is referred to as the abstract domain. We develop a novel framework for combining (a possibly large number of) abstract domains. It achieves the effect of the so-called reduced product without requiring a quadratic number of functions to translate information among abstract domains. A central notion is a reference domain, a medium for information exchange. Our approach suggests a novel and simpler way to manage the integration of large numbers of abstract domains. We instantiate our framework in the context of string analysis. Browser-embedded dynamic programming languages such as JavaScript and PHP encourage the use of strings as a universal data type for both code and data values. The ensuing vulnerabilities have made string analysis a focus of much recent research. String analysis tends to combine many elementary string abstract domains, each

*Address for correspondence: School of Computing and Information Systems, The University of Melbourne, Victoria 3010, Australia.
†Work performed while at Oracle Labs.

Received December 2016; revised October 2017.

designed to capture a specific aspect of strings. For this instance the set of regular languages, while too expensive to use directly for analysis, provides an attractive reference domain, enabling the efficient simulation of reduced products of multiple string abstract domains.

1. Introduction

Strings are extensively used in modern programming languages, and the interest in string analyses is active and growing in many fields, ranging from model checking to automated testing and web security [3, 4, 12, 19, 21, 27, 30]. Scripting languages such as JavaScript and PHP make extensive use of *dynamic* string expressions to be evaluated at runtime. Reasoning about the runtime behaviour of scripts written in these languages usually requires non-trivial analysis of the string operations that can be performed at run-time. In fact, precise reasoning about strings takes a prominent role in these languages, owing to the dynamic features such as reflection and dynamic field access; even the construction of reliable call graphs may depend on precise string analysis.

The task of the analysis is to determine, for each program point, the set of values a given string variable or expression can take. For reasons of undecidability, the statements that can be extracted from such string values are necessarily approximate. Abstract interpretation [13] is a well-established theory of reasoning with approximations, or *abstractions*. The language that is available to an analysis (in our case, the set of descriptions that can be used to denote string sets) is an *abstract domain*. In the following we shall refer to it as a string abstract domain, or just *string domain*.

Numerous string domains have been proposed [11, 12, 27, 30, 33]. Practical string analysis tools [24, 26, 29] tend to apply combinations of string domains, often involving a considerable number of component domains. The reason is that many different aspects of a string are relevant for precise reasoning: how long the string may or must be, which characters it may or must contain, whether it may/must represent a numerical entity, and so on. Thus implementors are faced with the practical problem of how to approximate string operations, such as concatenation and substring selection, in analyses that involve combinations of a large number of domains, without undue cost or loss of precision.

Abstract interpretation provides the tools needed to model this practical problem mathematically. In this paper, we apply the well-known concept of *reduced product* [7, 14] to capture the notion of optimal string analysis over combinations of string domains. Informally, we can see the reduced product as the most abstract domain among those that preserve combinations of properties from all component domains.

Operations on the reduced product can easily be defined mathematically, but for practical use, implementation-oriented definitions are needed. In practice, designers often settle for an approach in which operations are performed component-wise, followed or interspersed with inter-component information exchange, by "paraphrasing" information deduced in one component domain to the languages of the other domains. The result is commonly an analysis which is coarser than the ideal, that is, less precise than the one that uses the reduced product proper. Moreover, where many domains are involved, this way of realising (or approximating) the reduced product quickly becomes cumbersome, and the runtime cost of information exchange can become prohibitive. As pointed out by Cousot *et al*. [17], a direct implementation of the reduced product is difficult to extend, as

> *the implementation of the most precise reduction (if it exists) can hardly be modular since in general adding a new abstract domain to increase precision implies that the reduced product must be completely redesigned.*

In this paper, we propose a general and modular framework for the systematic design and implementation of certain reduced product abstract domains, based around a suitably chosen *reference domain*. We see this primarily as a contribution to theory, but to demonstrate its utility, we instantiate it for sophisticated string analysis.

We assume that the component domains are (order isomorphic to) *upper closure operators* on the lattice of regular languages (for some fixed alphabet Σ). This is not an onerous requirement; certainly it appears to be satisfied by most string abstract domains used in practice, albeit not by all.

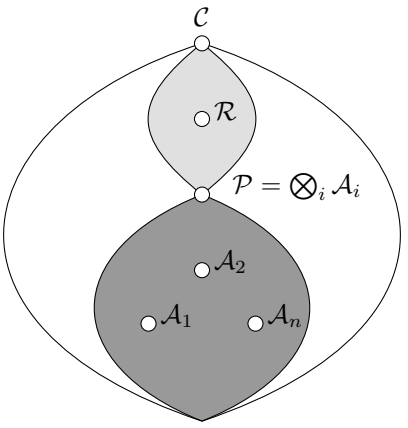


Figure 1. A *reference domain* $\mathcal{R}$ and the *reduced product* $\mathcal{P}$ in the space of abstract domains (lower means coarser)

Before we build the formal machinery, let us outline the idea. The diagram in Figure 1 should help the reader who is already familiar with the idea of the reduced product. Towards the bottom of the diagram are n incomparable abstract domains $\mathcal{A}_1, \ldots, \mathcal{A}_n$. At the top is the concrete domain $\mathcal{C}$. The latter could be, for example, the set of all languages over some alphabet, and the abstract domains could be the collection of string abstract domains used for this paper's examples. In the center of the diagram is the reduced product $\mathcal{P}$ of $\mathcal{A}_1, \ldots, \mathcal{A}_n$. Its location in the diagram indicates that it is, in a precise sense, an abstraction of $\mathcal{C}$, as are $\mathcal{A}_1, \ldots, \mathcal{A}_n$. $\mathcal{P}$ is, however, the most abstract domain that still manages to match the expressiveness of each $\mathcal{A}_i$.

The dark grey area below $\mathcal{P}$ contains abstract domains $\mathcal{P}'$ that are abstractions (that is, approximations) of the reduced product. A variety of "products" have been proposed previously; these are different approximations of the reduced product $\mathcal{P}$ and we discuss them in more detail towards the end of the paper. For now it suffices to note that, in the diagram, they all live in the dark grey area.

We, on the other hand, approach the reduced product "from above". A reference domain $\mathcal{R}$ is any suitable domain from the light grey area (we later elaborate "suitable"). We show how the full effect of using $\mathcal{P}$ can be obtained through the use of $\mathcal{R}$ as a medium in which $\mathcal{A}_1, \ldots, \mathcal{A}_n$ exchange information.

For a simple example, consider the reduced product of two incomparable numeric abstract domains, the well-known interval (or Box) domain and Karr's domain of affine equations [25]. An element of the former may tell us that $x \in [5, \infty], y \in [-\infty, \infty], z \in [1, 10]$. An element of the latter may tell us that $2x - y = 8$ and $x + 2z = 7$. It is not obvious how, in general, one component can inform the other; certainly a direct implementation of the necessary operators for optimal information

exchange seems a considerable undertaking. That is, a realisation of the reduced product seems hard. But we can take inspiration from the experience in mathematics that a hard problem can often be made easier by generalisation—solving instead a more general problem (Polya [34] refers to this as the Inventor's Paradox). Information from each abstract domain is easily turned into an element of the more expressive convex polyhedron domain [18], where a "meet" (or conjunction) of the elements can easily be calculated. For the example, the result is $x = 5, y = 2, z = 1$, information which is readily translated back to the component domains. It is generally agreed that in practice, the convex polyhedron domain is too expensive to use in abstract interpretation, but that need not stop it from serving as a powerful medium for information exchange, that is, as a reference domain.

The ultimate utility of a reference domain, however, is when a substantial number (say, $n > 3$) of incomparable abstract domains are in use. Then the reference-based approach that we propose has these advantages:

- Simplicity of implementation, as our approach requires $2n$ translation functions rather than a quadratic number.
- Modular implementation; incorporating yet another domain $\mathcal{A}_{n+1}$ is mostly straight-forward.
- Potentially lower computational cost at runtime.

Whether these advantages in fact ensue depends on the context, including the choice of reference domain—$\mathcal{D}$ must be chosen judiciously. We do not have a general recipe for this choice but we hope to demonstrate, through an example, what can be gained by approaching the "reduced product problem" from a new angle.

In short summary, the contributions of this paper are:

- A review of popular string abstract domains and their properties, with particular reference to their composition;
- A general framework, based on a notion of reference domain, for simulating the reduced product, that is, obtaining the same effect as the reduced product of a collection of abstract domains;
- An instantiation of this framework for string analyses, by using the lattice of regular languages as the reference domain.

Paper structure. Section 2 recapitulates some basic abstract interpretation concepts. The focus of Section 3 is on *string* abstract domains. Section 4 discusses the problem of combining such domains. Section 5 introduces the notion of a *reference* domain, while Section 6 exemplifies it, using the class of regular languages as reference domain, that is, a *lingua franca* for string abstract domains. Section 7 discusses some subtleties involved in *widening* for reduced product domains. Section 8 discusses related work, and Section 9 concludes.

2. Preliminaries

Abstract interpretation [13] is a well established theory supporting all types of reasoning about software's run-time behaviour. We assume the reader is familiar with basic concepts from abstract interpretation; this section merely sets the stage for what follows, fixing notation and terminology. Section 4 addresses the problem of how to combine abstract domains.

A *poset* is a set, equipped with a partial order. A binary relation $\sqsubseteq$, defined on a set D, is a *partial order* iff it is

1. reflexive: $\forall x \in D : x \sqsubseteq x$
2. transitive: $\forall x, y, z \in D : x \sqsubseteq y \wedge y \sqsubseteq z \rightarrow x \sqsubseteq z$
3. antisymmetric: $\forall x, y \in D : x \sqsubseteq y \wedge y \sqsubseteq x \rightarrow x = y$

As usual, $x \sqsubset y$ means $x \sqsubseteq y \wedge x \neq y$.

Two elements $x, y \in D$ are *comparable* iff $x \sqsubseteq y$ or $y \sqsubseteq x$. A poset D is a *chain* iff, for each pair $x, y \in D$, x and y are comparable.

Let $\langle D, \sqsubseteq \rangle$ be a poset. An element $x \in D$ is a *upper bound* for the set $D' \subseteq D$ iff $x' \sqsubseteq x$ for all $x' \in D'$. Dually we may define a *lower bound* for D'. An upper bound x for D' is the *least upper bound* for D' iff, for every upper bound x' for D', $x \sqsubseteq x'$. When it exists we denote this least upper bound by $\bigsqcup D'$. Dually we may define the *greatest upper bound* and denote it by $\bigsqcap D'$. We write $x \sqcup y$ for $\bigsqcup\{x, y\}$ and $x \sqcap y$ for $\bigsqcap\{x, y\}$.

A poset $\langle D, \sqsubseteq \rangle$ is a *lattice* iff every finite subset $D' \subseteq D$ has a least upper bound and a greatest lower bound. It is a *complete lattice* iff this condition applies to every subset, finite or not. The lattice D is *bounded* iff it has a unique least element (often written $\bot$) and a unique greatest element (often written $\top$). When we want to name its characteristic operations, we write the bounded lattice D as $\langle D, \sqsubseteq, \bot, \top, \sqcap, \sqcup \rangle$. Note that a complete lattice is necessarily bounded.

Let $\langle D, \sqsubseteq \rangle$ and $\langle D', \leq \rangle$ be posets. A function $f : D \rightarrow D'$ is *monotone* iff $\forall x, y \in D : x \sqsubseteq y \Rightarrow f(x) \leq f(y)$. A function $f : D \rightarrow D$ is *idempotent* iff $\forall x \in D : f(f(x)) = f(x)$. It is *reductive* iff $\forall x \in D : f(x) \sqsubseteq x$, and *extensive* iff $\forall x \in D : x \sqsubseteq f(x)$. A function which is monotone and idempotent is a *closure operator*. A *lower closure operator* is a reductive closure operator, whereas an *upper closure operator* is extensive.

We shall use different symbols for partial order relations, such as $\sqsubseteq$ and $\leq$. Sometimes $\sqsubseteq_D$ is used to denote the partial order associated with domain D. We use $\mathcal{P}$ as the powerset operator; $\subseteq$, $\cap$ and $\cup$ have their usual set-theoretic meanings.

In this paper a "concrete" domain is assumed to be a complete lattice. In fact, since we shall be primarily interested in a concrete domain whose elements are *sets* (of strings), we consider bounded lattices $\langle \mathcal{P}(\mathsf{S}), \subseteq, \emptyset, \mathsf{S}, \cap, \cup \rangle$. An abstract domain is a set $\mathcal{A}$, each element a of which denotes a set in $\mathcal{P}(\mathsf{S})$; we write this set as $\gamma(a)$. We assume that $\gamma(a) = \mathsf{S}$ for some $a \in \mathcal{A}$. A partial order $\sqsubseteq$ is induced on $\mathcal{A}$ by defining $a \sqsubseteq a'$ iff $\gamma(a) \subseteq \gamma(a')$. The resulting structure $\mathcal{A}$ may or may not have a unique least element, it may or may not be a lattice (many non-lattice abstract domains are found in the literature [20]) and, if it is a lattice, it may or may not be complete [15]. The function γ is referred to as the *concretisation* function. It is easy to generalise to the case where the codomain $\langle \mathcal{D}, \leq \rangle$ of γ is not a powerset, as long as it is a partial order.

Many desirable properties are obtained when γ has a (lower) adjoint $\alpha : \mathcal{D} \rightarrow \mathcal{A}$, referred to as the *abstraction* function. We then have a *Galois connection*: $\alpha(C) \sqsubseteq a \iff C \leq \gamma(a)$. In this case, α and γ are necessarily monotone functions, $\alpha \circ \gamma$ is a lower closure operator, and $\gamma \circ \alpha$ is an upper closure operator. Moreover, α and γ uniquely determine each other and every function $\varphi : \mathcal{D} \rightarrow \mathcal{D}$ has a unique optimal counterpart on $\mathcal{A}$, namely $\alpha \circ \varphi \circ \gamma$. In a program analysis context it is often the case that α is surjective, and we talk about a *Galois insertion* (aka *Galois surjection*). In that case, γ is injective and $\alpha \circ \gamma$ is the identity function.

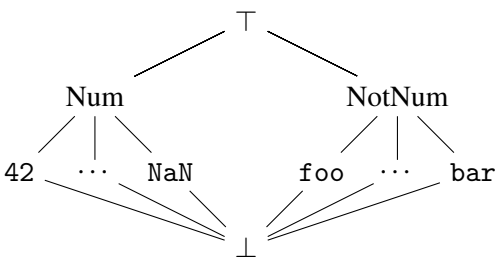


Figure 2. A simple lattice used for JavaScript string analysis

3. String abstract domains

A *string abstract domain* approximates concrete domain $\langle \mathcal{P}(\Sigma^*), \subseteq, \emptyset, \Sigma^*, \cap, \cup \rangle$ where Σ^* is the set of finite strings over finite alphabet Σ. We assume that $|\Sigma| > 1$.

The literature on string analysis contains a great variety of string abstract domains. For example, Costantini et al. [12] discuss four different domains, Madsen and Andreasen [30] twelve. Those abstract domains are language-agnostic, but we also find (combinations of) domains designed to capture information relevant to specific languages such as JavaScript [26, 29]. An example is the set of JavaScript "numeric strings"—strings that are acceptable in a context where a number literal is expected. This includes not only integer and float literals like `42` and `0.05`, but also special literals like `NaN` or `Infinity`. The Hasse diagram in Figure 2 shows an abstract domain used to discriminate between numeric and not-numeric strings in JavaScript. Its combination with a bounded-size "string set" domain is used in the SAFE analyzer [29]. A version that also identifies strings that name builtin methods such as `valueOf` is the default string domain in the JSAI analyzer [26].

In this section, we discuss a few language-agnostic string domains in some detail. We first consider the class of regular languages as an abstract domain. Then we turn attention to four more elementary string domains which have been used in previous works. The elementary domains offer fast implementations but suffer individually from limited expressiveness. Section 4 is concerned with the question of how to combine analyses that use a number of component string domains.

3.1. The regular language domain

One natural abstraction of a set of strings is a regular language. The class $\mathcal{RL}$ of regular languages is closed under concatenation, union, intersection, Kleene star, complement, and many other operations. Common string operations (concatenation, substring selection/test, substitution, ...) are readily expressed either as primitive automaton operations or finite state transducers. This domain and its variants have been applied to a range of practical problems [21, 32, 37].

A finite-state automaton is a tuple $R = \langle Q, \Sigma, \delta, q_0, F \rangle$, where Σ is the alphabet of symbols, Q is the set of states, $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times Q$ is a transition relation (where ϵ denotes a non-consuming transition), $q_0 \in Q$ is the initial state, and $F \subseteq Q$ the set of accept states. The *size* of the automaton R, denoted $|R|$, is the size of δ. The regular language recognised by R is written $\mathcal{L}(R)$. An automaton is a *deterministic* finite-state automaton (DFA) iff δ is a deterministic function on $Q \times \Sigma$. Let $q \to q'$ stand for $\exists x \in \Sigma(\delta(q, x) = q')$. The DFA $\langle Q, \Sigma, \delta, q_0, F \rangle$ is *trim* iff δ is a *partial* deterministic function on $Q \times \Sigma$, and for all $q \in Q \setminus \{q_0\}$, there is a $q' \in F$ such that $q_0 \to^+ q \wedge q \to^* q'$.

Concretely, δ can be represented as a transition table or adjacency lists (*symbolic automata* [37] use a more general "move" relation that captures sets of transitions, utilising intensional representations such as binary decision diagrams (BDDs) or character ranges). In Section 6 we will assume that regular languages are represented as trim DFAs.

The *regular* string domain is the bounded lattice $\langle \mathcal{RL}, \subseteq, \emptyset, \Sigma^*, \cap, \cup \rangle$. While $\mathcal{RL}$ is a lattice, it is not complete:

Proposition 3.1. $\mathcal{RL}$ is a lattice but not a complete lattice.

Proof:
Closure under union and intersection means $\mathcal{RL}$ is a lattice. However, there are chains of regular languages that have no regular least upper bound. Note that Σ^* is an upper bound of every set of regular languages. Hence the chain $[L_j]_{j\in\mathbb{N}}$ of regular languages $L_j = \{\mathtt{a}^i\mathtt{b}^i \mid 0 \leq i < j\}$ has an upper bound. Assume the regular language L is its *least* upper bound. Since L is an upper bound, every string of form $\mathtt{a}^i\mathtt{b}^i$ is in L, that is, $L' = \{\mathtt{a}^i\mathtt{b}^i \mid i \in \mathbb{N}\} \subseteq L$. But L' is not regular, so $L' \subset L$. Hence there is a string $s \in L \setminus L'$. Given this string s we have $L' \subset L \setminus \{s\} \subset L$, since $L \setminus \{s\}$ is also regular. Thus we have a contradiction and we conclude that $\mathcal{RL}$ is not a complete lattice. □

This lack of completeness is not a concern. It is not uncommon for lattices used as abstract domains to lack completeness; an example is the set of convex polyhedra used in the linear restraints analysis of Cousot and Halbwachs [18].

Rather, the main disadvantage of $\mathcal{RL}$ is the computational cost of its operations. Apart from the complement $\overline{\mathcal{L}(R)}$ computable in $O(|R|)$, other operations face a risk of explosion. An automaton R can have arbitrarily large size $|R|$, and the size of a DFA for $\mathcal{L}(R) \cap \mathcal{L}(R')$ (as for $\mathcal{L}(R) \cup \mathcal{L}(R')$) is, in the worst case, $|R||R'|$.

3.2. Elementary string domains

In the following we provide an overview of four simple string domains, often used in previous work on practical string analysis. For domain $\mathcal{D}$, we shall illustrate the abstraction and lattice operations, plus transformers for abstract concatenation ($x \cdot y$) and substring ($x[i..j]$, for constants $0 \leq i \leq j$).

3.2.1. Constant string ($\mathcal{CS}$)

This domain is a baseline to exactly represent a single, concrete string. It is the flat lattice $\langle \mathcal{CS}, \sqsubseteq_{\mathcal{CS}}, \bot_{\mathcal{CS}}, \top_{\mathcal{CS}}, \sqcap_{\mathcal{CS}}, \sqcup_{\mathcal{CS}} \rangle$ where

$$\mathcal{CS} = \{\bot_{\mathcal{CS}}\} \cup \Sigma^* \cup \{\top_{\mathcal{CS}}\}$$

and $\bot_{\mathcal{CS}} \sqsubseteq_{\mathcal{CS}} w \sqsubseteq_{\mathcal{CS}} \top_{\mathcal{CS}}$ for each $w \in \Sigma^*$. $\sqcap_{\mathcal{CS}}$ and $\sqcup_{\mathcal{CS}}$ are defined accordingly. The abstraction and concretisation functions are respectively:

$$\alpha_{\mathcal{CS}}(W) = \begin{cases} \bot_{\mathcal{CS}} & \text{if } W = \emptyset \\ w & \text{if } W = \{w\} \\ \top_{\mathcal{CS}} & \text{otherwise} \end{cases}$$

$$\gamma_{\mathcal{CS}}(w) = \begin{cases} \emptyset & \text{if } w = \bot_{\mathcal{CS}} \\ \Sigma^* & \text{if } w = \top_{\mathcal{CS}} \\ \{w\} & \text{otherwise} \end{cases}$$

Among the basic domains listed in this section, this is the only one which is able to keep track of a single concrete string w. $\top_{\mathcal{CS}}$ and $\bot_{\mathcal{CS}}$ are annihilators for the abstract concatenation and substring operations, which otherwise mimic their concrete counterparts.

Despite a limited expressive power, this domain is commonly used in practice [30]. A more precise extension of $\mathcal{CS}$ consists in representing at most k concrete strings, where $k \geq 1$ is a fixed parameter. This parametric generalisation is often referred to as String Set ($\mathcal{SS}$) [30]—a bounded-size variant is the main component of SAFE's default domain [29].

3.2.2. String length ($\mathcal{SL}$)

This domain keeps track of the minimum and the maximum length of strings. It is $\langle \mathcal{SL}, \sqsubseteq_{\mathcal{SL}}, \bot_{\mathcal{SL}}, \top_{\mathcal{SL}}, \sqcap_{\mathcal{SL}}, \sqcup_{\mathcal{SL}} \rangle$ where

$$\mathcal{SL} = \{\bot_{\mathcal{SL}}\} \cup \{[l, u] \mid l \in \mathbb{N}, u \in \mathbb{N} \cup \{\infty\}, l \leq u\}$$

and $[l, u] \sqsubseteq_{\mathcal{SL}} [l', u'] \iff l' \leq l \wedge u \leq u'$. The greatest element is $\top_{\mathcal{SL}} = [0, \infty]$. The meet and join preserve $\bot_{\mathcal{SL}}$, and for non-$\bot_{\mathcal{SL}}$ elements they are defined as follows: The meet $[l, u] \sqcap_{\mathcal{SL}} [l', u'] = [\max(l, l'), \min(u, u')]$ if $\max(l, l') \leq \min(u, u')$, $\bot_{\mathcal{SL}}$ otherwise. The join $[l, u] \sqcup_{\mathcal{SL}} [l', u'] = [\min(l, l'), \max(u, u')]$.

The abstraction and concretisation functions are respectively:

$$\alpha_{\mathcal{SL}}(W) = \begin{cases} \bot_{\mathcal{SL}} & \text{if } W = \emptyset \\ [min(L_W), max(L_W)] \text{ where } L_W = \{|w| \mid w \in W\} & \text{if } W \neq \emptyset \end{cases}$$

$$\gamma_{\mathcal{SL}}(a) = \begin{cases} \emptyset & \text{if } a = \bot_{\mathcal{SL}} \\ \{w \in \Sigma^* \mid l \leq |w| \leq u\} & \text{if } a = [l, u] \end{cases}$$

Abstract concatenation is defined by $[l, u] \odot_{\mathcal{SL}} [l', u'] = [l + l', u + u']$ while the substring operation $x[i..j]$ is abstracted by $x[i..j]_{\mathcal{SL}} = [k, k]$ where $k = j - i + 1$. Precise handling of the substring operation requires integer bounds analysis.

This abstraction uses natural numbers for abstracting strings. It is very simple and efficient to implement (all operations take $O(1)$). On the other hand, the precision is low: no character information is recorded. Also, the intervals abstraction is "convex": only the length bounds are considered.

3.2.3. Character inclusion ($\mathcal{CI}$)

This domain tracks the set of characters that *must* and *may* occur in a string. Each element of this domain is a "set interval" of characters $[L, U] = \{X \in \mathcal{P}(\Sigma) \mid L \subseteq X \subseteq U\}$. The lower bound L contains the characters of Σ that must occur in the concrete string(s), while the upper bound U represents the characters that may appear. Note that the latter is not void of information, unless $U = \Sigma$: the information provided by U is that no character in $\Sigma \setminus U$ appears in the string.

Formally, the domain is $\langle \mathcal{CI}, \sqsubseteq_{\mathcal{CI}}, \bot_{\mathcal{CI}}, \top_{\mathcal{CI}}, \sqcap_{\mathcal{CI}}, \sqcup_{\mathcal{CI}} \rangle$ where

$$\mathcal{CI} = \{\bot_{\mathcal{CI}}\} \cup \{[L, U] \mid L, U \in \mathcal{P}(\Sigma), L \subseteq U\}$$

and $[L, U] \sqsubseteq_{\mathcal{CI}} [L', U'] \iff L' \subseteq L \wedge U \subseteq U'$. The greatest element is $\top_{\mathcal{CI}} = [\varnothing, \Sigma]$. The meet operation is $[L, U] \sqcap_{\mathcal{CI}} [L', U'] = [L \cup L', U \cap U']$ except when $L \cup L' \not\subseteq U \cap U'$ in which case it returns $\bot_{\mathcal{CI}}$, while the join is $[L, U] \sqcup_{\mathcal{CI}} [L', U'] = [L \cap L', U \cup U']$.

Let $chars : \Sigma^* \to \mathcal{P}(\Sigma)$ return the set of characters occurring in a given string. The abstraction function is defined by

$$\alpha_{\mathcal{CI}}(W) = \begin{cases} \bot_{\mathcal{CI}} & \text{if } W = \emptyset \\ [\bigcap C_W, \bigcup C_W], \text{where } C_W = \{chars(w) \mid w \in W\} & \text{if } W \neq \emptyset \end{cases}$$

The concretisation function is defined by

$$\gamma_{\mathcal{CI}}(a) = \begin{cases} \emptyset & \text{if } a = \bot_{\mathcal{CI}} \\ \{w \in \Sigma^* \mid L \subseteq chars(w) \subseteq U\} & \text{if } a = [L, U] \end{cases}$$

Abstract concatenation is given by $[L, U] \odot_{\mathcal{CI}} [L', U'] = [L \cup L', U \cup U']$, while the substring operation $x[i..j]_{\mathcal{CI}}$ always returns $[\emptyset, U]$ unless $x = [\{a\}, \{a\}]$ (in which case x is returned).

This abstract domain is not computationally expensive (except when $|U|$ is big) and can provide useful information. However, it is not able to track a single, concrete string except for the empty string: $\alpha(\{\epsilon\}) = [\emptyset, \emptyset]$. In all other cases, abstraction completely loses the structure of the concrete strings.

3.2.4. Prefix/suffix ($\mathcal{PS}$)

An element of the prefix/suffix domain is a pair of strings $\langle p, s \rangle \in \Sigma^* \times \Sigma^*$, corresponding to all the concrete strings that start with p and end with s. The domain is $\langle \mathcal{PS}, \sqsubseteq_{\mathcal{PS}}, \bot_{\mathcal{PS}}, \top_{\mathcal{PS}}, \sqcap_{\mathcal{PS}}, \sqcup_{\mathcal{PS}} \rangle$ where

$$\mathcal{PS} = \{\bot_{\mathcal{PS}}\} \cup (\Sigma^* \times \Sigma^*).$$

Let $lcp(W)$ (respectively $lcs(W)$) be the longest common prefix (suffix) of a set of strings W. Then $\langle p, s \rangle \sqsubseteq_{\mathcal{PS}} \langle p', s' \rangle \iff lcp(\{p, p'\}) = p' \wedge lcs(\{s, s'\}) = s'$. The top element is $\top_{\mathcal{PS}} = \langle \epsilon, \epsilon \rangle$ The join is $\langle p, s \rangle \sqcup_{\mathcal{PS}} \langle p', s' \rangle = \langle lcp\{p, p'\}, lcs\{s, s'\} \rangle$. The meet is naturally induced by $\sqsubseteq_{\mathcal{PS}}$. We refer to Costantini *et al.* [12] for details. Abstraction is defined by

$$\alpha_{\mathcal{PS}}(W) = \begin{cases} \bot_{\mathcal{PS}} & \text{if } W = \emptyset \\ \langle lcp(W), lcs(W) \rangle & \text{if } W \neq \emptyset \end{cases}$$

Concretisation is defined by

$$\gamma_{\mathcal{PS}}(a) = \begin{cases} \emptyset & \text{if } a = \bot_{\mathcal{PS}} \\ \{p \cdot w \mid w \in \Sigma^*\} \cap \{w \cdot s \mid w \in \Sigma^*\} & \text{if } a = \langle p, s \rangle \end{cases}$$

The abstract concatenation is $\langle p, s \rangle \odot_{\mathcal{PS}} \langle p', s' \rangle = \langle p, s' \rangle$ while the substring selection is

$$\langle p, s \rangle [i..j]_{\mathcal{PS}} = \begin{cases} \langle p[i..j], p[i..j] \rangle & \text{if } j \leq |p| \\ \langle p[i..|p|], \epsilon \rangle & \text{if } i \leq |p| \leq j \\ \top_{\mathcal{PS}} & \text{otherwise} \end{cases}$$

Most of the operations of this domain cost $O(|p| + |s|)$.

3.3. Galois connections

The previous section provided abstraction and concretisation functions for four elementary string abstract domains. In each case we have a Galois insertion:

Proposition 3.2. For $i \in \{\mathcal{CS}, \mathcal{SL}, \mathcal{CI}, \mathcal{PS}\}$, (α_i, γ_i) is a Galois insertion.

Proof:
The four proofs follow the same simple pattern, so we cover only two of the cases, namely $(\alpha_{\mathcal{SL}}, \gamma_{\mathcal{SL}})$ and $(\alpha_{\mathcal{PS}}, \gamma_{\mathcal{PS}})$. First assume $\alpha_{\mathcal{SL}}(W) \sqsubseteq_{\mathcal{SL}} a$. If $W = \emptyset$ then $W \subseteq \gamma_{\mathcal{SL}}(a)$ holds trivially, so assume $W \neq \emptyset$. In this case $a = [l, u]$ where l and u are the min and max, respectively, of the set $\{|w| \mid w \in W\}$. So for each $w \in W$, we have $l \leq |w| \leq u$, that is, $W \subseteq \gamma_{\mathcal{SL}}([l, u])$.

Now assume $W \subseteq \gamma_{\mathcal{SL}}(a)$. If $a = \bot_{\mathcal{SL}}$ then $W = \emptyset$, so $\alpha_{\mathcal{SL}}(W) \sqsubseteq_{\mathcal{SL}} a$ by definition. So assume $a = [l, u]$. Then $\gamma_{\mathcal{SL}}(a) = \{w \in \Sigma^* \mid l \leq |w| \leq u\}$. Let $L_W = \{|w| \mid w \in W\}$. As $W \subseteq \gamma_{\mathcal{SL}}(a)$, we have $L_W \subseteq \{|w| \mid w \in \gamma_{\mathcal{SL}}(a)\} = \{l, \ldots, u\}$. Hence $min(L_W) \geq l$ and $max(L_W) \leq u$. That is, $\alpha_{\mathcal{SL}}(W) \sqsubseteq_{\mathcal{SL}} [l, u]$.

Hence $(\alpha_{\mathcal{SL}}, \gamma_{\mathcal{SL}})$ form a Galois connection. Moreover, $\alpha_{\mathcal{SL}}$ is clearly surjective. To produce $\alpha_{\mathcal{SL}}(W) = \bot_{\mathcal{SL}}$, choose $W = \emptyset$. To produce $\alpha_{\mathcal{SL}}(W) = [l, u]$ for arbitrary non-negative integers l and u, choose $W = \{w_l, w_u\}$ where w_l is any string of length l and w_u is any string of length u.

For the case $(\alpha_{\mathcal{PS}}, \gamma_{\mathcal{PS}})$, first assume that $\alpha_{\mathcal{PS}}(W) \sqsubseteq_{\mathcal{PS}} a$. If $W = \emptyset$ then $W \subseteq \gamma_{\mathcal{SL}}(a)$ holds, so assume $W \neq \emptyset$. Then $a = \langle p, s\rangle = \langle lcp(W), lcs(W)\rangle$. So each $w \in W$ is of the form $p \cdot w'$ and also of the form $w'' \cdot s$, for some $w', w'' \in \Sigma^*$. It follows that $W \subseteq \{p \cdot w' \mid w' \in \Sigma^*\} \cap \{w'' \cdot s \mid w'' \in \Sigma^*\} = \gamma_{\mathcal{PS}}(\langle p, s\rangle)$.

Now assume $W \subseteq \gamma_{\mathcal{PS}}(a)$. If $a = \bot_{\mathcal{PS}}$ then $W = \emptyset$, so $\alpha_{\mathcal{PS}}(W) \sqsubseteq_{\mathcal{PS}} a$. So assume $a = \langle p, s\rangle$. Then $\gamma_{\mathcal{PS}}(a) = \{p \cdot w' \mid w' \in \Sigma^*\} \cap \{w'' \cdot s \mid w'' \in \Sigma^*\}$. As $W \subseteq \gamma_{\mathcal{PS}}(a)$, we have $W \subseteq \{p \cdot w' \mid w' \in \Sigma^*\}$ and also $W \subseteq \{w'' \cdot s \mid w'' \in \Sigma^*\}$. That is, for each $w \in W$, p is a prefix of w and s is a suffix. It follows that p is a prefix of $lcp(W)$ and s is a suffix of $lcs(W)$. That is, $\alpha_{\mathcal{PS}}(W) \sqsubseteq_{\mathcal{PS}} \langle p, s\rangle$.

Again, $\alpha_{\mathcal{PS}}$ is surjective: To produce $\alpha_{\mathcal{PS}}(W) = \bot_{\mathcal{PS}}$, choose $W = \emptyset$. To produce $\alpha_{\mathcal{PS}}(W) = \langle p, s\rangle$, choose $x, y \in \Sigma$, with $x \neq y$ and let $W = \{p \cdot x \cdot s, p \cdot y \cdot s\}$. □

Inspection shows that, for each $i \in \{\mathcal{CS}, \mathcal{SL}, \mathcal{CI}, \mathcal{PS}\}$, and for each element a of abstraction i, $\gamma_i(a)$ is a regular language. So we can equally consider $(\widehat{\alpha}_i, \widehat{\gamma}_i)$, defined exactly like (α_i, γ_i), except that the domain of $\widehat{\alpha}_i$ (the co-domain of $\widehat{\gamma}_i$) is $\mathcal{RL}$ rather than $\mathcal{P}(\Sigma^*)$. Proposition 3.2 then carries over straight-forwardly:

Corollary 3.3. For $i \in \{\mathcal{CS}, \mathcal{SL}, \mathcal{CI}, \mathcal{PS}\}$, $(\widehat{\alpha}_i, \widehat{\gamma}_i)$ is a Galois insertion.

In the case of $\mathcal{PS}$ it is worth pointing out that we chose $\gamma_{\mathcal{PS}}$ with care. The definition $\gamma'(\langle p, s\rangle) = \{p \cdot w \cdot s \mid w \in \Sigma^*\}$ might seem like a plausible alternative, but it would not admit a Galois connection. For example, consider some $x \in \Sigma$. The use of γ' allows for $\langle x, \epsilon\rangle$ and $\langle \epsilon, x\rangle$ as minimal abstractions of the set $\{x\}$, and therefore offers no unique best abstraction. Note that $\alpha_{\mathcal{PS}}(\{x\}) = \langle x, x\rangle$ is the unique best abstraction under $\gamma_{\mathcal{PS}}$.

4. Combining abstract domains

The simplest way of obtaining an analysis that combines several different abstract domains is through their so-called *direct product*. Intuitively, the component domains are treated independently. This

means that the analysis can be systematically defined and performed in parallel. Suppose the n abstract domains $\langle \mathcal{A}_i, \sqsubseteq_i, \bot_i, \top_i, \sqcap_i, \sqcup_i \rangle$, $i = 1, \ldots, n$, all abstract a concrete domain $\mathcal{C}$. We can define their direct product as a structure $\langle \mathcal{A}, \sqsubseteq, \bot, \top, \sqcap, \sqcup, \rangle$ such that:

- $\mathcal{A} = \mathcal{A}_1 \times \cdots \times \mathcal{A}_n$
- $(a_1, \ldots, a_n) \sqsubseteq (a'_1, \ldots, a'_n) \iff a_i \sqsubseteq_i a'_i$ for all $i \in [1..n]$
- $\bot = (\bot_1, \ldots, \bot_n)$ and $\top = (\top_1, \ldots, \top_n)$
- $(a_1, \ldots, a_n) \sqcap (a'_1, \ldots, a'_n) = (a_1 \sqcap_1 a'_1, \ldots, a_n \sqcap_n a'_n)$
- $(a_1, \ldots, a_n) \sqcup (a'_1, \ldots, a'_n) = (a_1 \sqcup_1 a'_1, \ldots, a_n \sqcup_n a'_n)$
- $\gamma(a_1, \ldots, a_n) = \bigcap_{i=1}^{n} \gamma_i(a_i)$ and $\alpha(C) = (\alpha_1(C), \ldots, \alpha_n(C))$

In this approach, there is no exchange of information between the component domains. Performing a program analysis P with $\mathcal{A}$ is equivalent to performing n program analyses $P_1, \ldots, P_n$ where P_i uses abstract domain $\mathcal{A}_i$ only. A main drawback of the direct product is that γ may not be injective, which, from a practical perspective, suggests a loss of precision.

Example 4.1. Consider string analysis using the combined domain $\mathcal{SL} \times \mathcal{CI}$. The description $([0, 3], [\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}, \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}])$ represents the same concrete string set as $([3, 3], [\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}, \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}])$ since any string that contains all of `a`, `b` and `c` must have a length of 3 or more. By the component-wise ordering of the direct product,

$$([3, 3], [\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}, \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}]) \sqsubset ([0, 3], [\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}, \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}]).$$

So the components of the latter are unnecessarily imprecise. □

This shows that inability to exchange information between the components of a product domain can lead to a severe loss of precision. Ideally the combination should be more precise than the sum of its parts. The *reduced product* achieves this by forcing an injective γ. Consider the equivalence relation $\equiv$ defined by $(a_1, \ldots, a_n) \equiv (a'_1, \ldots, a'_n) \iff \gamma(a_1, \ldots, a_n) = \gamma(a'_1, \ldots, a'_n)$. The reduced product $\mathcal{A}' = \mathcal{A}_1 \otimes \ldots \otimes \mathcal{A}_n$ is the quotient set of $\equiv$, that is:

$$\mathcal{A}_1 \otimes \cdots \otimes \mathcal{A}_n = \{[(a_1, \ldots, a_n)]_{\equiv} \mid a_1 \in \mathcal{A}_1, \ldots, a_n \in \mathcal{A}_n\}$$

and we define (the injective) $\gamma : \mathcal{A}' \to \mathcal{C}$ and $\alpha : \mathcal{C} \to \mathcal{A}'$ by

$$\begin{aligned} \gamma([(a_1, \ldots, a_n)]_{\equiv}) &= \textstyle\bigcap_{i=1}^{n} \gamma_i(a_i) \\ \alpha(C) &= [(\alpha_1(C), \ldots, \alpha_n(C))]_{\equiv} \end{aligned}$$

If a greatest lower bound exists (for example, $\mathcal{A}_1, \ldots, \mathcal{A}_n$ are complete lattices) then $[(a_1, \ldots, a_n)]_{\equiv}$ may be identified with its minimal representative: $\bigsqcap [a_1, \ldots, a_n]_{\equiv}$.

Moreover, if each (γ_i, α_i) is a Galois connection then so is (γ, α).

4.1. Paraphrasing and combinatorial excess

Example 4.1 suggests the use of some kind of information exchange to translate insight from one component to other components, in order to calculate minimal representatives of equivalence classes. We call this improvement of one component using information from another *paraphrasing*. For example, the $\mathcal{CI}$ element $[\{\mathtt{a},\mathtt{b},\mathtt{c}\},\{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d}\}]$ can be seen as a "string length paraphraser" $\lambda v.\ v \sqcap_{\mathcal{SL}} [3,\infty]$ which tightens an $\mathcal{SL}$ component appropriately. More generally, the $\mathcal{SL}$ paraphraser corresponding to $[L,U] \in \mathcal{CI}$ is $\lambda v.\ v \sqcap_{\mathcal{SL}} [|L|,\infty]$. Similarly, a $\mathcal{PS}$ element $\langle p,s\rangle$ can be viewed as a $\mathcal{CI}$ paraphraser, as follows. The paraphraser $P^{\mathcal{CI}}_{\mathcal{PS}}$ is defined

$$P^{\mathcal{CI}}_{\mathcal{PS}}\langle p,s\rangle(v) = \begin{cases} [L \cup X, U] & \text{if } v=[L,U] \text{ and } L \cup X \subseteq U, \text{ where } X = \mathit{chars}(p \cdot s) \\ \bot_{\mathcal{CI}} & \text{otherwise} \end{cases}$$

Generally, if we have n abstract domains, we can have $n(n-1)$ such paraphrasers $P^j_i : \mathcal{A}_i \to \mathcal{A}_j \to \mathcal{A}_j$, so even for small n, a large number of paraphrasers may be needed. The strain of juggling many different kinds of information, delivered through different abstract domains, becomes prohibitive. This is very much the case in string analysis, since in this type of program analysis, so many abstract domains get applied in promiscuous combinations. If paraphrasers of type $(\mathcal{A}_1 \times \cdots \times A_k) \to \mathcal{A}_j \to \mathcal{A}_j$ are allowed (for $k > 1$), the number of possible paraphrasers is well beyond quadratic.

Even if each one-on-one paraphraser $P^j_i(a_i)$ is a lower closure operator, it may have to be applied repeatedly. In fact, the combined effect of paraphrasing until no more tightening is possible corresponds to computing the *greatest fixed point* of the operation P defined by

$$P(a_1,\ldots,a_n) = \left(a_1 \sqcap \prod_{i\in[1..n]} P^1_i(a_i)(a_1),\quad \ldots,\quad a_n \sqcap \prod_{i\in[1..n]} P^n_i(a_i)(a_n) \right) \tag{1}$$

In common practice, a limited selection of such paraphrasers, or approximations to them, are constructed by hand. Cost considerations may dictate that they are not applied exhaustively, leading to more or less *ad hoc* realisations of "not quite reduced" products.

Worse, one-on-one paraphrasing among the components of a direct product does not necessarily amount to an implementation of the reduced product, even if the individual paraphrasers are optimal. This is still true if we allow repeated use of paraphrasing, until a fixed point is reached:

Example 4.2. Let us fix the alphabet Σ to $\{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d}\}$. Consider the combination of three abstractions

$$x = [5,6] \in \mathcal{SL} \qquad y = [\Sigma,\Sigma] \in \mathcal{CI} \qquad z = \langle \mathtt{ab},\mathtt{aba}\rangle \in \mathcal{PS}$$

The system of paraphrasers for this example leads to an equation system whose solution is simply (x,y,z). That is, P, as defined in Equation 1, can provide no improvement (in this case it acts as the identity function). To see this, note that the knowledge (in y) that a string s uses the whole alphabet does not allow us to improve on x, nor on z. Conversely, neither x nor z can improve on y, since y is as precise as $\mathcal{CI}$ allows. Finally, it is easy to see that x and z cannot improve on each other.

In the reduced product $P = \mathcal{SL} \otimes \mathcal{CI} \otimes \mathcal{PS}$, however, the element (x,y,z) denotes $\emptyset$, that is, no concrete string is described by all of the three abstractions. Any string that is described by y and z is of the form $\mathtt{ab}\Sigma^*\mathtt{c}\Sigma^*\mathtt{d}\Sigma^*\mathtt{aba}$ or $\mathtt{ab}\Sigma^*\mathtt{d}\Sigma^*\mathtt{c}\Sigma^*\mathtt{aba}$, since it must contain all of Σ, and hence must have length at least 7.

Thus no amount of paraphrasing can lead to the correct reduction. For a similar reason, no amount of paraphrasing will suffice, in general, to implement optimal transfer functions. □

It is worth pointing out that the $\mathcal{SL}$, $\mathcal{CI}$, and $\mathcal{PS}$, domains themselves are reduced products of simpler domains which capture "minimum length", "maximum length", "definite inclusion", "possible inclusion", "prefix", and "suffix" information. However, in the literature, the domains $\mathcal{SL}$, $\mathcal{CI}$, and $\mathcal{PS}$, are almost always presented as we have done in this paper, that is, as three monolithic domains. Given the prominence of the "one-on-one paraphrasing" approach, that is a perfectly sensible choice, as we now show.

Example 4.3. Let Σ be as in Example 4.2, and consider these four pieces of information about a string s: (1) s has prefix `aba`, (2) s has suffix `aba`, (3) s has length 3 or more, and (4) s has length 4 or less. No amount of two-way information exchange will yield stronger information. However, given the $\mathcal{PS}$ information $\langle \texttt{aba}, \texttt{aba} \rangle$ and the $\mathcal{SL}$ information $[3, 4]$, one-on-one paraphrasing can yield the tighter length information $[3, 3]$. □

5. Reference abstract domains

Section 4's definition of reduced product is not constructive. In this section we propose a generic, simple, and modular way to achieve the effect of using a reduced product, through what we call a *reference domain*.

From a mathematical point of view, the reduced product of the n abstract domains $\mathcal{A}_1, \ldots, \mathcal{A}_n$ is straight-forward. However, a practical realisation, in the form of appropriate data structures and algorithms for its implementation, is often elusive. As we saw in Section 4, the common approach of using the Cartesian product and "paraphrasing" information between components falls short. What we propose is a way of overcoming the difficulty of realising a reduced product, in particular where a large number of abstract domains need to be combined. The idea behind the reference domain is that it will act as a mediator amongst all of the involved abstract domains. It is chosen so as to be as expressive as each $\mathcal{A}_i$. This way it is, if anything, "closer" to the concrete domain than the reduced product is.

In this section we provide a general framework for reference domains. Section 6 exemplifies the idea through one particular instantiation, in the context of string abstract domains. In fact it may seem that a reference domain annuls the need for a reduced product. If a reference domain is available, why not simply use *it* for program analysis in lieu of the reduced product? The answer is that the reference domain is likely to be too expensive to use as an abstract domain (otherwise surely we would have used it in the first place). Also, a highly expressive domain will often incorporate quite a coarse widening operation, so paradoxically, analysis with such an expressive domain can be *less* precise. In our proposal, the role of the reference domain is mediation, not exertion. Most of the work is left to be performed in the (cheap and simple) abstract domains $\mathcal{A}_1, \ldots, \mathcal{A}_n$. Apart from efficient translation to and from the reference domain, all that we require is an inexpensive meet operation in that domain.

Definition 5.1. (Reference domain)
Let the bounded lattices $\mathcal{A}_1, \ldots, \mathcal{A}_n$ and $\mathcal{R}$ be sound abstractions, of a concrete domain $\mathcal{C}$, defined by concretisation functions $\{\gamma_i\}_{i=1,\ldots,n}$ and $\gamma_{\mathcal{R}}$ respectively. The domain $\mathcal{R}$ is a *reference domain* for $\mathcal{A}_1, \ldots, \mathcal{A}_n$ if there is a family of functions $\{\widehat{\alpha}_i, \widehat{\gamma}_i\}_{i=1,\ldots,n}$ such that, for $i = 1, \ldots, n$:

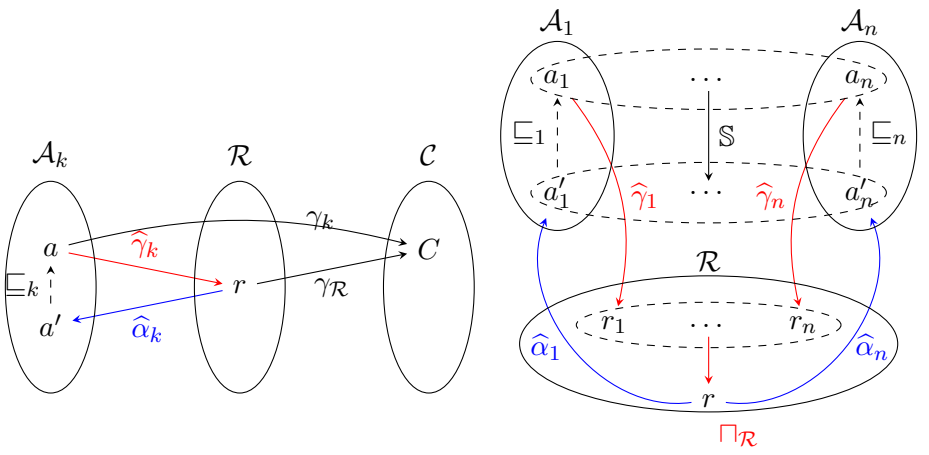


Figure 3. Left: Role of the reference domain $\mathcal{R}$ ($\alpha_{\mathcal{R}}$ and α_k might not be defined for some elements, as there may not be a best abstraction). Right: Role of the $\mathbb{S}$ operator.

(i) $\widehat{\alpha}_i : \mathcal{R} \to \mathcal{A}_i$ and $\widehat{\gamma}_i : \mathcal{A}_i \to \mathcal{R}$ are computable functions and $(\widehat{\alpha}_i, \widehat{\gamma}_i)$ forms a Galois connection;

(ii) $\gamma_{\mathcal{R}}$ is injective and $\gamma_{\mathcal{R}}(\widehat{\gamma}_i(a)) = \gamma_i(a)$ for each $a \in \mathcal{A}_i$.

The functions $\widehat{\alpha}_i$ will be called $\mathcal{R}$-abstractions, while the functions $\widehat{\gamma}_i$ will be called $\mathcal{R}$-concretisations. The joint effort by $\{\widehat{\alpha}_i, \widehat{\gamma}_i\}_{i=1,\ldots,n}$ is captured by $\widehat{\alpha}$ and $\widehat{\gamma}$ (without subscripts), defined as follows:

$$\begin{array}{lcl} \widehat{\alpha}(r) & = & (\widehat{\alpha}_1(r), \ldots, \widehat{\alpha}_n(r)) \\ \widehat{\gamma}(a_1, \ldots, a_n) & = & \bigsqcap_i \widehat{\gamma}_i(a_i) \end{array}$$

The definition of reference domain is generic in the sense that it is not bound to any specific concrete domain $\mathcal{C}$. Note that γ_i and γ_R are not required to have corresponding lower adjoints. Figure 3 (left) depicts the interactions between the abstract domains $\mathcal{A}_k$, the reference domain $\mathcal{R}$, and the concrete domain $\mathcal{C}$. We see a reference domain as a middleware layer between a collection of abstract domains and the corresponding concrete domain.

Since $\{\widehat{\alpha}_i, \widehat{\gamma}_i\}_{i=1,\ldots,n}$ are monotone, so are $\widehat{\alpha}$ and $\widehat{\gamma}$. In fact, $(\widehat{\alpha}, \widehat{\gamma})$ is a Galois connection between the direct product and the reference domain:

Proposition 5.2. Let $\mathcal{A}_1, \ldots, \mathcal{A}_n, \mathcal{R}, \widehat{\alpha}$ and $\widehat{\gamma}$ be defined as in Definition 5.1. Then $\widehat{\alpha}(r) \sqsubseteq (a_1, \ldots, a_n)$ iff $r \sqsubseteq_{\mathcal{R}} \widehat{\gamma}(a_1, \ldots, a_n)$.

Proof:

$$\begin{array}{lll} & \widehat{\alpha}(r) \sqsubseteq (a_1, \ldots, a_n) & \\ \text{iff} & \widehat{\alpha}_i(r) \sqsubseteq a_i \text{ for all } i \in \{1, \ldots, n\} & \text{(by definition of } \widehat{\alpha}) \\ \text{iff} & r \sqsubseteq_{\mathcal{R}} \widehat{\gamma}_i(a_i) \text{ for all } i \in \{1, \ldots, n\} & (\text{(}\widehat{\alpha}_i, \widehat{\gamma}_i\text{) being a Galois connection)} \\ \text{iff} & r \sqsubseteq_{\mathcal{R}} \bigsqcap\{\widehat{\gamma}_i(a_i) \mid 1 \leq i \leq n\} & \text{(by properties of } \textstyle\bigsqcap) \\ \text{iff} & r \sqsubseteq_{\mathcal{R}} \widehat{\gamma}(a_1, \ldots, a_n) & \text{(by definition of } \widehat{\gamma}) \end{array}$$

□

Given some function $f : \mathcal{C}^n \mapsto \mathcal{C}$ and best abstraction $F_{\mathcal{R}} : \mathcal{R}^n \mapsto \mathcal{R}$, we can then derive the corresponding optimal transformer $\widehat{F}$ over the reduced product:

$$\widehat{F}(a_1, \ldots, a_n) = (\widehat{\alpha}_1(r), \ldots, \widehat{\alpha}_n(r))$$

where $r = F_{\mathcal{R}}(\widehat{\gamma}(a_1) \sqcap_{\mathcal{R}} \ldots \sqcap_R \widehat{\gamma}(a_n))$.

Essentially, we execute f in the reference domain, and retrieve the most precise representation under the reduced product. This retrieval is critical: it restricts the size of the abstract state by "clamping" it down to the sub-domains between operations. However, applying transformers in the reference domain may be more expensive than we can tolerate. Next we discuss ways of approximating these optimal transformers.

5.1. Strengthening

Intuitively, $\mathcal{R}$ is a sound $\mathcal{C}$-approximation which is never less precise than any combination of properties from $\mathcal{A}_1, \ldots, \mathcal{A}_n$. The idea is to capture any information expressible in each of the abstract domains. Rather than applying the transformer for F under $\mathcal{R}$, we can instead use $\mathcal{R}$ as a medium for systematically transferring information from each domain to the others.

Definition 5.3. (Strengthening function)
Let $\mathcal{R}$ be a reference domain for $\mathcal{A}_1$, …, $\mathcal{A}_n$ with adjoint pairs $\{\widehat{\alpha}_i, \widehat{\gamma}_i\}_{i=1,\ldots,n}$. The corresponding *strengthening function* $\mathbb{S} : \mathcal{A}_1 \times \cdots \times \mathcal{A}_n$ is defined as follows:

$$\mathbb{S}(a_1, \ldots, a_n) = (\widehat{\alpha}_1(r), \ldots, \widehat{\alpha}_n(r))$$

where $r = \widehat{\gamma}(a_1, \ldots, a_n) = \widehat{\gamma}_1(a_1) \sqcap_{\mathcal{R}} \cdots \sqcap_{\mathcal{R}} \widehat{\gamma}_n(a_n)$.

Definition 5.3 captures how we intend to use a reference domain. As Figure 3 (right) shows, the operator $\mathbb{S}$ transforms a tuple $(a_1, \ldots, a_n)$ into a tuple $(a'_1, \ldots, a'_n)$ via the reference domain $\mathcal{R}$. The input tuple is "multiplexed" into element $r \in \mathcal{R}$ via the $\mathcal{R}$-concretisations $\widehat{\gamma}_i$ and the meet operator $\sqcap_{\mathcal{R}}$. Then, r is "de-multiplexed" into the tuple $(a'_1, \ldots, a'_n)$ thanks to the $\mathcal{R}$-abstractions $\widehat{\alpha}_i$.

We say that $\mathbb{S}$ is a strengthening since it is a closure operator returning an tuple, each component of which is as least as precise as the corresponding input components:

Proposition 5.4. A strengthening function $\mathbb{S} : (\mathcal{A}_1 \times \ldots \times \mathcal{A}_n) \to (\mathcal{A}_1 \times \ldots \times \mathcal{A}_n)$ is a lower closure operator.

Proof:
All three parts of the proof utilise the monotonicity of $\widehat{\alpha}_i$ and $\widehat{\gamma}_i$.

(i) [$\mathbb{S}$ is reductive.] To prove $\mathbb{S}(a_1, \ldots, a_n) \sqsubseteq (a_1, \ldots, a_n)$ we show that $\widehat{\alpha}_i(r) \sqsubseteq_{\mathcal{A}_i} a_i$ for each $i = 1, \ldots, n$, where r is given by Definition 5.3. Consider arbitrary $k \in \{1, \ldots, n\}$. Since $r = \bigsqcap_{i=1}^n \widehat{\gamma}_i(a_i)$, we have $r \sqsubseteq_{\mathcal{R}} \widehat{\gamma}_k(a_k)$ and therefore $\widehat{\alpha}_k(r) \sqsubseteq_{\mathcal{A}_k} \widehat{\alpha}_k(\widehat{\gamma}_k(a_k)) \sqsubseteq_k a_k$.

(ii) [$\mathbb{S}$ is monotone.] Let us assume $(a_1, \ldots, a_n) \sqsubseteq (a'_1, \ldots, a'_n)$. We wish to prove that, for each $i \in \{1, \ldots, n\}$, $\widehat{\alpha}_i(r) \sqsubseteq_{\mathcal{A}_i} \widehat{\alpha}_i(r')$, where $r = \bigsqcap_{i=1}^n \widehat{\gamma}_i(a_i)$ and $r' = \bigsqcap_{i=1}^n \widehat{\gamma}_i(a'_i)$. Consider and arbitrary $k \in \{1, \ldots, n\}$. From the hypothesis $a_k \sqsubseteq_{\mathcal{A}_k} a'_k$ we derive that $\widehat{\gamma}_k(a_k) \sqsubseteq_{\mathcal{A}_k} \widehat{\gamma}_k(a'_k)$ and thus $\bigsqcap_{i=1}^n \widehat{\gamma}_i(a_i) \sqsubseteq_{\mathcal{R}} \bigsqcap_{i=1}^n \widehat{\gamma}_i(a'_i)$, that is, $r \sqsubseteq_{\mathcal{R}} r'$. It follows that, for each $i \in \{1, \ldots, n\}$, $\widehat{\alpha}_i(r) \sqsubseteq_{\mathcal{A}_i} \widehat{\alpha}_i(r')$.

(iii) [$\mathbb{S}$ is idempotent.] We immediately have $\mathbb{S}(\mathbb{S}(a_1,\ldots,a_n)) \sqsubseteq \mathbb{S}(a_1,\ldots,a_n)$, since $\mathbb{S}$ is reductive *(i)*. It remains to show that $\mathbb{S}(a_1,\ldots,a_n) \sqsubseteq \mathbb{S}(\mathbb{S}(a_1,\ldots,a_n))$. That is, we need to show that $\widehat{\alpha}_i(r) \sqsubseteq_{\mathcal{A}_i} \widehat{\alpha}_i(r')$ for $i = 1,\ldots,n$ where $r = \bigsqcap_{j=1}^{n} \widehat{\gamma}_j(a_j)$ and $r' = \bigsqcap_{j=1}^{n} \widehat{\gamma}_j(\widehat{\alpha}_j(r))$. But for each $k \in \{1,\ldots,n\}$, we have $r \sqsubseteq_{\mathcal{R}} \widehat{\gamma}_k(\widehat{\alpha}_k(r))$. Hence $r \sqsubseteq_{\mathcal{R}} \bigsqcap_{j=1}^{n} \widehat{\gamma}_j(\widehat{\alpha}_j(r)) = r'$, and so $\widehat{\alpha}_i(r) \sqsubseteq_{\mathcal{A}_i} \widehat{\alpha}_i(r')$, as required. □

Lemma 5.5 lists some properties of $\mathbb{S}$ *vis-à-vis* the concretisation function γ, including soundness: the concrete counterparts of each abstract element are always preserved by $\mathbb{S}$.

Lemma 5.5. Let $\mathbb{S}$ be a strengthening function for $\mathcal{A}_1 \times \cdots \times \mathcal{A}_n$. We have:

(i) $\gamma(a) = \gamma(\mathbb{S}(a))$ for $a \in \mathcal{A}_1 \times \cdots \times \mathcal{A}_n$

(ii) $\gamma_i(a) = \gamma_i(a') \iff \widehat{\gamma}_i(a) = \widehat{\gamma}_i(a')$ for $i = 1,\ldots,n$ and $a, a' \in \mathcal{A}_i$.

Proof:

(i) The inclusion $\gamma(\mathbb{S}(a)) \sqsubseteq \gamma(a)$ is immediate from the monotonicity of γ and reductiveness of $\mathbb{S}$. We prove that $\gamma(a) \sqsubseteq \gamma(\mathbb{S}(a))$. Note that $\gamma(a) = \gamma_{\mathcal{R}}(\widehat{\gamma}(a))$ since $\gamma(a_i) = \gamma_{\mathcal{R}}(\widehat{\gamma}_i(a_i))$ by definition. So $\gamma(a) = \gamma_{\mathcal{R}}(\widehat{\gamma}(a)) \sqsubseteq \gamma_{\mathcal{R}}(\widehat{\gamma}(\mathbb{S}(a))) = \gamma(\mathbb{S}(a))$.

(ii) Consider $a_i, a_i' \in A_i$, with $\gamma_i(a_i) = \gamma_i(a_i')$. Let $r = \widehat{\gamma}_i(a_i), r' = \widehat{\gamma}_i(a_i')$. By Definition 5.1(ii), $\gamma_{\mathcal{R}}(r) = \gamma_i(a_i) = \gamma_i(a_i') = \gamma_{\mathcal{R}}(r')$. But $\gamma_{\mathcal{R}}$ is injective, so $r = r'$. Therefore $\widehat{\gamma}_i(a_i) = \widehat{\gamma}_i(a_i')$. Conversely, assume $\widehat{\gamma}_i(a_i) = \widehat{\gamma}_i(a_i')$. By Definition 5.1(ii), $\gamma_i(a_i) = \gamma_{\mathcal{R}}(\widehat{\gamma}_i(a_i)) = \gamma_{\mathcal{R}}(\widehat{\gamma}_i(a_i')) = \gamma_i(a_i')$. □

Theorem 5.6. Let $\mathbb{S}$ be a strengthening function. Given $a \in A_1 \times \ldots \times A_n$, $\mathbb{S}(a)$ is the least element of $[a]_{\equiv}$, that is, $\mathbb{S}(a) = \min_{\sqsubseteq}\{a' \mid \gamma(a') = \gamma(a)\}$.

Proof:
Let $a = (a_1,\ldots,a_n) \in A_1 \times \ldots \times A_n$, and $r = \widehat{\gamma}(a)$. Then $\mathbb{S}(a) = (\widehat{\alpha}_1(r),\ldots,\widehat{\alpha}_n(r))$ (by Definition 5.3).

Now let $a' = (a_1',\ldots,a_n') \in [a]_{\equiv}$. Then $\gamma(a) = \gamma(a')$, so by Lemma 5.5 we have $\mathbb{S}(a) \in [a]_{\equiv}$, since $\gamma(a) = \gamma(\mathbb{S}(a))$, $\widehat{\gamma}(a) = \widehat{\gamma}(a') = r$.

Consider some a_i'. We have $r \sqsubseteq \widehat{\gamma}_i(a_i')$. As $(\widehat{\alpha}_i, \widehat{\gamma}_i)$ is a Galois connection, $\widehat{\alpha}_i(r) \sqsubseteq a_i'$. As for each i we have $\widehat{\alpha}_i(r) \sqsubseteq a_i'$, we conclude $\mathbb{S}(a) \sqsubseteq a'$. Thus, $\mathbb{S}(a)$ is the least element of $[a]_{\equiv}$. □

Theorem 5.6 states the equivalence between the reduced product $\mathcal{A}_1 \otimes \cdots \otimes \mathcal{A}_n$ and the product induced by the strengthening $\mathbb{S}$ on $\mathcal{A}_1 \times \cdots \times \mathcal{A}_n$. It tells us that a strengthening operator $\mathbb{S}$ is sufficient to systematically mimic the reduced product of an arbitrary number of abstract domains.

This gives us a cheaper approximation of $\widehat{F}$: we apply the transformer for f in each sub-domain, and use $\mathbb{S}$ to transfer information between domains:

$$\widehat{F}_{\mathbb{S}}(a_1,\ldots,a_n) = \mathbb{S}(F_1(a_1),\ldots,F_n(a_n)).$$

This sacrifices precision relative to the optimal $\widehat{F}$, but reduces the computation we must do in $\mathcal{R}$.

5.2. Pros and cons

There have been other proposals for the combination of abstract domains. Section 8 discusses related work, but at this point we briefly reflect on benefits obtained with the reference domain approach.

First, the approach is generic; it is entirely transparent to the underlying domains and applicable to any type of abstract domain, including numerical abstract domains.

Second, it is simple in the sense that it is not parameterised by families of mediating functions, as seen in other approaches.

Third, it achieves a certain scalability and modularity: it is easier to realise than methods that rest upon a quadratic number of paraphrasing functions between abstract domains $\mathcal{A}_1, \ldots, \mathcal{A}_n$. We require only $2n$ translators $\{\widehat{\alpha}_i, \widehat{\gamma}_i\}_{i=1,\ldots,n}$, not a quadratic number. To add a new domain $\mathcal{A}_{n+1}$ to the product, we do not require knowledge of the other n domains for computing $\mathcal{A}_1 \otimes \cdots \otimes \mathcal{A}_n \otimes \mathcal{A}_{n+1}$. A pivotal property of $\mathbb{S}$ is computability: while systematically computing the reduced product is not possible in general by definition of γ, it is always possible to compute $\mathbb{S}$ in a finite time.

A drawback of $\mathbb{S}$ is that $\sqcap_{\mathcal{R}}$ can be arbitrarily costly depending on the considered domains. When $\sqcap_{\mathcal{R}}$ is too expensive to compute, particularly for products of many domains, it may be too costly to compute $\mathbb{S}(a_1, \ldots, a_n)$. In such cases we may prefer to adopt the 'conventional' approach of using each abstraction to strengthen the others, by defining a relaxed strengthening function $\widetilde{\mathbb{S}}$ from $n(n-1)$ pairwise strengthening functions $\mathbb{S}^i_j : \mathcal{A}_i \times \mathcal{A}_j \to \mathcal{A}_i$ such that $\mathbb{S}^i_j(a_i, a_j) \sqsubseteq_i a_i$ possibly refines element a_i with respect to the information provided by element a_j. The resulting 'weak' strengthening function $\widetilde{\mathbb{S}}$ is:

$$\widetilde{\mathbb{S}}(a_1, \ldots, a_n) = (a_1 \sqcap_1 a_{1,2} \sqcap_1 \cdots \sqcap_1 a_{1,n}, \ldots, a_{n,1} \sqcap_n \cdots \sqcap_n a_{n,n-1} \sqcap_n a_n).$$

where $a_{i,j} = \mathbb{S}^i_j(a_i, a_j)$ for $1 \leq i \neq j \leq n$. As discussed in Section 4, this approach may not produce the most precise results, even if $\widetilde{\mathbb{S}}$ is repeated to a fixed point. It is worth pointing out that, even with this weaker approach, using a reference domain simplifies matters. Indeed, the reference domain allows us to derive all (optimal) $\mathbb{S}^j_i$ functions systematically from the appropriate $\widehat{\alpha}$ and $\widehat{\gamma}$:

$$\mathbb{S}^j_i(a_i, a_j) = \widehat{\alpha}_i(\widehat{\gamma}_i(a_i) \sqcap_{\mathcal{R}} \widehat{\gamma}_j(a_j))$$

6. $\mathcal{RL}$ as a reference domain for string analysis

We now instantiate the framework just presented. String analysis is one relevant setting for this, because of the large number of meaningful (but incomparable) string abstract domains in use. Each individual domain may capture an aspect of strings that is considered relevant for a given application, but the sheer number of domains involved can make a traditional approach to reduced products unworkable.[1]

Here a suitable reference domain could be the set of regular languages, $\mathcal{RL}$. In this section, that is exactly the reference domain we choose. We describe efficient algorithms for abstraction and concretisation relations between $\mathcal{RL}$ and the four elementary string abstract domains from Section 3. The definitions and algorithms amount to constructive proofs that $\mathcal{RL}$ can be a reference domain for

[1]For example, the collection $\mathcal{CS}$, $\mathcal{SL}$, $\mathcal{CI}$, $\mathcal{PS}$, $\mathcal{JS}$, $\mathcal{SF}$, $\mathcal{TJ}$ of seven string abstract domains are pairwise incomparable (the last three are the default domains of the JavaScript analysis tools JSAI, SAFE, and TAJS, respectively), but each captures a useful kind of string property [1].

those domains, as we provide the corresponding $\mathcal{RL}$-functions $\{\widehat{\alpha}_i, \widehat{\gamma}_i\}$ for $i \in \{\mathcal{CS}, \mathcal{SL}, \mathcal{CI}, \mathcal{PS}\}$. Elements of $\mathcal{RL}$ are assumed to be represented as trim deterministic finite-state automata. Note that a non-trim DFA R can be trimmed in linear time $O(|R|)$. In all cases where it is less than obvious how to implement a (sub-)operation, or the implementation can make essential use of the fact that input is trimmed, we provide detailed pseudo-code descriptions of algorithms.

6.1. Constant string

Algorithms 1 and 2 show how $\widehat{\gamma}_{\mathcal{CS}}$ and $\widehat{\alpha}_{\mathcal{CS}}$ can be implemented when sets of string are represented as trim automata.

Algorithm 1 Converting a $\mathcal{CS}$ element to a trim DFA

function CSTORL(a)
  Input: A $\mathcal{CS}$ element a
  Output: A trim DFA R with $\mathcal{L}(R) = \widehat{\gamma}_{\mathcal{CS}}(a)$
  if $a = \bot_{\mathcal{CS}}$ **then**
    return $(\{q_0\}, \Sigma, \emptyset, q_0, \emptyset)$
  if $a = \top_{\mathcal{CS}}$ **then**
    return $(\{q_0\}, \Sigma, \{(q_0, x, q_0) \mid x \in \Sigma\}, q_0, \{q_0\})$
  let $x_1 \cdots x_n = a$
  return $(\{q_0, \ldots, q_n\}, \Sigma, \{(q_{i-1}, x_i, q_i) \mid 0 < i \leq n\}, q_0, \{q_n\})$

Algorithm 2 Abstracting a trim DFA in $\mathcal{CS}$

function RLTOCS(R)
  Input: A trim DFA $R = (Q, \Sigma, \delta, q_0, F)$
  Output: $\widehat{\alpha}_{\mathcal{CS}}(R)$
  if $F = \emptyset$ **then**
    return $\bot_{\mathcal{CS}}$
  return RLTOCS(R, q_0, ϵ)

function RLTOCS(R, q, w)
  Input: A productive trim DFA $R = (Q, \Sigma, \delta, q_0, F)$,
    current state q, and word w accumulated so far
  Output: $\alpha_{\mathcal{CS}}(R)$
  $successors \leftarrow \{q' \mid (q, x, q') \in \delta\}$
  if $|successors| > 1$ **then**
    return $\top_{\mathcal{CS}}$
  if $q \in F$ **then**
    if $|successors| = 0$ **then**
      return w
    return $\top_{\mathcal{CS}}$
  let $\{(q, x, q')\} = successors$
  return RLTOCS($R, q', w \cdot x$)

6.2. String length

Algorithm 3 Converting an $\mathcal{SL}$ element to a trim DFA

function SLTORL(a)
  Input: An $\mathcal{SL}$ element a
  Output: A trim DFA R with $\mathcal{L}(R) = \widehat{\gamma}_{\mathcal{SL}}(a)$
  if $a = \bot_{\mathcal{SL}}$ **then**
    return $(\{q_0\}, \Sigma, \emptyset, q_0, \emptyset)$
  let $[l, u] = a$
  if $u \neq \infty$ **then**
    return $(\{q_0, \ldots, q_u\}, \Sigma, \{(q_{i-1}, x, q_i) \mid 0 < i \leq u\}, q_0, \{q_l, \ldots, q_u\})$
  let $\delta = \{(q_{i-1}, x, q_i) \mid 0 < i \leq l \wedge x \in \Sigma\} \cup \{(q_l, x, q_l) \mid x \in \Sigma\}$
  return $(\{q_0, \ldots, q_l\}, \delta, q_0, \{q_l\})$

Algorithm 4 Extracting length of longest string from a productive trim DFA

function EXTRACTMAXLENGTH(R)
  Input: A trim DFA $R = (Q, \Sigma, \delta, q_0, F)$ with $\mathcal{L}(R) \neq \emptyset$
  Output: Length of the longest string accepted by R
  for $q \in Q$ **do**
    $dist[q] \leftarrow \bot$
  return LONGESTDFS($R, q_0, dist$)

function LONGESTDFS($R, q, dist$)
  Input: A trim DFA $R = (Q, \Sigma, \delta, q_0, F)$,
    current state q, and cache $dist$ of computed results
  Output: Length of the longest string accepted by R starting from q
  if $dist[q] = (Closed, len)$ **then**
    return len
  if $dist[q] = Open$ **then**
    return ∞
  $dist[q] \leftarrow Open$
  $successors \leftarrow \{q' \mid (q, x, q') \in \delta\}$
  if $successors = \emptyset$ **then**
    $len \leftarrow 0$
  else
    $len \leftarrow 1 + \textbf{max}(\{\text{LONGESTDFS}(R, q', dist) \mid q' \in successors\})$
  $dist[q] \leftarrow (Closed, len)$
  return len

Algorithm 3 shows the implementation of $\widehat{\gamma}_{\mathcal{SL}}$. The abstraction $\widehat{\alpha}_{\mathcal{SL}}$ from a trim DFA finds the shortest path from the start state to any end state to create l and the longest path to create u. The upper bound l may be retrieved in $O(|R|)$ time by a breadth-first traversal. If there is a cycle in the (trim) automaton,

the longest path is $+\infty$; thus u may be retrieved by a depth-first traversal which runs in $O(|R|)$ time. Algorithm 4 shows in detail how to extract the longest string accepted by a given trim DFA. This is done by determining $dist(q)$ for each automaton state q. During processing, a value of $\bot$ indicates that q has not yet been considered. A value of $Open$ indicates that q has been visited, but it has not yet been determined whether q is part of some cycle of transitions—if q is ever re-visited, a length of ∞ will be deduced. Finally, a value of $(Closed, len)$ indicates that the length of the longest path starting from q has been determined; it is len. Note that the correctness of this algorithm rests on the fact that input DFAs are assumed to be trim.

6.3. Character inclusion

An element $[L, U]$ of the character inclusion domain denotes $U^* \cap \bigcap_{x \in L} U^* x U^*$. The automaton for $\widehat{\gamma}_{\mathcal{CI}}([L, U])$ is the $2^{|L|}$-state trim automaton $(\mathcal{P}(L), \Sigma, \delta, \emptyset, \{L\})$, where

$$\delta = \{(q, c, q \cup \{c\}) \mid q \in \mathcal{P}(L), c \in L\} \cup \{(q, c, q) \mid q \in \mathcal{P}(L), c \in U \setminus L\}.$$

The $\mathcal{RL}$-abstraction $\widehat{\alpha}_{\mathcal{CI}}$ is more efficient. The upper set U is the union of all characters appearing in transitions of the trim automaton. The lower set L can be computed in $O(|R||\Sigma|)$ time by, for each $x \in \Sigma$, performing a depth first traversal (similar to that in Algorithm 4) searching for an accepting run not containing x (see Algorithm 5). Figure 4 shows the automaton $\widehat{\gamma}_{\mathcal{CI}}([\{\mathtt{a}, \mathtt{b}\}, \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}])$, as an example.

Algorithm 5 Character extraction from a trim DFA

```
1:  function CHARACTEREXTRACT(R)
2:      Input: A trim DFA R = (Q, Σ, δ, q0, F)
3:      Output: Sets of characters that must/may occur in all strings accepted by R
4:      U ← {x | (q, x, q') ∈ δ}
5:      for q ∈ Q do
6:          c[q] ← Σ
7:      c[q0] ← ∅
8:      P ← {q0}
9:      while ∃q ∈ P do
10:         P ← P\{q}
11:         for (q, x, q') ∈ δ do
12:             S ← c[q'] ∩ (c[q] ∪ {x})
13:             if c[q'] ≠ S then
14:                 c[q'] ← S
15:                 P ← P ∪ {q'}
16:     L ← ⋂{c[q] | q ∈ F}
17:     return [L, U]
```

As computation of $\widehat{\gamma}_{\mathcal{CI}}$ is expensive and $\mathcal{CI}$ provides little strengthening to other domains, it may be preferable to use pairwise strengtheners for interactions with $\mathcal{CI}$ alone.

6.4. Prefix/suffix

An element $\langle p, s \rangle \in \mathcal{PS}$ represents the language $p\Sigma^* \cap \Sigma^* s$. The corresponding automaton could be constructed using the well-known product automaton construction, which however has complexity

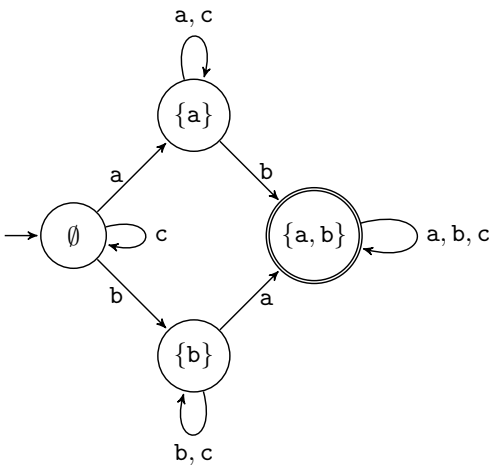


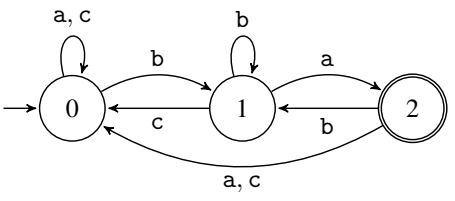
Figure 4. Automaton produced for $\widehat{\gamma}_{\mathcal{CI}}([\{\mathtt{a},\mathtt{b}\},\{\mathtt{a},\mathtt{b},\mathtt{c}\}])$

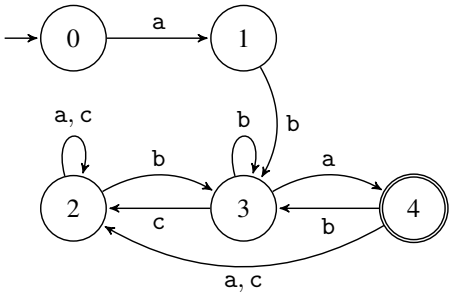
$O(|p| \cdot |s|)$. We can do better by directly constructing a minimal trim DFA with only $|p| + |s| + 1$ states. The construction follows these steps:

1. Construct the Knuth-Morris-Pratt automaton for $\Sigma^* s$, which has $|s| + 1$ states. The detailed algorithm for its construction is well-known [35]. Let the resulting DFA be $d = (Q, \Sigma, \delta, q_0, F)$.
2. If $p = \epsilon$, return d as the final result. Otherwise, run d on input p and note the state $q' \in Q$ which is reached as p is exhausted.
3. Generate $n = |p| > 0$ states $q_1, q_2, \ldots q_n$ where $p = x_1 x_2 \cdots x_n$.
4. Return the automaton $(Q \cup \{q_1, q_2, \ldots q_n\}, \Sigma, \delta', q_1, F)$ as the final result, where

$$\delta'(q,x) = \begin{cases} \delta(q,x) & \text{if } q \in Q \\ q_{i+1} & \text{if } q = q_i,\ i < n, \text{ and } x = x_i \\ q' & \text{if } q = q_n \text{ and } x = x_n \end{cases}$$

Example 6.1. Let the alphabet $\Sigma = \{\mathtt{a},\mathtt{b},\mathtt{c}\}$. Consider $\langle \mathtt{ab}, \mathtt{ba} \rangle \in \mathcal{PS}$. The Knuth-Morris-Pratt automaton for $\Sigma^*\mathtt{ba}$ is shown below (on the left). When this DFA processes $\mathtt{ab}$, it ends up in its middle state. Hence the (trim) automaton shown below (on the right) results.





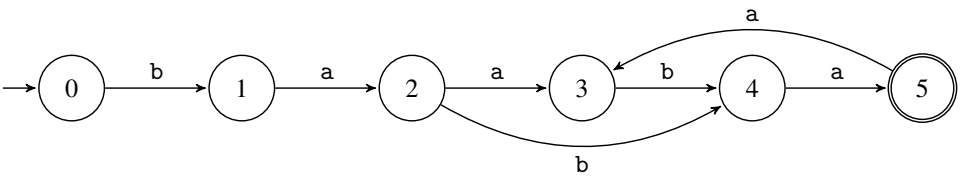
□

Given an automaton for $R \in \mathcal{RL}$, we can extract the longest prefix by following transitions from the start state so long as a unique successor exists, stopping if and when an accept state is reached, or multiple transitions apply (assuming the automaton is trim and deterministic). This may be done in $O(|Q|)$ time. The longest suffix is collected similarly, by traversing from all accept states backwards while all entering transitions to all states use the same symbol, stopping if we reach the start state. The process is $O(|\delta||Q|)$, as can observed from Algorithm 6.

Algorithm 6 Suffix extraction from a trim DFA

function SUFFIXEXTRACT(R)
  Input: A trim DFA $R = (Q, \Sigma, \delta, q_0, F)$
  Output: The longest suffix common to all strings accepted by R
  $suffix \leftarrow \epsilon$
  $layer \leftarrow F$
  while $q_0 \notin layer$ **do**
    $trans \leftarrow \{(q, x, q') \in \delta \mid q' \in layer\}$
    $syms \leftarrow \{x \mid (q, x, r) \in trans\}$
    if $|syms| = 1$ **then**
      let $\{x\} = syms$
    else
      return $suffix$
    $suffix \leftarrow x \cdot suffix$
    $layer \leftarrow \{q \mid (q, x, r) \in trans\}$
  return $suffix$

Example 6.2. Consider this automaton:



The forwards process yields longest prefix `ba`. The backwards process starts with accept state(s) $\{5\}$ obtaining `a`, and new set of states $\{4\}$, then obtaining `b` and states $\{2, 3\}$, then obtaining `a` and states $\{1, 2, 5\}$. At this point the process halts since there are arcs labelled `a` and `b` entering. The (longest) suffix collected in this process is `aba`. □

6.5. Meet and join

We have described how to translate various elementary string abstractions to and from trim DFAs. To complete the description of how to use this in an exact simulation of the reduced product, we need to explain how to calculate the meet operation $\sqcap_{\mathcal{RL}}$, that is, intersection of regular languages represented as trim DFAs. This, however, is well understood and involves the usual product automaton construction [36]. The only difference, when this idea is applied to trim automata, is that an extra implicit "failure" state must be accounted for, both in input and output; however, the algorithm is the

same. As in the non-trim case, the product automaton may not be minimal, so an implementor would be advised to apply a standard DFA minimisation algorithm, suitably adapted to trim DFAs.

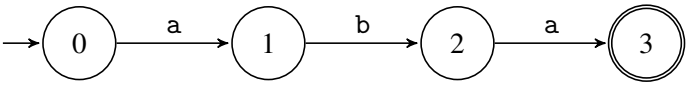
Moreover, the product automaton may not be trim. As noted, a DFA can be trimmed in linear time.

The join operation $\sqcup_{\mathcal{RL}}$, that is, union of regular languages represented as trim DFAs, is a simple variant of the meet, as in the case of standard DFAs. Although not strictly required for the reference domain to serve its purpose, the join can potentially be more precise than the reduced product's join.

Example 6.3. Having shown how $\mathcal{RL}$-abstractions $\widehat{\alpha}_i$ and $\mathcal{RL}$-concretisation $\widehat{\gamma}_i$ can be implemented for $i \in \{\mathcal{CS}, \mathcal{SL}, \mathcal{CI}, \mathcal{PS}\}$, we now show how the corresponding strengthening function $\mathbb{S}$ works. Consider the tuple:

$$a = (\top_{\mathcal{CS}}, [0,3], [\{\mathtt{a},\mathtt{b}\}, \{\mathtt{a},\mathtt{b},\mathtt{c}\}], \langle \mathtt{ab}, \mathtt{ba}\rangle) \in \mathcal{CS} \times \mathcal{SL} \times \mathcal{CI} \times \mathcal{PS}.$$

The $\mathcal{RL}$-concretisation for $\langle \mathtt{ab}, \mathtt{ba}\rangle$ and $[\{\mathtt{a},\mathtt{b}\}, \{\mathtt{a},\mathtt{b},\mathtt{c}\}]$ have already been shown previously, while $\gamma_{\mathcal{SL}}([0,3])$ and $\gamma_{\mathcal{CS}}(\top_{\mathcal{CS}})$ are trivial. The corresponding trim DFA $r = \widehat{\gamma}(a)$ is:



This actually corresponds to the concrete string aba, indeed $\gamma(s) = \{\mathtt{aba}\}$. This refined information is then sent back via $\widehat{\alpha}$ to the elementary domains: $\widehat{\alpha}_{\mathcal{CS}}(r) = \{\mathtt{aba}\}$, $\widehat{\alpha}_{\mathcal{SL}}(r) = [3,3]$, $\widehat{\alpha}_{\mathcal{CI}}([\{\mathtt{a},\mathtt{b}\}, \{\mathtt{a},\mathtt{b}\}])$, and $\widehat{\alpha}_{\mathcal{PS}}(r) = \langle \mathtt{aba}, \mathtt{aba}\rangle$. The resulting tuple $\widehat{\alpha}(\widehat{\gamma}(a))$ is much more precise than a, and in particular allows $\mathcal{CS}$ to greatly improve its precision from Σ^* to the singleton $\{\mathtt{aba}\}$. □

7. Widening

Where an abstract domain has infinite ascending chains (e.g., $\mathcal{SL}$), Kleene iteration is not guaranteed to reach a fixed point. Termination is recovered by defining a *widening* operator $\triangledown$ [13], such that for any sequence $\{a_1, a_2, \ldots\}$, the sequence $b_1 = a_1, b_{i+1} = b_i \triangledown a_{i+1}$ stabilizes after finitely many steps. For a direct product of domains, we may obtain a widening by combining the component widenings:

$$(x_1, \ldots, x_n) \triangledown (y_1, \ldots, y_n) = (x_1 \triangledown_1 y_1, \ldots, x_n \triangledown_n y_n)$$

One could attempt to do the same for reduced products, choosing some (ideally minimal) representative from the equivalence class of each operand, then apply the individual widenings pointwise. Unfortunately, this does not in general define a valid widening. This may be side-stepped by separating the results of widening, which are maintained non-reduced, from other computations in the analysis [7].

It can be challenging to define widening operators for expressive domains which balance convergence rate with preserving precision. A typical widening $R_1 \triangledown_{\mathcal{RL}} R_2$ for $\mathcal{RL}$ [2] defines an equivalence relation between two states $s_1 \sim s_2$ in the automata for R_1 and R_2 if there exists a string w where either (a) w reaches s_1 from the start state of R_1 and s_2 from the start state of R_2, or w reaches an accept state in R_1 from s_1 and an accept state in R_2 from s_2. This defines an equivalence relation $\equiv$ on states s, s' is R_1 as $s \equiv s'$ iff exists $s_2 \in R_2, s \sim s_2 \wedge s' \sim s_2$. The result of widening is the quotient $R_1/_{\equiv}$.

Example 7.1. Consider the pseudo-code

```
x = "aaa"
while (*)
   if (length(x) < 4) x = "a" + x
```

After the first iteration around the loop, the variable x gets the value `aaaa`. Assuming we widen at this point, then widening the automaton R_2 for `aaaa` with automaton R_1 for `aaa` generates the automaton for $\mathtt{a}^*$, since each pair of adjacent states in R_1 are made equivalent. □

Such aggressive widening means we can adopt a strictly more expressive domain, yet infer weaker invariants.

Example 7.2. Recall the analysis from Example 7.1, but this time under $\mathcal{PS}$. $\mathcal{PS}$ needs no separate widening operator, as it contains no infinite ascending chains. The initial abstraction is $\langle \mathtt{aaa}, \mathtt{aaa} \rangle$ and after the if statement we obtain the join with $\langle \mathtt{aaaa}, \mathtt{aaa} \rangle$, leading to the same description. The $\mathcal{PS}$ description maintains the information that there are at least 3 `a`s in the string x. □

This reveals an interesting possibility: having established $\mathcal{R}$ as a reference domain for $\mathcal{A}$, we can use $\mathcal{A}$ to strengthen the widening on $\mathcal{R}$. Let us define an operator $\triangledown_{\mathcal{R}|\mathcal{A}}$:

$$(R_1, A_1) \triangledown_{\mathcal{R}|\mathcal{A}} R_2 = (R_1 \triangledown_{\mathcal{R}} R_2, A_1 \triangledown_{\mathcal{A}} \widehat{\alpha}(R_2))$$

Essentially, we compute the widening under the reduced product of $\mathcal{R}$ and $\mathcal{A}$. We may then replace R_i with $R_i \sqcap_{\mathcal{R}} \widehat{\gamma}(A_i)$ in subsequent computations.

8. Related work

It appears that all previous approaches to combining several abstractions have been based on the idea of information exchange between the components, by means of dedicated exchange functions. We have referred to this as a *paraphrasing* approach and we illustrated it briefly in Section 4, in the setting of string analysis. A distinguishing characteristic of one-on-one paraphrasing is that the information exchange is incomplete, in the sense that there is no guarantee that it produces the same precision as the ideal, that is, the reduced product. Any resulting notion of "product" is therefore, intuitively, an abstraction of the reduced product; it is "further away" from the concrete domain, relative to the reduced product. In contrast, a reference abstract domain is chosen so that the reduced product is an abstraction of *it*. It is "closer" to the concrete domain.

The paraphrasing approach has many advantages, but in this paper we have argued against it where many (say, $n > 3$) incomparable abstract domains are involved. We noted that, for analysis of string-manipulating programs it is not uncommon to employ half a dozen relevant but incomparable string abstract domains. In such cases the number of required functions dedicated to information exchange becomes prohibitive. Even when the functions are restricted to providing one-on-one paraphrasing (as is almost always the case), the number is quadratic in the number n of abstract domains to be combined.

Since paraphrasing approaches are both common and important, let us briefly trace their history. Iterated paraphrasing was proposed by Granger [22], under the name of "local decreasing iterations". It is used as a vehicle for various improvement techniques in abstract interpretation. For example,

the analysis of a complex predicate (from a conditional) can be improved by repeated application of transfer functions for the components of the test. Granger also shows how the technique can be applied to constraints involving more than two variables, after appropriate isolation of variables.

Of relevance to the present paper, Granger [22] also discusses[2] a connection to abstract domain combinations, or rather to refining elements of direct products. Given components $\mathcal{A}_1$ and $\mathcal{A}_2$ of a Cartesian product, define operations $\rho_1 : (\mathcal{A}_1 \times \mathcal{A}_2) \to \mathcal{A}_1$ and $\rho_2 : (\mathcal{A}_1 \times \mathcal{A}_2) \to \mathcal{A}_2$ which tighten the two components respectively. That is, $\rho_1(a_1, a_2) \sqsubseteq a_1$ and $\gamma(\rho_1(a_1, a_2), a_2) = \gamma(a_1, a_2)$, and similarly for ρ_2. A decreasing chain $(a_1^n, a_2^n)_{n\in\mathbb{N}}$ is obtained as follows:

$$\begin{array}{lcl} (a_1^0, a_2^0) & = & (a_1, a_2) \\ (a_1^{i+1}, a_2^{i+1}) & = & (\rho_1(a_1^i, a_2^i), \rho_2(a_1^i, a_2^i)) \end{array}$$

For example, to combine the classical abstract domain $Intv$ of intervals with that of affine congruences $AffCong$, Granger considers, in lieu of the latter, the non-relational and *strictly less precise* domain $ArithCong$ of arithmetic congruences (the domain $Intv$ is incomparable with either of the two). Information $(d, e) \in Intv \times AffCong$ is first *weakened* to the closest consistent $(d, e') \in Intv \times ArithCong$; this is a matter of calculating projections. The reason for this weakening is that it enables a straightforward paraphrasing—so that d is *strengthened* to some $d' \in Intv$. The stronger d' may in turn allow strengthening of the original e, via paraphrasing; hence the overall result is a strengthening, although not necessarily as strong as what the reduced product would allow. Granger's techniques thus correspond to products that are parameterised on paraphrasing functions and strategies, unlike the reference domains proposed in this paper. Relative to the reduced product, Granger's products are further removed from the concrete domain, whereas a reference domain is *closer* to the concrete domain.

Granger's observations about the utility of repeating or iterating the application of a transfer function were also made (independently) in the context of logic program analysis [5, 28]. Le Charlier and Van Hentenryck [28] refer to this as "reexecution" and define a "REFINE" operation that ensures reexecution is in fact a strengthening, a requirement which was neglected elsewhere [31].

Cortesi, Le Charlier and Van Hentenryck [9] proposed another notion of product, namely the *open product*. In this framework, abstract operations are made higher-order, parameterised on "queries". The idea is that an abstract operation can interrogate the various components of a Cartesian product, using a fixed set of predicates. An abstract operation $O : (\mathcal{A}_1 \times \mathcal{A}_2) \to (\mathcal{A}_1 \times \mathcal{A}_2)$ which makes use of m queries $Q_1^1, \ldots, Q_m^1$ on $\mathcal{A}_1$ and (say also) m queries $Q_1^2, \ldots, Q_m^2$ on $\mathcal{A}_2$ is defined in terms of the higher-order O_1 and O_2 as follows:

$$O(a_1, a_2) = \left(\begin{array}{c} O_1(\langle Q_1^1(a_1) \vee Q_1^2(a_2), \ldots, Q_m^1(a_1) \vee Q_m^2(a_2)\rangle)(a_1), \\ O_2(\langle Q_1^1(a_1) \vee Q_1^2(a_2), \ldots, Q_m^1(a_1) \vee Q_m^2(a_2)\rangle)(a_2) \end{array} \right)$$

Again, relative to the reduced product, the open product is further removed from the concrete domain, whereas a reference domain is *closer* to the concrete domain.

Techniques that are reminiscent of the open product idea are also found in the ASTRÉE approach to combinations of abstractions [16]. ASTRÉE employs a one-way strengthening approach (components can benefit from ones computed earlier, including on previous iterations), based on "communication channel" domains that are further abstractions of a number of component domains. Components

[2] Granger's presentation is by example. We follow the presentation given by Cortesi *et al.* [9].

can then volunteer information to others that share the same communication channel, and can query the information channel for previously computed information. This means that abstract domains are organised in some hierarchical way, based on pragmatic considerations of "who can best inform who", and the resulting notion of product is not commutative. ASTRÉE does not insist on Galois connections, nor that abstract domains are lattices, so a reference domain approach would not necessarily apply in the context of the abstract domains used by ASTRÉE.

A final notion of product is defined by Gulwani and Tiwari [23], namely the "logical product". Gulwani and Tiwari characterize an abstract state as a conjunction of atomic facts in some logical theory, then use a Nelson-Oppen-style technique to derive a new lattice over the combination of two such "logical lattices". This is in general incomparable with the reduced product, though at least as precise when the underlying theories are convex, stably infinite and disjoint.

To avoid a possible confusion we should point out also that Cortesi, Filé and Winsborough [8] provide a framework for comparing abstract domains for their ability to express certain kinds of information. The "kind of information" is itself given as an abstract domain, which is called a "reference domain". However, that domain plays a rather different role to the concept of reference domain that we have proposed. Again, for Cortesi *et al.* [8], the reference domain is usually more abstract than the domains D and D' that are being compared, as it is intended to capture limited aspects of D and D'.

A seminal paper on practical string analysis was the description of the Java String Analyzer, JSA, by Christensen, Møller and Schwartzbach [6]. JSA uses regular languages as approximations, and the JSA tool has found wide application. Later work on tools such as TAJS [24], JSAI [26] and SAFE [29] has drawn inspiration from JSA.

Apart from the regular language domain, the domains we use are domains (or are similar to domains) discussed by Madsen and Andreasen [30]. It is difficult to make an exact comparison because Madsen and Andreasen do not formally define their domains. Thus, for example, it is not clear what the semantics of elements of their prefix-suffix domain would be. Their definition of the character inclusion domain is also rather different from this paper's. Costantini et al. [10, 12] discuss a prefix domain, and a suffix domain, the reduced product of which yields $\mathcal{PS}$. They also discuss $\mathcal{CI}$ and propose two additional (more complex) string domains.

Amadini *et al.* [1] describe the implementation, in the SAFE framework, of a dozen string abstract domains. The implementation enables easy combination of domains, and this allows a systematic exploration of combinations, using a fixed suite of benchmarks. However, "combination" here means direct product—there is no attempt to let the individual domains interact.

9. Conclusion

We have presented a modular framework for implementation of the reduced product of a number of component abstract domains. The key concept in this framework is a *reference domain* whose role is to mediate and translate information amongst the component domains. The reference domain needs to be chosen so that it is at least as expressive as each of the component domains.

Normally, in abstract interpretation, the use of a highly expressive abstract domain has both advantages and drawbacks. On the negative side, a high runtime cost is incurred, as basic abstract operations can be expected to be relatively expensive. More importantly, the expected precision gains often fail to materialise because (coarse) widening operations must be employed for the analysis to reach a fixed point, and this can incur considerable loss of precision. Hence, combinations of simple abstract domains are often preferred.

Traditionally, a reduced product is computed by writing separate *ad hoc* "paraphrasing" functions for each pair of domains, to facilitate one-on-one information exchange. For improved precision, paraphrasing can be iterated until a fixed point is reached. When the number of abstract domains involved is large, this paraphrasing approach becomes unmanageable. Moreover, we have shown that this process of iterated paraphrasing, in spite of the massive labour it may involve, in general yields results that are weaker than the ideal—the reduced product. Our framework replaces the $n(n-1)$ paraphrasers by $2n$ conversion functions, to and from a suitably chosen reference domain. We have shown that this approach is equivalent to using the reduced product.

In this paper we have applied the framework to the problem of string analysis, using the set of regular languages as the reference domain. We have presented algorithms for converting between the regular language domain and four commonly used string domains. This approach can be used for both computing and testing the reduced product of string domains. Our work forms a basis for our ongoing implementation of string abstract domains in the SAFE analysis framework [1, 29]. Because it is common for string analyses to combine many separate string domains that capture overlapping aspects of strings (and SAFE is no exception), our framework is attractive.

The main message to take from this paper is that a highly expressive abstract domain may well be too expensive to use in abstract interpretation, in the traditional manner. However, if used as a *lingua franca*, it can facilitate the exchange of information and enable the cooperation between simpler abstract domains. That is the role of a reference domain. There is no recipe that we know of, for how to best choose a reference domain. As with other abstract domains, somebody needs to make the design choice, based on the requirements of an application. Natural choices, however, do not appear difficult to come by, as we hope we have demonstrated by example.

Acknowledgements

We thank the anonymous reviewers for helpful comments on our initial manuscript. The work has been supported by the Australian Research Council through Linkage Project Grant LP140100437 and Discovery Early Career Researcher Award DE160100568.

References

[1] Amadini R, Jordan A, Gange G, Gauthier F, Schachte P, Søndergaard H, Stuckey PJ, and Zhang Ch. Combining string abstract domains for JavaScript analysis: An evaluation. In A. Legay and T. Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 10205 of *Lecture Notes in Computer Science*, pages 41–57. Springer, 2017. URL `https://doi.org/10.1007/978-3-662-54577-5_3`.

[2] Bartzis C, and Bultan T. Widening arithmetic automata. In R. Alur and D. A. Peled, editors, *Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 321–333. Springer, 2004. URL `https://doi.org/10.1007/978-3-540-27813-9_25`.

[3] Bisht P, Hinrichs TL, Skrupsky N, and Venkatakrishnan VN. WAPTEC: Whitebox analysis of web applications for parameter tampering exploit construction. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pages 575–586. ACM Publ., 2011.

[4] Bjørner N, Tillmann N, and Voronkov A. Path feasibility analysis for string-manipulating programs. In S. Kowalewski and A. Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 5505 of *Lecture Notes in Computer Science*, pages 307–321. Springer, 2009. URL https://doi.org/10.1007/978-3-642-00768-2_27.

[5] Bruynooghe M. A practical framework for the abstract interpretation of logic programs. *Journal of Logic Programming*, 1991;10(2):91–124. doi:10.1016/0743-1066(91)80001-T.

[6] Christensen AS, Møller A, and Schwartzbach MI. Precise analysis of string expressions. In R. Cousot, editor, *Static Analysis*, volume 2694 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2003. ISBN:3-540-40325-6.

[7] Cortesi A, Costantini G, and Ferrara P. A survey on product operators in abstract interpretation. In *Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his Sixtieth Birthday*, pages 325–336, 2013. doi:10.4204/EPTCS.129.19.

[8] Cortesi A, Filé G, and Winsborough W. Comparison of abstract interpretations. In W. Kuich, editor, *Automata, Languages and Programming: Proceedings of the 19th International Colloquium (ICALP'92)*, volume 623 of *Lecture Notes in Computer Science*, pages 521–532. Springer, 1992. URL https://doi.org/10.1007/3-540-55719-9_101.

[9] Cortesi A, Le Charlier B, and Van Hentenryck P. Combinations of abstract domains for logic programming: Open product and generic pattern construction. *Science of Computer Programming*, 2000;38(1-3):27–71. URL https://doi.org/10.1016/S0167-6423(99)00045-3.

[10] Costantini G. Lexical and numerical domains for abstract interpretation, PhD Thesis, Ca' Foscari University of Venice, 2014.

[11] Costantini G, Ferrara P, and Cortesi A. Static analysis of string values. In S. Qin and Z. Qiu, editors, *Formal Methods and Software Engineering*, volume 6991 of *Lecture Notes in Computer Science*, pages 505–521. Springer, 2011. URL https://doi.org/10.1007/978-3-642-24559-6_34.

[12] Costantini G, Ferrara P, and Cortesi A. A suite of abstract domains for static analysis of string values. *Software Practice and Experience*, 2015;45(2):245–287. doi:10.1002/spe.2218.

[13] Cousot P, and Cousot R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM Publ., 1977. doi:10.1145/512950.512973.

[14] Cousot P, and Cousot R. Systematic design of program analysis frameworks. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 269–282. ACM Publ., 1979. doi:10.1145/567752.567778.

[15] Cousot P, and Cousot R. Abstract interpretation frameworks. *Journal of Logic and Computation*, 1992;2(4):511–547. doi:10.1093/logcom/2.4.511.

[16] Cousot P, Cousot R, Feret J, Mauborgne L, Miné A, Monniaux D, and Rival X. Combination of abstractions in the ASTRÉE static analyzer. In M. Okada and I. Satoh, editors, *Advances in Computer Science – ASIAN 2006. Secure Software and Related Issues*, volume 4435 of *Lecture Notes in Computer Science*, pages 272–300. Springer, 2006. URL https://doi.org/10.1007/978-3-540-77505-8_23.

[17] Patrick Cousot, Radhia Cousot, and Laurent Mauborgne. A framework for combining algebraic and logical abstract interpretations. September 2010. URL https://hal.inria.fr/inria-00543890.

[18] Cousot P, and Halbwachs N. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th Annual ACM Symposium on Principles of Programming Languages*, pages 84–96. ACM Publ., 1978. doi:10.1145/512760.512770.

[19] Emmi M, Majumdar R, and Sen K. Dynamic test input generation for database applications. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 151–162. ACM Publ., 2007. ISBN:978-1-59593-734-6. doi:10.1145/1273463.1273484.

[20] Gange G, Navas JA, Schachte P, Søndergaard H, and Stuckey PJ. Abstract interpretation over non-lattice abstract domains. In F. Logozzo and M. Fähndrich, editors, *Static Analysis*, volume 7935 of *Lecture Notes in Computer Science*, pages 6–24. Springer, 2013. URL https://doi.org/10.1007/978-3-642-38856-9_3.

[21] Gange G, Navas JA, Stuckey PJ, Søndergaard H, and Schachte P. Unbounded model-checking with interpolation for regular language constraints. In N. Piterman and S. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 7795 of *Lecture Notes in Computer Science*, pages 277–291. Springer, 2013. URL https://doi.org/10.1007/978-3-642-36742-7_20.

[22] Granger P. Improving the results of static analyses of programs by local decreasing iterations. In R. Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 652 of *Lecture Notes in Computer Science*, pages 68–79. Springer, 1992. URL https://doi.org/10.1007/3-540-56287-7_95.

[23] Gulwani S, and Tiwari A. Combining abstract interpreters. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 376–386. ACM Publ., 2006. doi:10.1145/1133255.1134026.

[24] Jensen SH, Møller A, and Thiemann P. Type analysis for JavaScript. In J. Palsberg and Z. Su, editors, *Static Analysis*, volume 5673 of *Lecture Notes in Computer Science*, pages 238–255. Springer, 2009. ISBN: 978-3-642-03236-3. doi:10.1007/978-3-642-03237-0_17.

[25] Karr M. Affine relationships among variables of a program. *Acta Informatica*, 1976;6:133–151. URL https://doi.org/10.1007/BF00268497.

[26] Kashyap V, Dewey K, Kuefner EA, Wagner J, Gibbons K, Sarracino J, Wiedermann B, and Hardekopf B. JSAI: A static analysis platform for JavaScript. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 121–132. ACM Publ., 2014. ISBN: 978-1-4503-3056-5. doi:10.1145/2635868.2635904.

[27] Kim S-W, Chin W, Park J, Kim J, and Ryu S. Inferring grammatical summaries of string values. In J. Garrigue, editor, *Programming Languages and Systems: Proceedings of the 12th Asian Symposium (APLAS'14)*, volume 8858 of *Lecture Notes in Computer Science*, pages 372–391. Springer, 2014. URL `https://doi.org/10.1007/978-3-319-12736-1_20`.

[28] Le Charlier B, and Van Hentenryck P. Reexecution in abstract interpretation of Prolog. In K. Apt, editor, *Logic Programming: Proceedings of the Joint International Conference and Symposium*, pages 750–764. MIT Press, 1992.

[29] Lee H, Won S, Jin J, Cho J, and Ryu S. SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript. In *Proceedings of the 19th International Workshop on Foundations of Object-Oriented Languages (FOOL'12)*, 2012.

[30] Madsen M, and Andreasen E. String analysis for dynamic field access. In A. Cohen, editor, *Compiler Construction*, volume 8409 of *Lecture Notes in Computer Science*, pages 197–217. Springer, 2014. URL `https://doi.org/10.1007/978-3-642-54807-9_12`.

[31] Marriott K, and Søndergaard H. On propagation-based analysis of logic programs. In S. Michaylov and W. Winsborough, editors, *Proceedings of the ILPS 93 Workshop on Global Compilation*, pages 47–65, 1993.

[32] Minamide Y. Static approximation of dynamically generated web pages. In *Proceedings of the 14th International Conference on World Wide Web*, pages 432–441. ACM Publ., 2005. ISBN:1-59593-046-9. doi:10.1145/1060745.1060809.

[33] Park Ch, Im H, and Ryu S. Precise and scalable static analysis of jQuery using a regular expression domain. In *Proceedings of the 12th Symposium on Dynamic Languages*, pages 25–36. ACM Publ., 2016. ISBN: 978-1-4503-4445-6. doi:10.1145/2989225.2989228.

[34] Polya G. *How to Solve It*. Doubleday Anchor Books, second edition, 1957.

[35] Sedgewick R, and Wayne K. *Algorithms*. Addison-Wesley, fourth edition, 2011.

[36] Sipser M. *Introduction to the Theory of Computation*. Thomson Course Technology, third edition, 2012. ISBN-10:113318779X, 13:978-1133187790.

[37] Veanes M, de Halleux P, and Tillmann N. Rex: Symbolic regular expression explorer. In *Proceedings of the Third International Conference on Software Testing, Verification and Validation*, pages 498–507. IEEE Comp. Soc., 2010. doi:10.1109/ICST.2010.15.