Volume 4, Number 2
April 2017

Published by the Association for Computing Machinery
Special Interest Group on Logic and Computation

SIG**LOG** *news*

TABLE OF CONTENTS

General Information

Technical Columns

Regular Features

Association for
Computing Machinery

Advancing Computing as a Science & Profession

SIGLOG NEWS

Published by the ACM Special Interest Group on Logic and Computation

SIGLOG News (ISSN 2372-3491) is an electronic quarterly publication by the Association for Computing Machinery.

From the Editor

In this issue

– Mikołaj Bojańczyk's Automata column presents a category-theoretic perspective on automata and minimization in an article by Thomas Colcombet and Daniela Petrişan.

– Mai Gehrke and Andreas Krebs introduce us to Stone duality in Neil Immerman's column on Complexity.

– Neha Rungta's Verification column features two articles:
  - Stephen Siegel discusses CIVL Solutions to the VerifyThis 2016 Challenges, and
  - Chao Wang and Patrick Schaumont write about formal methods for detecting and eliminating side-channel leaks.

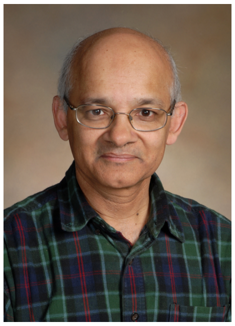
– Bernardo Toninho reports on this year's POPL in the Conference Reports section, edited by Jorge A. Pérez.

– As usual, we wrap up with the latest issue of SIGLOG Monthly, prepared by Daniela Petrişan.

SIGLOG News is still looking for a volunteer to coordinate a section on book reviews. Please email `editor@siglog.org` if you are interested.

Andrzej Murawski
University of Warwick
SIGLOG News Editor

Chair's Letter



The summer conference and travel season is almost upon us. This year the activity will be focussed on Europe. We will have LICS in Iceland, MFPS in Slovenia and ICALP in Poland. Other smaller workshops and meetings will also happen, notably QPL in the Netherlands. Later in the summer CSL will be in Sweden and CONCUR will be held together with FORMATS, QEST and EPEW in Germany. Next year, of course, we will have the mammoth FLoC meeting in Oxford where the whole community gets together. It is always great to see each other face-to-face and renew ties and exchange ideas.

SIGLOG membership seems to be in a slight decline and I am wondering why. With deep discounts at many conferences and extremely low annual dues it seems to me that SIGLOG membership is a bargain. Also the sterling work of our Newsletter Editor and the column editors gives us a newsletter than rivals that of any other SIG. This issue is over a 100 pages and is filled with technical content and community news. If you have let your membership lapse please do remember to renew. Looking forward to seeing many of you this summer.

Prakash Panangaden
McGill University
ACM SIGLOG Chair

AUTOMATA COLUMN

MIKOŁAJ BOJAŃCZYK, University of Warsaw
bojan@mimuw.edu.pl

This column is about minimising automata, which is one of the fundamental themes in automata theory. The classical result is that deterministic finite automata on finite words can be minimised. However minimisation (or more importantly, the existence of a unique canonical device) is also true in a very wide variety of other settings, including different types of inputs (words, infinite words, trees, etc.) as well as different types of outputs (yes/no values, numbers, or other words, etc.). In this column, Thomas Colcombet and Daniela Petrişan have a look at minimisation from an abstract (categorical) perspective, and establish what are the conditions needed for a model of automata to admit minimisation.

Automata and minimization[1]

Thomas Colcombet
IRIF, CNRS

Daniela Petrişan
IRIF, CNRS

Already in the seventies, strong results illustrating the intimate relationship between category theory and automata theory have been described and are still investigated. In this column, we provide a uniform presentation of the basic concepts that underlie minimization results in automata theory. We then use this knowledge for introducing a new model of automata that is an hybrid of deterministic finite automata and automata weighted over a field. These automata are very natural, and enjoy minimization result by design.

The presentation of this paper is indeed categorical in essence, but it assumes no prior knowledge from the reader. It is also non-conventional in that it is neither algebraic, nor co-algebraic oriented.

1. INTRODUCTION

In this column, we attempt to give a simple, user-friendly, description of how category theory sheds an interesting light on some aspects of automata theory and in particular concerning the existence of minimal recognizers.

Seen from distance, an automaton is a machine that

processes an input, respecting its *structure* (word, tree, infinite word or tree, trace, . . .)	and	outputs a quantity in some *universe of output values* (Boolean values, probabilities, vector space, words, . . .)

These two aspects, structure of the input and universe of outputs play an essentially orthogonal role, and provide a good organization scheme in the description of the landscape of automata. Though sometimes interacting, these two axes deserve to be understood in an independent way. An important unification step for understanding the structure of the input axis has been described by Bojańczyk [Bojańczyk 2015] thanks to the use of 'monads' from category theory. Our focus is on the universe of output values axis.

In this paper, we focus our attention to the universe of outputs, and the related notion of state space. Here, once more, category theory offers a neat understanding of phenomena. We assume no knowledge of category theory from the reader.

THEOREM 1.1. *Given an automaton $\mathcal{A}$ in a class C, there exists another automaton $\mathcal{M}$ in C which is algebraically minimal*[2] *while having the same semantics.*

[1] This work was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No.670624), and by the DeLTA ANR project (ANR-16-CE40-0007). The authors also thank the Simon's Institute for the Theory of Computing where this work has been partly developed.

[2] It is a common approximation to say that an automaton is minimal if its number of states is minimal. We emphasize that this is not the notion we consider by using the terminology algebraically minimal.

This theorem is very well known for deterministic (and complete) automata as presented in the seminal work of Rabin and Scott [Rabin and Scott 1959]. It also known from Schützenberger's work on automata weighted over a field [Schützenberger 1961] (that we will present in more detail). Other similar results involve automata over trees [Brainerd 1968], deterministic transducers [Choffrut 1979; Choffrut 2003], syntactic monoids [Schützenberger 1965], syntactic ω/$\diamond$/$\circ$-semigroups [Perrin and Pin 1995; Bedon 1996; Bedon et al. 2010; Carton et al. 2011; Colcombet and A. V. 2015], stabilisation monoids [Colcombet 2009; Kuperberg 2011; Colcombet 2013], syntactic semirings for languages [Polák 2001], syntactic forest algebras [Bojańczyk and Walukiewicz 2008], syntactic nominal monoids [Bojańczyk 2011], and so on. However, it also fails for many other classes, starting with non-deterministic automata or deterministic automata over infinite inputs.

A goal of this paper is to give a neat description of what this approach means at an abstract level, and why it sometimes works, and sometimes not. This will be also the occasion to describe new forms of automata, namely hybrid-set-vector automata that were not known in the literature, that extend both deterministic finite automata and automata weighted over a field, while preserving the property of admitting algebraically minimal automata (the computation of which being still open). This is an example of the process of using a category-theoretic approach for defining new devices (here automata) that enjoy strong properties by design.

Structure of the column

In Section 2, we present the very natural notion of an automaton in a category, the examples of deterministic automata and vector space automata, as well as the minimalistic concepts of category theory that are required for this definition to make sense. In Section 3 we explain the concepts that are behind results of minimization, and in particular the one of factorization. In Section 4 we present the hybrid-set-vector automata. In Section 5 we discuss the connection between automata and category theory and some of the literature on this topic.

2. AUTOMATA IN A CATEGORY AND THEIR SEMANTICS

In this section, after studying classical examples, we introduce the notion of an automaton in a category. This presentation does not contain any new material, but departs from the literature in that it doesn't adopt the algebraic or the coalgebraic presentation of these objects. We hope that the resulting presentation is simpler, requires less background, and more faithfully follows the spirit of standard automata theory.

Before pursuing this description, we need to understand what is the semantics of an automaton. Let us start with these two examples:

DETERMINISTIC AUTOMATA

A *deterministic automaton* (finite or infinite), or simply a `Set`-*automaton* is a tuple

$$\mathcal{A} = \langle Q, A, i, f, \delta \rangle$$

in which Q is a *set of states*, A is the *input alphabet*, $i\colon 1 \to Q$ is the *initial map* (where 1 is some one element set, let us say $\{0\}$), $f\colon Q \to 2$ is the *final map* (where 2 is some two element set, let us say $\{0,1\}$), and $\delta_a\colon Q \to Q$ is the *transition map for the letter* a for all $a \in A$.

VECTOR SPACE AUTOMATA

Let us consider vector spaces over a base field $\mathbb{K}$. A *vector space automaton*, or simply a `Vec`-*automaton*, is a tuple

$$\mathcal{A} = \langle Q, A, i, f, \delta \rangle$$

in which Q is a *vector space of configurations* , A is the *input alphabet*, $i\colon \mathbb{K} \to Q$ is the *initial map*, $f\colon Q \to \mathbb{K}$ is the *final map*, and $\delta_a\colon Q \to Q$ is the linear *transition map for letter* $a \in A$.

Given a word $u = a_1 \dots a_n \in A^*$, the `Set`-automaton $\mathcal{A}$ *accepts* the map

$$[\![\mathcal{A}]\!](u) = f \circ \delta_{a_n} \circ \dots \circ \delta_{a_1} \circ i \,.$$

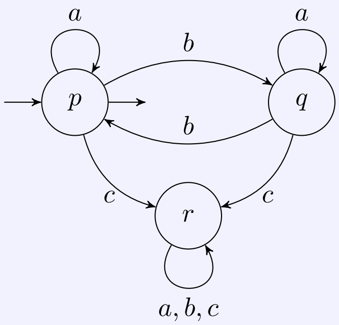
Since a map from 1 to 2 can take only two possible values: the constant 0 and the constant 1, $[\![\mathcal{A}]\!]$ can be understood as associating to each input word u either 1 (and we say that *the word is accepted*), or 0 (and we say that *the word is rejected*). Hence, it *computes a language*. We recognize here, in a barely disguised wording, the standard definition of a DFA.

Example 2.1. Consider the language L_{set} over the finite alphabet $A = \{a, b, c\}$ defined by:

$$L_{\text{set}} = \{u \in A^* \mid |u|_b \text{ is even and } |u|_c = 0\} \,.$$

An automaton $\mathcal{A}_{\text{set}}$ for this L is as follows:

— the set of states is $Q = \{p, q, r\}$,
— the initial map selects $p \in Q$,
— the final map maps p to 1, and q, r to 0.
— the transition maps are described below in the standard way



Given a word $u = a_1 \dots a_n \in A^*$, the `Vec`-automaton $\mathcal{A}$ *accepts* the linear map

$$[\![\mathcal{A}]\!](u) = f \circ \delta_{a_n} \circ \dots \circ \delta_{a_1} \circ i \,.$$

Since a linear map from $\mathbb{K}$ to $\mathbb{K}$ is of the form $x \mapsto ax$ for some $a \in \mathbb{K}$, $[\![\mathcal{A}]\!]$ can be understood as associating to each input word u a scalar $a \in \mathbb{K}$. This is a variation around the idea of an *automaton weighted over a field* of Schützenberger [Schützenberger 1961][3].

Example 2.1. Consider the following map L_{vec} which to a word u associates the linear map $L_{\text{vec}}(u) \colon \mathbb{R} \to \mathbb{R}$ defined by:

$$L_{\text{vec}}(u)(x) = \begin{cases} 2^{|u|_a} x & \text{if } |u|_b \text{ is even} \\ & \text{ and } |u|_c = 0, \\ 0 & \text{otherwise} \end{cases}$$

An automaton $\mathcal{A}_{\text{vec}}$ for this L is as follows:

— the vector space of configurations is $\mathbb{R}^2$,
— the initial map maps x to $(x, 0)$,
— the final map maps (x, y) to x,
— the transition map for a maps (x, y) to $(2x, 2y)$,
— the transition map for b maps (x, y) to (y, x),
— the transition map for c maps (x, y) to $(0, 0)$.

It is easy to check that this vector space automaton x accept L_{vec}.

Inspecting the two above definitions of automata, it is obvious that these can be unified. Category theory is certainly the proper language for such a unification. Let us give as a starter some very elementary definitions concerning categories.

WHAT IS A CATEGORY?

A *category* has essentially two parts:

$$\text{(objects, arrows)}\,,$$

[3] The two differences being that in our case, (a) the vector spaces can be of infinite dimension, and (b) there is no need for choosing a basis for Q, as it is done in the original definition.

where the *objects* are denoted $X, Y, \dots$, and each *arrow*, denoted $f \colon X \to Y$, goes from a *source object* X to a *target object* Y.

Typical categories are:

$\texttt{Set} = (\text{sets}, \text{functions between sets})$,
$\texttt{Pos} = (\text{ordered sets}, \text{order preserving maps})$,
$\texttt{Rel} = (\text{sets}, \text{relations between sets})$,
$\texttt{Top} = (\text{topological spaces}, \text{continuous maps})$,
$\texttt{Vec} = (\text{vector spaces}, \text{linear maps})$,
$\texttt{Aff} = (\text{affine spaces}, \text{affine maps})$,
$\texttt{Grp} = (\text{groups}, \text{group morphisms})$,
$\dots$

Furthermore, properly defined categories have to contain the following extra pieces of information:

(a) for all objects X, there is an arrow $\mathrm{Id}_X \colon X \to X$ called the *identity of* X, and
(b) given arrows $f \colon X \to Y$, $g \colon Y \to Z$, there exists a *composite arrow* $g \circ f \colon X \to Z$.

The complete the definition, the composition of arrows has to be associative, and the identity has to act as a neutral element for it (on the left and on the right).

Finally, define an arrow $f \colon X \to Y$ to be an *isomorphism* if there exists an arrow $g \colon Y \to X$ such that $g \circ f = \mathrm{Id}_X$ and $f \circ g = \mathrm{Id}_Y$. If such an isomorphism exists, then X and Y are *isomorphic*.

All these properties are obvious in the above examples, with the natural notion of identity arrow and composition of arrows. In what follows the categories that we used are essentially `Set` and `Vec`, in which all the categorical notions that we are interested in have a natural meaning.

We are now ready to describe what is a (word) language in a category and an automaton in a category.

Definition 2.2. Let us fix an alphabet A, a category $\mathcal{C}$, and two of its objects I and F. A $(\mathcal{C}, I, F)$*-language* L is a map that associates to each word $u \in A^*$ an arrow $L(u) \colon I \to F$ in $\mathcal{C}$.

A $(\mathcal{C}, I, F)$*-automaton* is a tuple:

$$\mathcal{A} = \langle Q, A, i, f, \delta \rangle$$

in which Q is an object from $\mathcal{C}$ called the *state object*, A is the *input alphabet*, $i \colon I \to Q$ is an arrow of $\mathcal{C}$ called the *initial arrow* $f \colon Q \to F$ is an arrow of $\mathcal{C}$ called the *final arrow* and $\delta(a) \colon Q \to Q$ is an arrow of $\mathcal{C}$ for all letters $a \in A$, called the *transition arrows*.

Given a word $u = a_1 \dots a_n \in A^*$, a $(\mathcal{C}, I, F)$-automaton $\mathcal{A}$ *recognizes* the $(\mathcal{C}, I, F)$-language $[\![\mathcal{A}]\!]$ defined for all $u \in A^*$ by:

$$[\![\mathcal{A}]\!](u) = f \circ \underbrace{\delta(a_n) \circ \dots \circ \delta(a_1)}_{\delta^*(u)} \circ i \ .$$

Given a $(\mathcal{C}, I, F)$-language L, an *automaton for* L is a $(\mathcal{C}, I, F)$-automaton that recognizes L.

DETERMINISTIC AUTOMATA

A deterministic automaton is nothing but a $(\texttt{Set}, 1, 2)$-automaton.

VECTOR SPACE AUTOMATA

A vector space automaton is nothing but a $(\texttt{Vec}, \mathbb{K}, \mathbb{K})$-automaton.

As usual when adopting a category theoretic approach, it is not sufficient to know what are the objects we are interested in, but we need also to know how these are related through arrows. In our case, the arrows are the morphisms of automata that we introduce now.

Definition 2.3 (*category of automata for a language*). A *morphism of $(\mathcal{C}, I, F)$-automata* from $\mathcal{A} = \langle Q_{\mathcal{A}}, A, i_{\mathcal{A}}, f_{\mathcal{A}}, \delta_{\mathcal{A}} \rangle$ to $\mathcal{B} = \langle Q_{\mathcal{B}}, A, i_{\mathcal{B}}, f_{\mathcal{B}}, \delta_{\mathcal{B}} \rangle$, is an arrow of $\mathcal{C}$ $h \colon Q_{\mathcal{A}} \to Q_{\mathcal{B}}$ such that for all letters $a \in A$,

$$h \circ i_{\mathcal{A}} = i_{\mathcal{B}}, \qquad h \circ \delta_{\mathcal{A}}(a) = \delta_{\mathcal{B}}(a) \circ h, \qquad \text{and} \quad f_{\mathcal{A}} = f_{\mathcal{B}} \circ h,$$

or said differently, such that the following three diagrams commute:

$$\begin{tikzcd}
& Q_{\mathcal{A}} \arrow[dd, "h"] & Q_{\mathcal{A}} \arrow[r, "\delta_{\mathcal{A}}(a)"] \arrow[dd, "h"'] & Q_{\mathcal{A}} \arrow[dd, "h"] & Q_{\mathcal{A}} \arrow[dd, "h"'] \arrow[dr, "f_{\mathcal{A}}"] & \\
I \arrow[ur, "i_{\mathcal{A}}"] \arrow[dr, "i_{\mathcal{B}}"'] & & & & & F \\
& Q_{\mathcal{B}} & Q_{\mathcal{B}} \arrow[r, "\delta_{\mathcal{B}}(a)"'] & Q_{\mathcal{B}} & Q_{\mathcal{B}} \arrow[ur, "f_{\mathcal{B}}"'] &
\end{tikzcd} \tag{1}$$

Given a $(\mathcal{C}, I, F)$-language L, define the *category of automata for* L to be the category which has as objects the $(\mathcal{C}, I, F)$-automata that recognize L, and as arrows the morphisms of automata. We denote it by:

$$\mathtt{Auto}_L \ .$$

Note that whenever there is a morphisms of automata between two automata, then both have to recognize the same language.

3. ALGEBRAIC MINIMIZATION OF AUTOMATA, AND FACTORIZATIONS IN A CATEGORY

In this section, we explain some features of the category of automata for a language that make minimization of automata possible. There are essentially three required properties: (1) the existence of an initial automaton for the language, and (2) symmetrically, the existence of a final automaton for the language, and (3) the fact that the category of automata for the language has a factorization system. We begin our description with this last point.

3.1. Divisibility and factorization

The standard definition is that a deterministic automaton $\mathcal{M}$ is said *algebraically minimal* if for all other deterministic automata $\mathcal{A}$ for the same language, $\mathcal{M}$ divides $\mathcal{A}$ with the definition:

"$\mathcal{B}$ *divides* $\mathcal{A}$ if $\mathcal{B}$ is the quotient of a subautomaton of $\mathcal{A}$."

Hence we need to understand what is a quotient and what is a subautomaton. Both notions are related to the one of morphisms of automata: indeed, a *quotient* is the image of the automaton under a 'surjective morphism', and a *subautomaton* is an automaton which is sent into the other one under an 'injective morphism'.

The notion of 'surjectivity' and 'injectivity' is the subject of the notion of factorizations in a category that we recall here.[3] An accessible and comprehensive reference for all matters concerning factorization systems is [Adámek et al. 1990].

[3] Usually, an emphasis is put on the fact that quotients correspond to 'regular epis', and subobjects to 'monomorphism'. We try to avoid these case specific considerations, and concentrate here on the properties that arrows should have to be considered as 'surjective like' and 'injective like': namely that the two classes form a factorization system.

Definition 3.1. For two classes of arrows $\mathcal{E}$ and $\mathcal{M}$, we say that $(\mathcal{E}, \mathcal{M})$ forms a *factorization system* if the following conditions hold, where we denote the arrows in $\mathcal{E}$ by two-headed arrows $\twoheadrightarrow$ and the arrows in $\mathcal{M}$ by $\rightarrowtail$.

— The arrows that are both in $\mathcal{E}$ and $\mathcal{M}$ are exactly the isomorphisms.
— The $\mathcal{E}$-arrows are closed under composition.
— The $\mathcal{M}$-arrows are closed under composition.
— For every arrow $f\colon X \to Y$, there exists an $\mathcal{E}$-arrow $e\colon X \twoheadrightarrow Z$ and a $\mathcal{M}$-arrow $m\colon Z \rightarrowtail Y$ such that

$$f = m \circ e\,.$$

This composition is called the *factorization of* f. We also refer to the object Z as the *factorization of* f.

— For every arrow $e\colon X \twoheadrightarrow T$ in $\mathcal{E}$, $g\colon T \to Y$, $f\colon X \to S$ and $m\colon S \rightarrowtail Y$ in $\mathcal{M}$ such that $g \circ e = m \circ f$, there exists one and exactly one arrow $d\colon T \to S$ such that $d \circ e = f$ and $m \circ d = g$. In other words, if the following square commutes, then there exists a unique diagonal arrow such that the resulting diagram commutes:

$$\begin{tikzcd}
X \arrow[r, two heads, "e"] \arrow[d, "f"'] & T \arrow[d, "g"] \arrow[dl, dashed, "d"] \\
S \arrow[r, tail, "m"'] & Y
\end{tikzcd} \tag{2}$$

This property is usually called the *diagonal property* and the unique morphism d is called a *diagonal fill*.

Note that the combination of the above properties make the factorization of an arrow unique up to isomorphisms: indeed, if $m \circ e = m' \circ e' = f$ are two factorizations of an arrow f through Z, respectively Z', then by the diagonal property there exist unique arrows $d\colon Z \to Z'$ and $d'\colon Z' \to Z$, so that $d \circ e = e'$, $d' \circ e' = e$, $m' \circ d = m$ and $m \circ d' = m'$. It readily follows that d and d' are isomorphisms inverse to each other. For example, both $d' \circ d$ and Id_Z are diagonal fills for the square

$$\begin{tikzcd}
X \arrow[r, two heads, "e"] \arrow[d, two heads, "e"'] & Z \arrow[d, tail, "m"] \arrow[dl, dashed, bend right, "\mathrm{Id}_Z"'] \arrow[dl, dashed, bend left, "d' \circ d"] \\
Z \arrow[r, tail, "m"'] & Y\,,
\end{tikzcd} \tag{3}$$

and hence, by uniqueness of the diagonal fill, we have $d' \circ d = \mathrm{Id}_Z$. This is a very standard argument in category theory.

Given a factorization system $(\mathcal{E}, \mathcal{M})$, an *$\mathcal{E}$-quotient* (or simply a *quotient* if $\mathcal{E}$ is clear from the context) of an object X is an arrow $e\colon X \twoheadrightarrow Y$ that belongs to $\mathcal{E}$. For ease of language, it also happens that Y itself is called a *quotient* of X. Similarly, an *$\mathcal{M}$-subobject* of an object X (or simply a *subobject* if $\mathcal{M}$ is clear from the context) is an arrow $m\colon Y \rightarrowtail X$ with m in $\mathcal{M}$.

There may be several pairs of classes of arrows $(\mathcal{E}, \mathcal{M})$ that yield a factorization system in the same category. For instance, in all categories, we can take $\mathcal{E}$ to be the isomorphisms and $\mathcal{M}$ to be all arrows (or the other way round), though this does not give us a very interesting notion… This is the reason why quotients and subobjects are notions relative to the choice of a factorization system. It is nevertheless true that in a lot of situations, a satisfying choice is to chose $\mathcal{E}$ to be the class of 'regular epis', and $\mathcal{M}$ to be the class of 'monomorphisms'; in particular, this works for categories of algebraic structures such as Set, Grp, Vec or Aff. It also works well as for topological spaces Top.

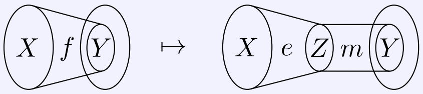
It is good to see some examples.

IN THE CATEGORY OF SETS

In the category of sets, a natural ($\mathcal{E}$,$\mathcal{M}$)-factorization is obtained by taking $\mathcal{E}$ to be the class of surjections and $\mathcal{M}$ the class the injections. The important thing is that every map $f\colon X \to Y$ can be decomposed as a composite of an injective map m and a surjection e:

$$\begin{tikzcd} X \arrow[r, two heads, "e"] & Z \arrow[r, tail, "m"] & Y \end{tikzcd}$$

The codomain of e is the image of f, as can be seen in the next picture.



Moreover, the diagonal property holds. Indeed, interpreting diagram (4) in the category of sets, we define d as follows. For $t \in T$ we put $d(t) = f(x)$ for some $x \in e^{-1}(t)$. Such an x exists since e is surjective, and, moreover, the definition of $d(t)$ does not depend on its choice, since m is injective. Indeed, for any other $x' \in e^{-1}(t)$ we have $f(x) = f(x')$ since m is injective and $m(f(x)) = m(f(x')) = g(t)$.

IN THE CATEGORY OF VECTOR SPACES

In the category of vector spaces, we define $\mathcal{E}$ as the class of surjective linear maps and $\mathcal{M}$ as the class of injective linear maps. Analogously to the Set case, one obtains an ($\mathcal{E}$,$\mathcal{M}$)-factorization.

Furthermore, the notion of factorization naturally yields the one of rank. Indeed if we decompose a linear map $f\colon X \to Y$ as the composite:

$$\begin{tikzcd} X \arrow[r, two heads, "e"] & Z \arrow[r, tail, "m"] & Y \end{tikzcd}$$

then the dimension of Z is exactly the rank of the linear map f.

A reason why the definition of a factorization system is so important is that it is extremely robust: in particular it naturally 'extends component-wise to functor categories', (we will briefly discuss functor categories and their relevance for automata in this context in Section 5.2). In our case, this robustness appears in the following lemma, which follows a standard categorical line of proof:

LEMMA 3.2. *Let $(\mathcal{E}, \mathcal{M})$ be a factorization system for the category $\mathcal{C}$. Then $(\mathcal{E}_{\text{Auto}}, \mathcal{M}_{\text{Auto}})$ forms a factorization system for the category* Auto$_L$ *for all $(\mathcal{C}, I, F)$-languages L, where*

— *$\mathcal{E}_{\text{Auto}}$ consists of these morphisms of automata that happen to belong to $\mathcal{E}$, and*
— *$\mathcal{M}_{\text{Auto}}$ consists of these morphisms of automata that happen to belong to $\mathcal{M}$.*

3.2. Initial and final automata

Apart from factorizations, the other ingredient that is required for having minimal automata is the existence of an initial automaton and of a final automaton in the category of automata for a language.

Definition 3.3. An object I in a category is *initial* if for all objects X there exists a unique arrow $I \to X$. Similarly, an object F is *final* if for all objects X there exists a unique arrow $X \to F$.

Initial and final objects, when they exist, are unique up to isomorphism. We shall see now that in our two running examples of categories of automata, the initial and final objects do exist.

DETERMINISTIC AUTOMATA

Let L be a $(\mathtt{Set}, 1, 2)$-language, i.e. $L(u)$ is a map from 1 to 2.

The *initial automaton for* L is the $(\mathtt{Set}, 1, 2)$-automaton such that:

— The set of states is A^*.
— The initial map $1 \to A^*$ selects ε.
— The final map sends a state $u \in A^*$ to $L(u) \in 2$.
— The transition map for a letter a is $\delta(a)(u) = ua$.

It is easy to check that this automaton recognizes the language L, hence it belongs to $\mathtt{Auto}_L$. A closer inspection reveals that automaton is in fact initial in the category $\mathtt{Auto}_L$.

The *final automaton for* L is the $(\mathtt{Set}, 1, 2)$-automaton such that:

— The states are the $(\mathtt{Set}, 1, 2)$-languages, i.e. the maps from A^* to maps from 1 to 2.
— The initial map sends 1 to L.
— The final map sends state R to $R(\varepsilon)$.
— The transition map for letter a sends the state R to $a^{-1}(R)$ which maps each word u to $R(au)$.

Once more, this automaton recognizes the language L, hence it belongs to $\mathtt{Auto}_L$. Again, a closer inspection reveals that automaton is in fact final in the category $\mathtt{Auto}_L$.

VECTOR SPACE AUTOMATA

Let $L(u)$ be a linear map from $\mathbb{K}$ to $\mathbb{K}$ for all words u. The *initial automaton for* L is such that: is the $(\mathtt{Vec}, \mathbb{K}, \mathbb{K})$-automaton such that:

— The state space is the vector space with basis $(e_u)_{u \in A^*}$.
— The initial map sends $x \in \mathbb{K}$ to xe_ε.
— The final map sends e_u to $L(u)(1_\mathbb{K})$.
— The transition map for the letter a sends e_u to e_{ua}.

This vector space automaton recognises the $(\mathtt{Vec}, \mathbb{K}, \mathbb{K})$-language L. A closer inspection shows that it is in fact initial with this property.

The *final automaton for* L is such that

— The state space is the vector space $\mathbb{K}^{A^*}$ of all functions from A^* to $\mathbb{K}$.
— The initial map sends $1_\mathbb{K} \in \mathbb{K}$ to the function mapping $u \in A^*$ to $L(u)(1_\mathbb{K})$.
— The final map sends $h \in \mathbb{K}^{A^*}$ to $h(\varepsilon)$.
— The transition map for the letter a sends $h \in \mathbb{K}^{A^*}$ to $\lambda u.h(au) \in \mathbb{K}^{A^*}$.

This automaton recognizes the language L, hence it belongs to $\mathtt{Auto}_L$. Yet this time, a closer inspection reveals that automaton is in fact final in the category $\mathtt{Auto}_L$.

In fact, there are some cases—like in the above examples—in which the existence of such initial and final automata for a language exist for easy category theoretic arguments. This is witnessed by the following lemma (which we include here for completeness, although we do no provide the very classical definitions of power and copower in this column, see [Mac Lane 1978]):

LEMMA 3.4. *If the countable copower of I exists in $\mathcal{C}$, then for all $(\mathcal{C}, I, F)$-languages L the category $\mathtt{Auto}_L$ has an initial object, called the* initial automaton for L. *If the countable power of F exists in $\mathcal{C}$, then for all $(\mathcal{C}, I, F)$-languages L the category $\mathtt{Auto}_L$ has a final object, called the* final automaton for L.

In particular, in the running examples, we have:

DETERMINISTIC AUTOMATA

Let L be a $(\texttt{Set}, 1, 2)$-language, i.e. $L(u)$ is a map from 1 to 2.

The set of states A^* of the initial automaton for L is the copower (or coproduct) of A^*-many copies of 1.

The set of states 2^{A^*} of the final automaton for L is the power (or product) of A^*-many copies of 2.

VECTOR SPACE AUTOMATA

Let $L(u)$ be a linear map from $\mathbb{K}$ to $\mathbb{K}$ for all words u.

The vector space of configurations of the initial automaton for L is the copower (or coproduct) of A^*-many copies of $\mathbb{K}$, that is, the direct sum $\bigoplus_{u \in A^*} \mathbb{K}$ of A^*-many copies of $\mathbb{K}$.

The vector space of configurations of the final automaton for L is the power (or product) of A^*-many copies of $\mathbb{K}$, that is, the direct product $\prod_{u \in A^*} \mathbb{K}$ of A^*-many copies of $\mathbb{K}$.

3.3. Minimal automaton and minimization

At last, we are able to provide a general description of why there exists a minimal automaton for a language, and what is the general procedure for minimizing a given automaton.

In fact, the notion of a minimal automaton is now generic, it is a notion that works whenever there is an initial object, a final object, and some factorization system $(\mathcal{E}, \mathcal{M})$.

Consider a factorization system $(\mathcal{E}, \mathcal{M})$ for a category $\mathcal{A}$ and two of its objects X, Y. Let us say that:

X $(\mathcal{E}$,$\mathcal{M})$-*divides* Y if X is an $\mathcal{E}$-quotient of an $\mathcal{M}$-subobject of Y.

Let us note immediately that in general this notion of $(\mathcal{E}$,$\mathcal{M})$-divisibility may not be transitive[4]. It is now natural to define an object M to be $(\mathcal{E}$,$\mathcal{M})$-*minimal* in the category, if it $(\mathcal{E}$,$\mathcal{M})$-divides all objects of the category. Note that there is no reason a priori that an $(\mathcal{E}$,$\mathcal{M})$-minimal object in a category, if it exists, be unique up to isomorphism. Nevertheless, is our case, when the category has both initial and a final object, we can state the following minimization lemma:

LEMMA 3.5. *Let $\mathcal{A}$ be a category with initial object I and final object F and let $(\mathcal{E}, \mathcal{M})$ be a factorization system for $\mathcal{A}$. Define for every object X:*

— $\texttt{Min}$ *to be the factorization of the only arrow from I to F,*
— $\texttt{Reach}(X)$ *to be the factorization of the only arrow from I to X, and*
— $\texttt{Obs}(X)$ *to be the factorization of the only arrow from X to F.*

Then

— $\texttt{Min}$ *is $(\mathcal{E}$,$\mathcal{M})$-minimal, and*
— $\texttt{Min}$ *is isomorphic to both* $\texttt{Obs}(\texttt{Reach}(X))$ *and* $\texttt{Reach}(\texttt{Obs}(X))$ *for all objects X.*

PROOF. The proof essentially consists of a diagram:

[4] There are nevertheless many situations for which it is the case; In particular when the category is regular, and $\mathcal{E}$ happens to be the class of regular epis. This covers in particular the case of all algebraic categories with $\mathcal{E}$-quotients being the standard quotients of algebras, and $\mathcal{M}$-subobjects being the standard subalgebras.

$$\begin{tikzcd}
 & & X \arrow[drr] & & \\
I \arrow[urr, bend left] \arrow[r, two heads] \arrow[drr, bend right, two heads] & \mathtt{Reach}(X) \arrow[ur, tail] \arrow[rr, two heads] & & \mathtt{Obs}(\mathtt{Reach}(X)) \arrow[r, tail] & F \\
 & & \mathtt{Min} \arrow[ur, dashed, no head] \arrow[urr, tail, bend right] & &
\end{tikzcd}$$

Using the definition of `Reach` and `Obs`, and the fact that $\mathcal{E}$ is closed under composition, we obtain that $\mathtt{Obs}(\mathtt{Reach}(X))$ is an ($\mathcal{E}$,$\mathcal{M}$)-factorization of the only arrow from I to F. Thus, thanks to the diagonal property of an factorization system, `Min` and $\mathtt{Obs}(\mathtt{Reach}(X))$ are isomorphic. Hence, furthermore, since $\mathtt{Obs}(\mathtt{Reach}(X))$ ($\mathcal{E}$,$\mathcal{M}$)-divides X by construction, the same holds for `Min`. In a symmetric way, $\mathtt{Reach}(\mathtt{Obs}(X))$ is also isomorphic to `Min`. □

COROLLARY 3.6. *Let L be a $(\mathcal{C}, I, F)$-language for which $\mathcal{C}$ has a factorization system $(\mathcal{E}, \mathcal{M})$ such that* $\mathtt{Auto}_L$ *contains both an initial automaton $\mathcal{I}$ and a final automaton $\mathcal{F}$. Then:*

- *The $(\mathcal{C}, I, F)$-automaton $\mathcal{S}$ for L that is at the middle point of an ($\mathcal{E}_{\mathrm{Auto}}$,$\mathcal{M}_{\mathrm{Auto}}$)-factorization of the only automata morphism from $\mathcal{I}$ to $\mathcal{F}$ is called the* syntactic automaton for L.
- *The syntactic automaton for L $\mathcal{S}$ ($\mathcal{E}_{\mathrm{Auto}}$,$\mathcal{M}_{\mathrm{Auto}}$)-divides every automaton for L.*
- *For all automata $\mathcal{A}$ for L, $\mathcal{S}$ is isomorphic to both* $\mathtt{Reach}(\mathtt{Obs}(\mathcal{A}))$ *and* $\mathtt{Obs}(\mathtt{Reach}(\mathcal{A}))$.

The process of starting from an automaton, and applying to it `Reach` then `Obs` (in any order) is called *'minimization*. Note that implementing `Reach` and `Obs` in an effective way is a problem that may prove difficult on its own, and we do not elaborate on this aspect.

DETERMINISTIC AUTOMATA

It is well known that for all languages $L \subseteq A^*$, there exists a minimal deterministic automaton for it, that furthermore is finite if and only L is regular. Indeed, if L is accepted by a finite automaton $\mathcal{A}$, then, by Corollary 3.6 the syntactic automaton for L divides $\mathcal{A}$, hence its state space must be finite since it is the quotient of a subset of a finite set.

VECTOR SPACE AUTOMATA

Similarly, it is well known that for all languages $L \colon A^* \to \mathbb{K}$, there exists a minimal vector space automaton for it, that furthermore is finite dimensional if and only L is regular. Indeed, if L is accepted by a finite dimensional vector space automaton $\mathcal{A}$, then the syntactic automaton is a quotient of $\mathcal{A}$, hence its state space must be finite dimensional since it is the quotient of a subspace of a finite dimensional space.

4. A NOVEL FORM OF AUTOMATA: HYBRID-SET-VECTOR AUTOMATA

In this section, we describe a new form of automata, which we can call hybrid-set-vector automata, and which extend both deterministic finite automata and vector space automata, while still possessing syntactic automata. We will see how this minimization is obtained along the same lines described in this column so far.

4.1. An intuition

Let us consider the vector space automaton $\mathcal{A}_{\text{vec}}$ from Example 2.1 accepting the map L_{vec}, which corresponds to the weighted language $A^* \to \mathbb{R}$ mapping u to $2^{|u|_a}$ when u does not contain any c and has even number of b's, and to 0 otherwise.

Let us think for a moment on how one would "implement" the function L_{vec} as an online device that would get letters as input, and would modify its internal state accordingly. Would we implement concretely the automaton of Example 2.1 directly? Probably not, since there is a more economic[5] way to obtain the same result: we can maintain 2^m where m is the number of a's seen so far, together with one bit for remembering whether the number of b's is even or odd. Such an automaton would start with 1 in its unique real valued register. Each time an a is met, the register is doubled, each time b is met, the bit is reversed, and when c is met, the register is set to 0. At the end of the input word, the automaton would output 0 or the value of the register depending on the current value of the bit.

If we consider the configuration space that we use in this encoding, we use $\mathbb{R} \uplus \mathbb{R}$ instead of $\mathbb{R} \times \mathbb{R}$. Essentially, the set of vectors spanned by applying in arbitrary order the linear transformations δ_a, δ_b and δ_c from Example 2.1 to the vector $(1, 0) \in \mathbb{R}^2$ is the infinite set of vectors described in the above diagram. Of course, in the category of vector spaces this set spans the whole $\mathbb{R}^2$. Yet, in this example this set lies on the "union" of two one dimensional spaces. Can we define an automaton model that would be able to faithfully implement this example?

4.2. A first generalization: disjoint unions of vector spaces.

A way to achieve this is to interpret the generic notion of automata in the category of finite disjoint unions of vector spaces (*duvs*). One way to define such a *finite disjoint unions of vector spaces* is to use a finite set N of '*indices*' $p, q, r \dots$, and to each index p associate a vector space V_p, possibly with different dimensions. The *corresponding set* is:

$$\{(p, \vec{v}) \mid p \in N,\ \vec{v} \in V_p\}\ .$$

A 'map' between duvs represented by (N, V) and (N', V') is then a pair $h : N \to N'$ together with a linear map f_p from V_p to $V'_{h(p)}$ for all $p \in N$. It can be seen as mapping each $(p, \vec{v}) \in N \times V_p$ to $(h(p), f_p(\vec{v}))$. Call this a *duvs map*. Such duvs maps are composed in a natural way. This defines a category, and hence we can consider *duvs automata* which are automata with a duvs for its state space, and transitions implemented by duvs maps.

For instance, we can pursue with the computation of L_{vec} and provide a duvs automaton

$$\mathcal{A}^{\text{duvs}} = (Q^{\text{duvs}}, i^{\text{duvs}}, f^{\text{duvs}}, \delta^{\text{duvs}})$$

where

$$Q^{\text{duvs}} = \{(s, x) \mid s \in \{\texttt{even}, \texttt{odd}\},\ x \in \mathbb{K}\}$$

(considered as a disjoint union of vector spaces with set of indices $\{\texttt{even}, \texttt{odd}\}$ and all associated vector spaces $V_{\texttt{even}} = V_{\texttt{odd}} = \mathbb{K}$). The maps can be conveniently defined as

[5] Under the assumption that maintaining a real is more costly than maintaining a bit.

follows:

$$\begin{array}{rcc}
i^{\texttt{duvs}}(x) = (\texttt{even}, x) & (\texttt{even}, x) = (\texttt{even}, 2x) & \delta_a^{\texttt{duvs}}(\texttt{odd}, x) = (\texttt{odd}, 2x) \\
f^{\texttt{duvs}}(\texttt{even}, x) = x & \delta_b^{\texttt{duvs}}(\texttt{even}, x) = (\texttt{odd}, x) & \delta_b^{\texttt{duvs}}(\texttt{odd}, x) = (\texttt{even}, x) \\
f^{\texttt{duvs}}(\texttt{odd}, x) = 0 & \delta_c^{\texttt{duvs}}(\texttt{even}, x) = (\texttt{even}, 0) & \delta_c^{\texttt{duvs}}(\texttt{odd}, x) = (\texttt{odd}, 0)
\end{array}$$

This automaton computes the language L_{vec}. Automata over finite disjoint unions of vector spaces generalize both deterministic finite state automata (using only 0-dimensional vector spaces), and vector space automata (using only one index). In this particular example, it can also be seen as a semi-direct product of a two state machine with a purely vector space automaton, (but this remark fails when the spaces V_p have different dimensions.) However, is it the joint generalization that we hoped for? The answer is no...

4.3. Failure to minimize.

We could think that the above automaton $\mathcal{A}^{\texttt{duvs}}$ is minimal. However, it involved some arbitrary decisions when defining it. This can be seen in the fact that when $\delta_c^{\texttt{duvs}}$ is applied, we chose to not change the index (and set to null the real value): this is arbitrary, and we could have exchanged `even` and `odd`, or fixed it arbitrarily to `even`, or to `odd`. All these *variants* would be equally valid as far as computing L_{vec} is concerned.

Let us provide some high level intuitions, invoking some standard automata-theoretic concepts. The first remark is that every configuration in $Q^{\texttt{duvs}}$ is 'reachable' in this automaton: indeed $(\texttt{even}, x) = i^{\texttt{duvs}}(x)$ and $(\texttt{odd}, x) = \delta_b^{\texttt{duvs}} \circ i^{\texttt{duvs}}(x)$ for all $x \in \mathbb{K}$. Hence there is no hope to improve the automaton $\mathcal{A}^{\texttt{duvs}}$ or one of its variants by some form of 'restriction to its reachable configurations'. Only 'quotienting of configurations' remains. However, (using the only reasonable definition of quotient in duvs), one can show that none among $\mathcal{A}^{\texttt{duvs}}$ and its variants is the quotient of another. More precisely, if we keep in mind the Myhill-Nerode equivalence, what we would like to do is to merge the configurations $(\texttt{even}, 0)$ and $(\texttt{odd}, 0)$ since these are observationally equivalent:

$$f^{\texttt{duvs}} \circ \delta_u^{\texttt{duvs}}(\texttt{even}, 0) = 0 = f^{\texttt{duvs}} \circ \delta_u^{\texttt{duvs}}(\texttt{odd}, 0) \qquad \text{for all words } u \in A^*.$$

However, if we try to merge using a duvs map the configurations $(\texttt{even}, 0)$ and $(\texttt{odd}, 0)$, we eventually obtain an automaton with one index associated to a one-dimensional vector space. This would in fact be a vector space automaton, and we already mentioned that such an automaton cannot compute L_{vec}. Overall, there is no minimal duvs automaton for L_{vec}.

4.4. The category of gluings of vector spaces

The subject of this section is to introduce hybrid-set-vector automata, and for this we need to describe the category that they use: the category of gluings of vector spaces

Indeed, after the failure to minimize of the last section, the only reasonable thing to do is to try to merge $(\texttt{even}, 0)$ and $(\texttt{odd}, 0)$, but nothing else (because no other pairs of distinct states are observationally equivalent). This is made possible thanks to a change of category, in which 'gluings' can be performed.

A direct definition

The first solution is to do it naturally: a gluing of vector spaces would be a disjoint union of vector spaces enriched with some 'equivalence' representing how the different components of the duvs have to be glued. Formally, a *gluing of vector spaces* (N, V) consists of

- a set of indices N and vector spaces $(V_p)_{p\in N}$, thus forming a disjoint union of vector spaces, together with
- a *gluing equivalence* $\sim_{\mathrm{glue}}$ which is an equivalence relation on the corresponding set

$$|(N,V)| = \{(p,\vec{v}) \mid p \in N,\ \vec{v} \in V_p\}$$

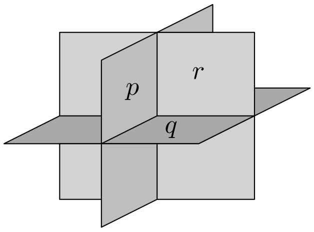
such that for all distinct indices $p, q \in N$,

  - the equivalence classes of $\sim_{\mathrm{glue}}$ restricted to each $\{p\} \times V_p$ are singletons, and
  - the equivalence classes of $\sim_{\mathrm{glue}}$ restricted to $\{p\} \times V_p \cup \{q\} \times V_q$ that are of size 2 form a linear bijection between a subspace of V_p and a subspace of V_q.

An example consists of three copies of $\mathbb{R}^2$, say p, q, r such that furthermore

$$\begin{aligned} (p,x,0) &\sim_{\mathrm{glue}} (q,0,x)\ , \\ (q,x,0) &\sim_{\mathrm{glue}} (r,0,x)\ , \\ \text{and} \qquad (r,x,0) &\sim_{\mathrm{glue}} (p,0,x)\ . \end{aligned}$$

This could be roughly described as the picture to the right.



The definition of *maps of gluings of vector spaces* is then the one of maps of disjoint union of vector spaces, but that would furthermore be required to preserve $\sim_{\mathrm{glue}}$.[6] For instance, the map g which for all $x, y \in \mathbb{R}$ is defined by:

$$\begin{aligned} g(p,x,y) &= (q,y,x) \\ g(q,x,y) &= (r,y,x) \\ g(r,x,y) &= (p,y,x)\ , \end{aligned}$$

does preserve the structure of the above gluing equivalence, and hence is a valid map of gluings of vector spaces.

Note that a map of gluings of vector spaces $f\colon S \to T$ induces a map $|f|\colon |S| \to |T|$. (In fact, this translation is a functor from the category of gluings of vector spaces to the category of sets)

Definition 4.1. The gluings of vector spaces together with the maps between them form a category called the *category of gluings of vector spaces*. We denote it `Glue(Vec)` (we shall give some more explanations about this notation).

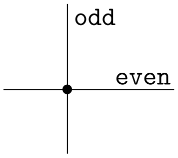
An *hybrid-set-vector automaton* is then simply an automaton in the category of gluings of vector spaces.

Example 4.2. The hybrid-set-vector automaton for L_{vec} that we are interested in can now be described formally:

- the state object is a gluing of vector spaces that consists of two copies of the vector space $\mathbb{R}$, indexed as even and odd that are glued at 0, i.e.,

$$(\mathtt{even}, 0) \sim_{\mathrm{glue}} (\mathtt{odd}, 0)\ .$$

This could be depicted as in the right figure, in which the gluing equivalence is emphasized using a black dot.



[6] In fact, we are making an approximation here, since several such maps should be considered as equivalent. For instance, in our example, the map of gluing of spaces (corresponding to the transition of letter c in $\mathcal{A}^{\mathtt{duvs}}$) that sends $(\mathtt{even}, x) \mapsto (\mathtt{even}, 0)$ and $(\mathtt{odd}, x) \mapsto (\mathtt{odd}, 0)$ is equivalent to the one that sends $(\mathtt{even}, x) \mapsto (\mathtt{even}, 0)$ and $(\mathtt{odd}, x) \mapsto (\mathtt{even}, 0)$, simply because $(\mathtt{odd}, 0) \sim_{\mathrm{glue}} (\mathtt{even}, 0)$.

— The initial map sends x to $(\texttt{even}, x)$.
— The final map sends $(\texttt{even}, x)$ to x and $(\texttt{odd}, x)$ to 0.
— The transition map for letter a sends $(\texttt{even}, x)$ to $(\texttt{even}, 2x)$ and $(\texttt{odd}, x)$ to $(\texttt{odd}, 2x)$.
— The transition map for letter b sends $(\texttt{even}, x)$ to $(\texttt{odd}, x)$ and $(\texttt{odd}, x)$ to $(\texttt{even}, x)$.
— The transition map for letter c sends $(\texttt{even}, x)$ to $(\texttt{even}, 0)$ and $(\texttt{odd}, x)$ to $(\texttt{even}, 0)$.

The category of gluings of vector spaces can be seen as a joint extension of the category of sets and the category of vector spaces. This is the reason for the name "hybrid-set-vector automaton". This remark is made formal now:

LEMMA 4.3. *The category of gluings of vector spaces restricted to gluings of 0-dimension vector spaces is equivalent*[7] *to the category of sets.*
The category of gluings of vector spaces restricted to gluings of vector spaces with one index only is equivalent to the category of vector spaces.

Thanks to the above lemma, we obtain that:

— deterministic automata, which are $(\texttt{Set}, 1, 2)$-automata, are also hybrid-set-vector automata; namely the ones in which the definition of the state object does only involve 0-dimension vector spaces),
— vector space automata, which are $(\texttt{Vec}, \mathbb{K}, \mathbb{K})$-automata, are also hybrid-set-vector automata; namely the one that have only one index in the definition of their state object.

A categorical approach to gluing: the category $\texttt{Glue}(\texttt{Vec})$

The few lines that follow require some background in category theory. However, these are not necessary for understanding the rest of the paper.

Another approach for defining gluings of vector spaces is to consider $\texttt{Vec}$ as a generic category $\mathcal{C}$, and define in categorical terms the 'gluings of objects in $\mathcal{C}$'. We name it $\texttt{Glue}(\mathcal{C})$.

In fact, the reader used to categorical construction knows the standard approach for gluing objects in a category, based on the concept of a free colimit. In this view, the category of gluings of vector spaces can be seen as a subcategory of the free cocompletion of $\texttt{Vec}$. Informally, an element of the free cocompletion of $\texttt{Vec}$ is an (equivalence class of) 'diagrams' that describe a set of objects of $\mathcal{C}$ and give constraints on how these should be 'glued together'. However, this generic description is much less constrained that the one we have described in the previous definition of the gluing of vector spaces. For instance, it is possible using free colimits to take a copy of $\mathbb{R}^2$ and 'glue' together the axis $\mathbb{R}(0,1)$ and $\mathbb{R}(1,0)$ (say using $(0,x) \sim (x,0)$). Such a construction would yield a formal object that is different than the gluings of vector spaces we are interested in.

Thus, our definition of $\texttt{Glue}(\mathcal{C})$ does only use the diagrams that we consider 'meaningful'. It can be (informally) stated as follows:

Definition 4.4. $\texttt{Glue}(\mathcal{C})$ is the free cocompletion of $\mathcal{C}$ restricted to the diagrams that have a cocone in $\mathcal{C}$, all the arrows of which are monos (or more generally in $\mathcal{M}$ for a suitable class $\mathcal{M}$).

The advantage of this approach is that it can be used with other categories; for instance for constructing the category of *gluings of affine spaces* (i.e. $\texttt{Glue}(\texttt{Aff})$) or even *gluing of sets* (i.e. $\texttt{Glue}(\texttt{Set})$). These categories have interest on their own, that we do not develop in this column.

[7] *Equivalent* is the proper notion of 'isomorphism' for categories. Technically, it is an isomorphism 'up to isomorphisms of the objects'.

4.5. The minimization of hybrid-set-vector automata

In the previous section, we introduced the concept of hybrid-set-vector automata; these automata live in the category of gluings of vector spaces. Our motivation was to explain how these can be minimized. In fact, we shall see that maybe these are not exactly the automata we are interested in.

Let us recall that we had identified three ingredients for the existence of minimal automata for a language: the existence of an initial automaton, the existence of a final automaton, and the existence of a factorization system. Let us review what is the status of the category of hybrid-set-vector automata for a language with respect to these three points.

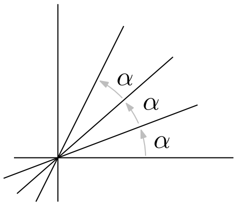
The following lemma can be proved using a more generic argument for handling initial and final automata:

LEMMA 4.5. *The category of hybrid-set-vector automata for a language has an initial and a final object.*[8]

The more interesting part concerns the third ingredient required for having minimal automata: the existence of a factorization system for $\mathtt{Glue}(\mathtt{Vec})$. Indeed, it has one, but the issue is that... this is not what we really are looking for, as shown by the following example:

Example 4.6. Consider the language which to a word $u \in a^*$ associates the value $\cos(\alpha|u|)$ for some α which is not a rational multiple of π. This can be recognized by a vector space automaton as follows:

- the vector space of configurations is $\mathbb{R}^2$,
- the initial map maps x to $(x, 0)$,
- the final map maps (x, y) to x, and
- the transition map for the letter a performs a rotation of α radian of the plane $\mathbb{R}^2$: it maps (x, y) to $(\cos(\alpha)x - \sin(\alpha)y, \sin(\alpha)x + \cos(\alpha)y)$:



(This automaton can also be seen as a hybrid-set-vector automaton.) Now, the $\mathtt{Glue}(\mathtt{Vec})$-automaton obtained by minimizing (in fact, simply by restriction to the reachable states), consists of countably many copies of the line $\mathbb{R}$, all glued at 0. More precisely,

- the state object is the gluing of vector spaces that has indices $\mathbb{N}$, each vector space V_n is $\mathbb{R}$, and the gluing equivalence merges all the 0 points: $(m, x) \sim_{\text{glue}} (n, y)$ if $m = n$ and $x = y$, or if $x = y = 0$,
- the initial map maps $\mathbb{R}$ to the vector space of index 0, i.e. it maps every $x\mathbb{R}$ to $(0, x)$,
- the final map maps (m, x) to $\cos(\alpha m)x$,
- and the transition maps sends (m, x) to $(m + 1, x)$.

[8] In fact, $\mathtt{Glue}(\mathcal{C})$ has all coproducts (and as a consequence all copowers), and furthermore all colimits that $\mathcal{C}$ has. If $\mathcal{C}$ has all colimits and products, then $\mathtt{Glue}(\mathcal{C})$ also has all products (and hence all powers). If $\mathcal{C}$ has all limits and colimits then $\mathtt{Glue}(\mathcal{C})$ also has them.

It happens that this automaton is the minimal one (This automaton would still be correct if α would be a rational multiple of π, but in this case, it could not be minimal). What we see here is that this automaton is merely storing the "current rotation" in the index part of the gluing of vector spaces of configurations.

The above example shows that minimizing without further caution leads to a problem. Indeed, we started from a perfectly valid dimension 2 vector space used for recognizing a language, and after minimizing it it became an automaton that uses a countable union of vector spaces as configurations: something that is more difficult to handle effectively. The answer to this problem lies in a assumption that was left unspoken so far. We are interested only in automata involving 'finite gluings of finite dimension vector spaces' only, because these are the automata that can be used algorithmically. We get the following definition:

Definition 4.7. A gluing of vector spaces is *effective* if it has a finite index, and all the vector spaces involved in its definition are of finite dimension. In the same way, the hybrid-set-vector automata that have an effective state object are also named *effective*.

At the categorical level, this means to define $\mathtt{Glue}_{\mathsf{fin}}(\mathcal{C})$, and construct this way the category $\mathtt{Glue}_{\mathsf{fin}}(\mathtt{Vec}_{\mathsf{fin}})$ where $\mathtt{Vec}_{\mathsf{fin}}$ is the category:

$$\mathtt{Vec}_{\mathsf{fin}} = (\text{finite dimension vector spaces}, \text{linear maps}),$$

with the natural notion of composition of linear maps and identity map.

To complete the picture, it is necessary to explain how to minimize in the world of the effective hybrid-set-vector automata. However, following the line of descriptions seen so far, there is a problem:

— We need to use hybrid-set-vector automata in their general form, since the initial and the final automaton, which are essential parts in the minimization arguments, are never effective (unless the input alphabet is empty...).

— However, we want the 'minimal automaton' that we construct as a result of a factorization to be effective.

The way to resolve this conflict is to modify in a simple and natural, yet unconventional to our knowledge, way the notion of factorization system. We substitute to it the notion of "factorization system through". The heart of this definition is to consider a subcategory $\mathcal{S}$ (that we think of as the category of small/manageable objects) of a larger category $\mathcal{C}$:

Definition 4.8. Consider a category $\mathcal{C}$, a full subcategory $\mathcal{S}$ of $\mathcal{C}$. Call *$\mathcal{S}$-small* an arrow $f\colon X \to Y$ of $\mathcal{C}$ that factors through $\mathcal{S}$, i.e. such that $f = h \circ g$ with $g\colon X \to Z$ and $h\colon Z \to Y$ for some Z object of $\mathcal{S}$.

Example 4.9. Consider the category $\mathcal{C}$ of $(\mathtt{Set}, 1, 2)$-automata, which are the deterministic automata, and its subcategory $\mathcal{S}$ of $(\mathtt{Set}_{\mathsf{fin}}, 1, 2)$-automata (where $\mathtt{Set}_{\mathsf{fin}}$ is the subcategory of finite sets), which is the category of finite deterministic automata. Then for a language L the only morphism from the initial automaton for L to the final automaton for L is $\mathcal{S}$-small if and only if L is a regular language. Indeed, being $\mathcal{S}$-small in this example means exactly being accepted by a finite deterministic automaton.

If one takes the category $\mathcal{C} = \mathtt{Vec}$ and its subcategory $\mathcal{S} = \mathtt{Vec}_{\mathsf{fin}}$, then a linear map in $\mathcal{C}$ is $\mathcal{S}$-small if and only if it is of finite rank.

We now introduce the refined notion of factorization through. It essentially formalizes what it is to be a factorization system that factorizes only $\mathcal{S}$-small arrows, and further do it in $\mathcal{S}$. Formally, the definition is only a slight variation around the one for a factorization system.

Definition 4.10. Let $\mathcal{S}$ be a subcategory of a category $\mathcal{C}$, and $\mathcal{E}_\mathcal{S}, \mathcal{M}_\mathcal{S}$ be two classes of arrows such that:

- all arrows in $\mathcal{E}_\mathcal{S}$ end in $\mathcal{S}$, and
- all arrows in $\mathcal{M}_\mathcal{S}$ start in $\mathcal{S}$,

then $(\mathcal{E}_\mathcal{S}, \mathcal{M}_\mathcal{S})$ forms a *factorization system through* $\mathcal{S}$ if the following conditions hold, where we denote the arrows in $\mathcal{E}_\mathcal{S}$ by two-headed arrows $\twoheadrightarrow$ and the arrows in $\mathcal{M}_\mathcal{S}$ by $\rightarrowtail$.

- The arrows that are both in $\mathcal{E}_\mathcal{S}$ and $\mathcal{M}_\mathcal{S}$ are exactly the isomorphisms in $\mathcal{S}$.
- The $\mathcal{E}_\mathcal{S}$-arrows are closed under composition.
- The $\mathcal{M}_\mathcal{S}$-arrows are closed under composition.
- For all $\mathcal{S}$-small arrows $f\colon X \to Y$, there exists some object Z of $\mathcal{S}$, an $\mathcal{E}_\mathcal{S}$-arrow $e\colon X \twoheadrightarrow Z$ and a $\mathcal{M}_\mathcal{S}$-arrow $m\colon Z \rightarrowtail Y$ such that
$$f = m \circ e\ .$$
This composition is called the *factorization of f through $\mathcal{S}$*. We also refer to the object Z as the *factorization of f through $\mathcal{S}$*.
- For all arrows $e\colon X \twoheadrightarrow T$ in $\mathcal{E}_\mathcal{S}$, $g\colon T \to Y$, $f\colon X \to S$ and $m\colon S \rightarrowtail Y$ in $\mathcal{M}_\mathcal{S}$ such that $g \circ e = m \circ f$, there exists one and exactly one arrow $d\colon T \to S$ (of $\mathcal{S}$) such that $d \circ e = f$ and $m \circ d = g$. In other words, if the following square commutes, then there exists a unique diagonal arrow such that the resulting diagram commutes:
$$\begin{tikzcd} X \arrow[r, two heads, "e"] \arrow[d, "f"'] & T \arrow[d, "g"] \arrow[dl, dashed, "d"] \\ S \arrow[r, tail, "m"'] & Y \end{tikzcd} \tag{4}$$
As can be expected, this property is also called the *diagonal property*, and the unique morphism d is called a *diagonal fill*.

In fact, what happens is that all the proofs that we have done so far, and in particular the existence of a minimal object, Lemma 3.5, can be adapted to this variation of the notion of factorization system.

For instance, Lemma 3.5, becomes:

LEMMA 4.11. *Let $\mathcal{A}$ be a category with initial object I and final object F and let $(\mathcal{E}_\mathcal{S}, \mathcal{M}_\mathcal{S})$ be a factorization system though a subcategory $\mathcal{S}$ for $\mathcal{A}$. Assume furthermore that the only arrow from I to F is $\mathcal{S}$-small, and define for all objects X of $\mathcal{S}$:*

- $\mathtt{Min}_\mathcal{S}$ *to be the factorization through $\mathcal{S}$ of the only arrow from I to F,*
- $\mathtt{Reach}_\mathcal{S}(X)$ *to be the factorization through $\mathcal{S}$ of the only arrow from I to X (note that this arrow is $\mathcal{S}$-small since X is an object of $\mathcal{S}$), and*
- $\mathtt{Obs}_\mathcal{S}(X)$ *to be the factorization through $\mathcal{S}$ of the only arrow from X to F (note that it is $\mathcal{S}$-small since X is an object of $\mathcal{S}$).*

Then

- $\mathtt{Min}_\mathcal{S}$ *is ($\mathcal{E}_\mathcal{S}$,$\mathcal{M}_\mathcal{S}$)-minimal (for the natural definition of it), and*
- $\mathtt{Min}_\mathcal{S}$*,* $\mathtt{Obs}_\mathcal{S}(\mathtt{Reach}_\mathcal{S}(X))$ *and* $\mathtt{Reach}_\mathcal{S}(\mathtt{Obs}_\mathcal{S}(X))$ *are isomorphic for all objects X of $\mathcal{S}$.*

Now, almost all the landscape is ready. We have a class of automata, the hybrid-set-vector automata. It has an initial and a final object according to Lemma 4.5. We have also identified the subcategory of effective hybrid-set-vector automata for the language. It remains to tell what are the classes $\mathcal{E}_\mathcal{S}$ and $\mathcal{M}_\mathcal{S}$ that we use for 'factorizing through' (Lemma 4.11).

Definition 4.12. The *effective monos* are the arrows $m\colon X \to Y$ in the category of hybrid-set-vector automata where X is effective, and $|m|$ is injective, i.e., the map is injective when all structure has been forgotten.

The *effective extremal epis* are the arrows $e\colon X \to Y$ with Y effective, and such that whenever $e = m \circ f$ for some effective mono m, then m is an isomorphism.

Under this definition, we obtain the expected factorization system through.

LEMMA 4.13. *(Effective extremal epis, effective monos) forms a factorization system of the category of hybrid-set-vector automata through the subcategory of effective hybrid-set-vector automata.*

If we briefly summarize what we have, we get the following statement.

THEOREM 4.14. *For all languages accepted by an effective hybrid-set-vector automaton, there exists a (effective extremal epis, effective monos)-minimal equivalent one.*

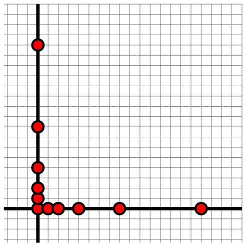
In particular, the effective hybrid-set-vector automaton of Example 4.2 is minimal as in the above theorem.

Though we do not provide more detail here on why this result holds (not to mention the effectivity of the constructions), let us emphasize that it relies crucially on the following statement:

> In a finite dimension vector space E, for every set $X \subseteq E$, there exists a finite union of vector spaces $F \subseteq E$ such that $X \subseteq F$ and which is minimal for inclusion.

This statement is not difficult to establish. At any rate, it gives a good intuition of what is happening.

At the very beginning, we have introduced the vector space automaton $\mathcal{A}_{\text{vec}}$ that accepts the language L_{vec}. Its state space is $\mathbb{R}^2$. However, starting, say, from $(1, 0)$, we can draw all the configurations that are reachable by applications of the transition maps associated to the letters. We obtain the picture to the right.



Now, we can apply the above statement: there is a least finite union of subspaces that covers the red points. Indeed, it consists of the union $\mathbb{R}(0, 1) \cup \mathbb{R}(1, 0)$. If now, we forget the fact that these two dimension 1 vector spaces are subspaces of $\mathbb{R}^2$, then what remains is a union of two copies of $\mathbb{R}$ that are glued at 0. This is how one obtains the automaton of Example 4.2.

5. DISCUSSION: AUTOMATA AND CATEGORY THEORY

The idea of using category theory to provide a unifying framework for automata theory and linear systems of control theory goes back all the way to the late sixties, and ever since there has been a substantial body of research on this topic. Perhaps it is not a coincidence, that Samuel Eilenberg, one of founders of the field of category theory is also the author of the influential automata theory books [Eilenberg 1974; Eilenberg 1976], which axiomatise (without using category theory!) the algebraic approach to automata theory and provide among others his celebrated variety theorem.

On the other hand, cutting-edge research in automata theory relies at times on combinatorial aspects, which seem difficult, if not impossible, to explain in a generic setting. The purpose of this column is to convince people coming from the combinatorial background, and who have not been previously exposed to category theory, that the conceptual view point is not just an idle exercise in abstract nonsense, and that sometimes it can lead to new interesting combinatorial problems.

As a disclaimer, it was not our intention to provide an exhaustive survey of the interplay between automata and category theory, nor of the huge literature on this very topic. But we feel necessary to highlight some contributions and perspectives.

Early days: Machines in a category. Already in the late 60s, Eilenberg and Wright [Eilenberg and Wright 1967] gave a generalised notion of language recognizability in categories of algebras and formulated their results in terms of Lawvere's algebraic theories, the back then fresh approach to categorical algebra. A couple of years later, Arbib and Manes further advanced the category-theoretic unification of sequential machines and linear systems of control theory in a series of seminal papers [Arbib and Manes 1975; Arbib and Manes 1974a; Arbib and Manes 1974b]. One of their contributions was the connection between minimization and factorization systems, as well as an account of the duality between reachability and observability in this setting, building on the work of Kalman on linear systems [Kalman 1963]. In parallel, Goguen [Goguen 1972] also developed a theory of minimal realization in the setting of monoidal closed categories. A nice survey of these early developments can be found in [Arbib and Manes 1980].

Automata as algebras for a functor. Arbib and Manes advanced the view of sequential machines and linear systems as algebras for a functor, although they adopted a different terminology in those early papers (algebras for a functor were called dynamorphisms). In this setting, the transition map is captured as an algebra for a functor F, that is a map of the form $\delta\colon FQ \to Q$. For example, for deterministic finite automata on a finite alphabet A, we would use the functor $F\colon \mathtt{Set} \to \mathtt{Set}$ given by $FX = A \times X$. This approach was further developed in the work of the Prague seminar on General Mathematical Structures by Věra Trnková, Jiři Adámek, Jan Reiterman, Václav Koubek, see for example the book [Adámek and Trnková 1989] and the references therein. These works explore free algebras for finitary functors, existence and universality of minimal realisations, as well descriptions of languages via rational operations. In particular [Adámek and Trnková 1989, Theorem III.2.14], similar in spirit to the developments we presented in Section 3.3, establishes sufficient conditions for the existence of minimal realization via factorization systems for automata modeled as algebras for coadjoint functors preserving epimorphisms.

Yet, automata are not entirely modelled as algebras. The initial state can be incorporated in the type of the functor, but specifying the final states is outside the realm of algebra. For example, for deterministic finite automata, one can consider the functor given by $FX = 1 + A \times X$. Formally a deterministic finite automaton is a map $[i, \delta]\colon 1 + A \times Q \to Q$, plus the characteristic function of the subset of final states $f\colon Q \to 2$.

Automata as coalgebras for a functor. Alternatively, one could model deterministic finite automata as maps of the form $\langle f, \gamma\rangle\colon Q \to 2 \times Q^A$ obtained by pairing the characteristic function of the subset of accepting states and the map $\gamma\colon Q \to Q^A$, obtained from the transition map δ via currying. The map $\langle f, \gamma\rangle$ is an example of a coalgebra for the functor $G\colon \mathtt{Set} \to \mathtt{Set}$, defined by $GX = 2 \times X^A$, however, in this framework, it is the initial state which is left out.

The view of automata (and more generally of systems) as coalgebras was put forward in the work of Rutten and Jacobs, see [Jacobs and Rutten 1997] and [Rutten 2000]. However, the roots of coalgebras in computer science go back to the work of Aczel and his insights into the coinductive nature of Milner-Park notion of bisimulation from concurrency theory. The coalgebraic view of automata, and the connections to other fields of computer science proved surprisingly useful. An example of a success story is the work of Bonchi and Pous [Bonchi and Pous 2013], which gives an efficient algorithm for deciding language equivalence for non-deterministic finite automata, using enhancements of the coinductive proof techniques (the so-called up-to techniques) and drawing on previous developments from concurrency theory.

The coalgebraic method (along with the numerous contributions in this research area) was described in the *Semantics Column* of a previous SIGLOG news issue [Silva 2015] and was illustrated by giving a category-theoretic account of Brzozowski's minimization algorithm. Which brings us to...

Minimization. A dual narrative: algebra vs. coalgebra. Automata minimization was understood both algebraically at different levels of generality, (as in the work of Arbib and Manes, Adámek and Trnková, etc.), as well as coalgebraically, see for example [Adámek et al. 2012; Bonchi et al. 2012]. Notably, [Bonchi et al. 2012] carries out an elegant study of minimization algorithms for linear weighted automata from a coalgebraic viewpoint.

The interplay between these two views of automata, both as algebras and as coalgebras, was fully exploited in [Bonchi et al. 2014] to give a category-theoretic account of Brzozowski's minimization algorithm. The paper [Rot 2016] showcases the connection between two approaches to minimization (either by partition refinement or reverse-determinisation)—one involving an initial algebra construction, the other a final coalgebra construction. The deep connection between minimization and duality theory were also investigated in [Bezhanishvili et al. 2012].

We should mention in passing that duality theory plays a fundamental role in language theory on aspects related to recognition (see [Gehrke et al. 2008; Gehrke et al. 2010]), and, coincidentally, this is also featured in this issue, in the *Complexity* column. These works were the starting point of the ERC project *DuaLL* which made this collaboration possible. In the same spirit, the recent paper [Gehrke et al. 2017] explores other ideas from category and duality theory for tackling problems in language theory.

5.1. Beyond that point

In this column, we have attempted to show how category theory gives an insight into the nature of automata and the questions of minimization. These facts are well known for decades, though we adopted a presentation which we believe to be simpler and more direct. There are many continuations of this description that could be followed from that point, and space does not permit it.

5.2. Automata as functors

For the category-theoretic minded readers, we would like to emphasise some aspects of the approach described here. In this subsection we will assume more category-theoretic background on the part of the reader.

Although we haven't mentioned this explicitly thus far, our view of automata is neither algebraic, nor coalgebraic, but a "combination" of the two. Formally, we view automata as functors

$$\mathcal{A}\colon \mathcal{I} \to \mathcal{C}$$

where $\mathcal{I}$ is a category of inputs and $\mathcal{C}$ is the category which specifies the universe of output values. For example, for word automata, the category $\mathcal{I}$ has three objects $\mathsf{in}, \mathsf{states}$ and out, and for each $w \in A^*$, arrows $\triangleright w\colon \mathsf{in} \to \mathsf{states}$, $w\triangleleft\colon \mathsf{states} \to \mathsf{out}$ and $w\colon \mathsf{states} \to \mathsf{states}$, generated from the arrows pictured below, so that $w' \circ w$ is defined as the concatenation ww'.

$$\begin{tikzcd}
\mathsf{in} \arrow[r, "\triangleright"] & \mathsf{states} \arrow[loop above, "a"] \arrow[r, "\triangleleft"] & \mathsf{out}
\end{tikzcd}$$

DETERMINISTIC AUTOMATA

A deterministic automaton is obtained by instantiating $\mathcal{C}$ to $\mathtt{Set}$ and considering functors that map in to 1 and out to 2.

VECTOR SPACE AUTOMATA

A vector space automaton is obtained by instantiating $\mathcal{C}$ to $\mathtt{Set}$ and considering functors that map both in and out to $\mathbb{K}$.

In this approach, a language on words can be seen as a functor $L\colon \mathcal{O} \to \mathcal{C}$ from the full subcategory $\mathcal{O}$ of $\mathcal{I}$ on objects in and out

$$\mathsf{in} \xrightarrow{\triangleright w \triangleleft} \mathsf{out}\,,$$

the arrows of which are $\triangleright w \triangleleft\colon \mathsf{in} \to \mathsf{out}$ for all $w \in A^*$. We denote by $\iota\colon \mathcal{O} \to \mathcal{I}$ the inclusion of the category $\mathcal{O}$ in $\mathcal{I}$. An automaton $\mathcal{A}$ accepts the language L if

$$\mathcal{A} \circ \iota = L$$

DETERMINISTIC AUTOMATA

A language accepted by a deterministic automaton is a functor

$$L\colon \mathcal{O} \to \mathtt{Set}$$

mapping in to 1 and out to 2.

VECTOR SPACE AUTOMATA

A language accepted by a vector space automaton is a functor

$$L\colon \mathcal{O} \to \mathtt{Vec}$$

mapping both in and out to $\mathbb{K}$.

It is easy to see that in these cases, we retrieve the running examples discussed in Section 2. If the category $\mathcal{C}$ has countable products and coproducts, then the existence of the initial and final automaton accepting a given language can also be explained in terms of more generic category-theoretic constructions (left and right Kan extensions). In the process of writing this paper, we discovered that a similar approach based on Kan extensions, was considered in an old (and seemingly forgotten) paper of Bainbridge [Bainbridge 1974].

In this framework, we can obtain new automata models by varying the input and the output categories, $\mathcal{I}$, respectively $\mathcal{O}$. For example, the hybrid-set-vector automata of Section 4 are obtained by instantiating $\mathcal{C}$ with $\mathtt{Glue}(\mathtt{Vec})$ and $\mathtt{Glue}_{\mathsf{fin}}(\mathtt{Vec}_{\mathsf{fin}})$.

Syntactic algebras. In a recent paper [Bojańczyk 2015], Bojańczyk considered languages recognised by monads and described syntactic algebras in this setting. By tuning the input category $\mathcal{I}$ so that the algebraic structure at issue is hard-wired in the morphisms of $\mathcal{I}$, we obtain a unifying view of syntactic algebras and minimization.

Tree automata. For handling tree automata, it is necessary to possess a way to describe ‘maps’ of several arguments. Of course in the category $\mathtt{Set}$ of sets we can just use the cartesian product. But in $\mathtt{Vec}$ one has to use the tensor product instead. More generally, the right setting for this is to use monoidal categories. These are categories equipped with a ‘bifunctor’ $\otimes$ called the *tensor product* that satisfies sufficiently many properties for allowing to aggregate several objects into one in a ‘category-meaningful way'. The results of minimization as described in this column can be mimicked in this generalised context. In particular, hybrid-set-vector automata extend to this tree automata.

Enriched forms of automata. Another extension that is important to consider is when the alphabet also possesses some structure. For instance, one could imagine that the alphabet is infinite and computations are meant to be permutation-invariant, as in nominal automata [Bojańczyk et al. 2014]. Again, we can choose the category $\mathcal{I}$ so that the additional structure (e.g. permutation-invariance) is captured by its structure. In this way one can retrieve minimization results for the resulting automata model.

REFERENCES

Jiří Adámek, Filippo Bonchi, Mathias Hülsbusch, Barbara König, Stefan Milius, and Alexandra Silva. 2012. A Coalgebraic Perspective on Minimization and Determinization. In *Proceedings of the 15th International Conference on Foundations of Software Science and Computational Structures (FOSSACS'12)*. Springer-Verlag, Berlin, Heidelberg, 58–73. DOI:http://dx.doi.org/10.1007/978-3-642-28729-9_4

Jiří (ing) Adámek, Horst Herrlich, and George E. Strecker. 1990. *Abstract and concrete categories : the joy of cats*. Wiley, New York. http://opac.inria.fr/record=b1087598 A Wiley-Interscience publication.

Jiří (ing) Adámek and Věra Trnková. 1989. *Automata and Algebras in Categories*. Springer Netherlands, New York. http://www.springer.com/fr/book/9780792300106

Michael A. Arbib and Ernest G. Manes. 1974a. Basic concepts of category theory applicable to computation and control. In *Category Theory Applied to Computation and Control (Lecture Notes in Computer Science)*, Vol. 25. Springer, 1–34.

Michael A. Arbib and Ernest G. Manes. 1974b. A categorist's view of automata and systems. In *Category Theory Applied to Computation and Control (Lecture Notes in Computer Science)*, Vol. 25. Springer, 51–64.

Michael A. Arbib and Ernest G. Manes. 1975. Adjoint machines, state-behavior machines, and duality. *Journal of Pure and Applied Algebra* 6, 3 (1975), 313 – 344. DOI:http://dx.doi.org/10.1016/0022-4049(75)90028-6

Michael A. Arbib and Ernest G. Manes. 1980. Machines in a category. *Journal of Pure and Applied Algebra* 19 (1980), 9 – 20. DOI:http://dx.doi.org/10.1016/0022-4049(80)90090-0

E. S. Bainbridge. 1974. Adressed machines and duality. In *Category Theory Applied to Computation and Control (Lecture Notes in Computer Science)*, Vol. 25. Springer, 93–98.

Nicolas Bedon. 1996. Finite automata and ordinals. (1996), 119–144.

Nicolas Bedon, Alexis Bès, Olivier Carton, and Chloe Rispal. 2010. Logic and Rational Languages of Words Indexed by Linear Orderings. *Theory Comput. Syst.* 46, 4 (2010), 737–760. DOI:http://dx.doi.org/10.1007/s00224-009-9222-6

Nick Bezhanishvili, Clemens Kupke, and Prakash Panangaden. 2012. *Minimization via Duality*. Springer Berlin Heidelberg, Berlin, Heidelberg, 191–205. DOI:http://dx.doi.org/10.1007/978-3-642-32621-9_14

Mikołaj Bojańczyk. 2011. Data Monoids. In *STACS 2011: 28th International Symposium on Theoretical Aspects of Computer Science (LIPIcs)*, Thomas Schwentick and Christoph Dürr (Eds.), Vol. 9. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 105–116.

Mikołaj Bojańczyk. 2015. Recognisable Languages over Monads. In *DLT (Lecture Notes in Computer Science)*, Vol. 9168. Springer, 1–13.

Mikołaj Bojańczyk, Bartek Klin, and Slawomir Lasota. 2014. Automata theory in nominal sets. *Logical Methods in Computer Science* 10, 3 (2014).

Mikołaj Bojańczyk and Igor Walukiewicz. 2008. Forest algebras. In *Logic and Automata: History and Perspectives [in Honor of Wolfgang Thomas].* 107–132.

Filippo Bonchi, Marcello Bonsangue, Michele Boreale, Jan Rutten, and Alexandra Silva. 2012. A coalgebraic perspective on linear weighted automata. *Information and Computation* 211 (2012), 77 – 105. DOI:http://dx.doi.org/10.1016/j.ic.2011.12.002

Filippo Bonchi, Marcello M. Bonsangue, Helle Hvid Hansen, Prakash Panangaden, Jan J. M. M. Rutten, and Alexandra Silva. 2014. Algebra-coalgebra duality in Brzozowski's minimization algorithm. *ACM Trans. Comput. Log.* 15, 1 (2014), 3:1–3:29.

Filippo Bonchi, Marcello M. Bonsangue, Jan J. M. M. Rutten, and Alexandra Silva. 2012. Brzozowski's Algorithm (Co)Algebraically. In *Logic and Program Semantics (Lecture Notes in Computer Science)*, Vol. 7230. Springer, 12–23.

Filippo Bonchi and Damien Pous. 2013. Checking NFA equivalence with bisimulations up to congruence. In *POPL*. ACM, 457–468.

Walter S. Brainerd. 1968. The Minimalization of Tree Automata. *Information and Control* 13, 5 (1968), 484–491. DOI:http://dx.doi.org/10.1016/S0019-9958(68)90917-0

Olivier Carton, Thomas Colcombet, and Gabriele Puppis. 2011. Regular Languages of Words over Countable Linear Orderings. In *ICALP 2011 (2): Automata, Languages and Programming - 38th International Colloquium*, Luca Aceto, Monika Henzinger, and Jiri Sgall (Eds.), Vol. 6756. 125–136.

Christian Choffrut. 1979. A Generalization of Ginsburg and Rose's Characterization of G-S-M Mappings. In *Automata, Languages and Programming, 6th Colloquium, Graz, Austria, July 16-20, 1979, Proceedings (Lecture Notes in Computer Science)*, Hermann A. Maurer (Ed.), Vol. 71. Springer, 88–103. DOI:http://dx.doi.org/10.1007/3-540-09510-1_8

Christian Choffrut. 2003. Minimizing subsequential transducers: a survey. *Theor. Comput. Sci.* 292, 1 (2003), 131–143. DOI:http://dx.doi.org/10.1016/S0304-3975(01)00219-5

Thomas Colcombet. 2009. The theory of stabilisation monoids and regular cost functions. In *ICALP 2009 (2): Automata, Languages and Programming, 36th International Colloquium*, Susanne Albers, Alberto Marchetti-Spaccamela, Yossi Matias, Sotiris E. Nikoletseas, and Wolfgang Thomas (Eds.), Vol. 5556. 139–150.

Thomas Colcombet. 2013. Regular cost functions, Part I: logic and algebra over words. 9, 3 (2013), 47.

Thomas Colcombet and Sreejith A. V. 2015. Limited Set Quantifiers over Countable Linear Orders. In *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015 (Lecture Notes in Computer Science)*, Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann (Eds.), Vol. 9135. Springer, 146–158. DOI:http://dx.doi.org/10.1007/978-3-662-47666-6_12

Samuel Eilenberg. 1974. *Automata, Languages, and Machines*. Vol. Volume A. Academic Press, Inc., Orlando, FL, USA.

Samuel Eilenberg. 1976. *Automata, Languages, and Machines*. Vol. Volume B. Academic Press, Inc., Orlando, FL, USA.

Samuel Eilenberg and Jesse B. Wright. 1967. Automata in general algebras. *Information and Control* 11, 4 (1967), 452 – 470. DOI:http://dx.doi.org/10.1016/S0019-9958(67)90670-5

Mai Gehrke, Serge Grigorieff, and Jean-Eric Pin. 2008. Duality and Equational Theory of Regular Languages. In *ICALP (2) (Lecture Notes in Computer Science)*, Vol. 5126. Springer, 246–257.

Mai Gehrke, Serge Grigorieff, and Jean-Eric Pin. 2010. A Topological Approach to Recognition. In *ICALP (2) (Lecture Notes in Computer Science)*, Vol. 6199. Springer, 151–162.

Mai Gehrke, Daniela Petrişan, and Luca Reggio. 2017. Quantifiers on languages and codensity monads. *LICS* (2017).

J. A. Goguen. 1972. Minimal realization of machines in closed categories. *Bull. Amer. Math. Soc.* 78, 5 (09 1972), 777–783. http://projecteuclid.org/euclid.bams/1183533991

Bart Jacobs and Jan Rutten. 1997. A Tutorial on (Co)Algebras and (Co)Induction. *EATCS Bulletin* 62 (1997), 62–222.

R. E. Kalman. 1963. Mathematical Description of Linear Dynamical Systems. *Journal of the Society for Industrial and Applied Mathematics Series A Control* 1, 2 (Jan. 1963), 152–192. DOI:http://dx.doi.org/10.1137/0301010

Denis Kuperberg. 2011. Linear temporal logic for regular cost functions. In *STACS 2011: 28th International Symposium on Theoretical Aspects of Computer Science (LIPIcs)*, Thomas Schwentick and Christoph Dürr (Eds.), Vol. 9. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 627–636.

Saunders Mac Lane. 1978. Categories for the Working Mathematician. (1978). http://link.springer.com/book/10.1007/978-1-4757-4721-8

Dominique Perrin and Jean-Éric Pin. 1995. Semigroups and automata on infinite words. In *NATO Advanced Study Institute Semigroups, Formal Languages and Groups*, J. Fountain (Ed.). Kluwer academic publishers, 49–72.

Libor Polák. 2001. Syntactic Semiring of a Language. In *Mathematical Foundations of Computer Science 2001, 26th International Symposium, MFCS 2001 Marianske Lazne, Czech Republic, August 27-31, 2001, Proceedings*. 611–620. DOI:http://dx.doi.org/10.1007/3-540-44683-4_53

Michael O. Rabin and Dana Scott. 1959. Finite automata and their decision problems. *IBM J. Res. and Develop.* 3 (April 1959), 114–125.

Jurriaan Rot. 2016. Coalgebraic Minimization of Automata by Initiality and Finality. *Electronic Notes in Theoretical Computer Science* 325 (2016), 253 – 276. DOI:http://dx.doi.org/10.1016/j.entcs.2016.09.042

J.J.M.M. Rutten. 2000. Universal coalgebra: a theory of systems. *Theoretical Computer Science* 249, 1 (2000), 3 – 80. DOI:http://dx.doi.org/10.1016/S0304-3975(00)00056-6

M.P. Schützenberger. 1961. On the definition of a family of automata. *Information and Control* 4, 2 (1961), 245 – 270. DOI:http://dx.doi.org/10.1016/S0019-9958(61)80020-X

Marcel-Paul Schützenberger. 1965. On finite monoids having only trivial subgroups. *Information and Control* 8 (1965), 190–194.

Alexandra Silva. 2015. A Short Introduction to the Coalgebraic Method. *ACM SIGLOG News* 2, 2 (April 2015), 16–27. DOI:http://dx.doi.org/10.1145/2766189.2766193

Complexity Column

NEIL IMMERMAN, University of Massachusetts Amherst
immerman@cs.umass.edu

Computational Complexity Questions, such as trying to prove that the NP-complete problem three colorability of graphs (3COLOR) cannot be checked in polynomial time, seemed deceptively simple when I first encountered them as a grad student in the 1970's. So many years later, we still don't even know whether 3COLOR is in the uniform circuit class ACC^0_6, or equivalently, whether it is expressible in first-order logic with sum mod 6 quantifiers and numeric relations $+, *$.

Down at the Regular Language level, we know a great deal. In particular, we have algebraic equations which provide decision procedures for whether or not a given language is a member of certain classes of regular languages.

Mai Gehrke and Andreas Krebs give us here a readable introduction to Stone Duality – the theory behind those magical equations. They explain a program towards using these topological and algebraic methods towards generating equations which might help us decide which problems are in AC^0 and ACC^*. Can we use Stone Duality to characterize the power of first-order quantifiers – in the presence of numeric relations? Read this column to find out.

Stone duality for languages and complexity[1]

Mai Gehrke
CNRS & University Paris Diderot – Paris 7

Andreas Krebs
University of Tübingen

1. INTRODUCTION

Complexity theory and the theory of regular languages both belong to the branch of computer science where the use of resources in computing is the main focus. However, they operate at different levels. While complexity theory seeks to classify computational problems by resource use, such as space and time, regular language theory remains at the very base of this hierarchy and is concerned with classes of computational problems for which membership is (potentially) decidable.

The theory of regular languages has a highly sophisticated topo-algebraic theory developed over the past 50 years. Over the past decade, it has been observed that this theory is in fact a special case of the Stone duality theory applied in semantics, and this makes a generalisation to the setting of language classes from complexity theory possible. This column provides a tailor-made introduction to Stone duality and describes a program and the potential first goals of such a program of generalisation and application in Boolean circuit complexity.

Algebraic automata theory.

The theory of regular languages and automata is an original computer science topic with a rich theory and a wide and still growing range of practical and theoretical applications. The availability of sophisticated mathematical tools from algebra and topology is one of the main strengths of the classical framework. Finite algebras were introduced into the theory early on by Myhill, Nerode and Rabin and Scott, and their power was established by Schützenberger's effective characterisation of star-free languages by means of their syntactic monoids [Schützenberger 1965]. Syntactic monoids provide an abstract and canonical notion of recognition for regular languages and Eilenberg's theorem [Eilenberg 1976] supplies a general framework in which to apply the strategy of Schützenberger's result by characterising those classes of regular languages

[1] This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No.670624) and from the DFG Emmy Noether program (KR 4042/2). The authors also thank the Simons Institute for the Theory of Computing where this contribution was initiated.

for which the corresponding class of monoids is a pseudo-variety, i.e., is closed under homomorphic images, subalgebras, and finite Cartesian products. The success of the algebraic methods was greatly augmented by the introduction in the 1980s of profinite monoids [Pin 2009; Almeida 2005; Weil 2002]. Profinite algebra is required for recognition of *classes* of regular languages. A most powerful combination in this setting is that of Eilenberg's and Reiterman's theorems. Reiterman's theorem is a generalisation of Birkhoff's Variety Theorem from universal algebra. It states that pseudo-varieties of finite algebras are precisely the ones given by *profinite equations* [Reiterman 1982]. Thus Eilenberg-Reiterman theory allows the equational description of certain classes of regular languages and, in cases where researchers have found finite equational bases, this leads to decidable criteria for membership in the corresponding classes. Given the success of the method, the search for generalisations applicable to more general classes of regular languages and closely related objects has been very active.

Tools and separation results for Boolean circuit classes.

There are very few separation results in complexity theory. The most well-known question is whether non-deterministic polynomial time Turing machines have more computational power than deterministic polynomial time Turing machines, or simply $\mathrm{P} \neq \mathrm{NP}$? There are numerous complexity classes besides the two classes P and NP, e.g.: $\mathrm{PSPACE}, \mathrm{NP}, \mathrm{P}, \mathrm{NC}, \mathrm{TC}^k, \mathrm{ACC}^k, \mathrm{AC}^k, \mathrm{NC}^k, \mathrm{NL}$, and L. The definitions of all these classes and many more can be found in any current textbook. Yet the number of complexity classes is outnumbered by the open questions about their relations: P vs. NP, NL vs. L, P vs. NL, to name only a few. Many attempts to directly attack these questions have failed over the last 40 years.

We will focus on the few places where separations are known. Furst, Saxe, and Sipser [Furst et al. 1984] showed in 1981 by a combinatorial and stochastic argument that the language PARITY, consisting of all bit words with an odd number of 1s, is not in AC^0. AC^0 is a circuit complexity class, that is, its members are specified by sequences of Boolean circuits, one for each input length, identifying which words of the given length are accepted. For each n, AC^n is the class of languages given by families of Boolean circuits for which the size of the circuits is polynomial in the length of the input word and the depth of the circuit is of order $\log^n$ in the length of the word. The class AC is the union of the hierarchy AC^n over all n. The ACC hierarchy is obtained by adding gates that can count modulo q for each q. Clearly PARITY is in ACC^0, so the result of Furst, Saxe, and Sipser separates AC^0 from ACC^0. However, the class ACC^0 has not been separated from anything all the way up to NP. We have the following chain of inclusions:

$$\mathrm{PSPACE} \geqslant \mathrm{NP} \geqslant \mathrm{P} \geqslant \mathrm{AC} \ldots \geqslant \mathrm{AC}^2 \geqslant \mathrm{AC}^1 \geqslant \mathrm{NL} \geqslant \mathrm{L} \geqslant \mathrm{ACC}^0 > \mathrm{AC}^0$$

where L stands for logarithmic space. We do know that PSPACE strictly contains AC, but we do not even know that $\mathrm{NP} \neq \mathrm{ACC}^0$.

So while many separations are known and algebraic and topological tools are available for formal language classes inside the regular languages, complexity classes are missing such tools or even more general patterns of how to understand relations between them.

One connection that is available in this setting is through logic: most computational complexity classes have been given characterisations as finite model classes of appropriate logic fragments [Immerman 1999]. For example,

$$\mathrm{AC}^0 = \mathrm{FO}[\mathcal{N}] \quad \text{and} \quad \mathrm{ACC}^0 = (\mathrm{FO} + \mathrm{MOD})[\mathcal{N}] \tag{1}$$

where $\mathcal{N}$ is the set of all numerical predicates, FO is usual first-order logic, and MOD stands for the modular quantifiers MOD_q (one for each remainder), which count the number of true instances (in a finite word) of a formula modulo q. The presence of *arbitrary numerical predicates* is what brings one far beyond the scope of the profinite algebraic theory of regular languages.

In [Behle and Lange 2006] it was shown that logic classes like $FO[<]$ and $FO + MOD[<]$ correspond to 'very uniform' constant depth circuit classes. Hence the difference in our knowledge varies rapidly when one moves to less uniform circuit classes. While $[<]$ and mostly also $[<, +]$ uniform circuit classes are somewhat understood, little is known for $[<, +, *]$ or for non-uniform circuit classes. Non-uniform circuit classes can use a completely different circuit to recognise the set of words of each length, without any connection between the circuits used for different lengths.

Most results in the field are proved using combinatorial and probabilistic, as well as algorithmic methods [Williams 2014]. However, there are a few connections with the topo-algebraic tools of the theory of regular languages. A famous result of Barrington, Compton, Straubing, and Thérien [Barrington et al. 1992] states that a regular language belongs to AC^0 if and only if its syntactic homomorphism is quasi-aperiodic. Although this result relies on [Furst et al. 1984] and no purely algebraic proof is known, being able to characterise the class of regular languages that are in AC^0 gives some hope that the non-uniform classes might be amenable to treatment by the generalised topo-algebraic methods.

Indeed, the pseudo-variety of quasi-aperiodic quotient maps, or so-called quasi-aperiodic stamps, has been characterised in terms of profinite identities by Kunc [Kunc 2003]. Combining [Barrington et al. 1992] and [Kunc 2003] one gets

$$\mathrm{AC}^0 \cap \mathrm{Reg} = [\![\ (x^{\omega-1}y)^{\omega+1} = (x^{\omega-1}y)^{\omega} \text{ for } x, y \text{ words of the same length }]\!] \qquad (2)$$

where the ω power of an element in a compact semigroup is the unique idempotent in its cyclic closure. This result is useful because the equations can easily be checked in a finite monoid while being in $\mathrm{AC}^0 \cap \mathrm{Reg}$ is difficult to show otherwise (see Example 2.20 for an explanation of how this equational characterisation shows that $\mathrm{AC}^0 \cap \mathrm{Reg}$ is decidable). In addition, the algebraic properties of the quasi-aperiodic stamps are well understood as they are essentially obtained as Mal'cev products of aperiodic monoids by cyclic groups [Pin and Straubing 2005], with additional restrictions on the allowed recognising morphisms. On the logic side this gives the formula

$$\mathrm{FO}[\mathcal{N}] \cap \mathrm{Reg} = \mathrm{FO}[\mathrm{Reg}]. \qquad (3)$$

The situation is analogous for $\mathrm{ACC}^0 \cap \mathrm{Reg}$ and $(\mathrm{FO{+}MOD})[\mathrm{Reg}]{=}(\mathrm{FO{+}MOD})[<]$ and Straubing [Straubing 1994] has conjectured similar formulas for the classes ACC$_q$, for which only gates modulo a single q are allowed. However, in the case of ACC^0, the circuit complexity question whether ACC^0 is strictly contained in NC^1, is a long-open problem.

Now the idea how the generalised topo-algebraic theory might help separate complexity classes goes as follows: The hope is that one can generalise the tools of algebraic automata theory and thus be able to obtain equational characterisations of the logic fragments corresponding to Boolean circuit complexity classes. With these equations in hand, the goal is of course not decidability as in the regular setting but simply separation. Thus, the second hope is that one can show that *there exists* a language in the bigger fragment and *there exists* an equation which is not satisfied by that language but which is satisfied by all languages in the smaller fragment.

Now we discuss why we think these two goals might be achievable: Combining (1), (2), and (3), we have

$$\mathrm{FO}[\mathrm{Reg}] = [\![\ (x^{\omega-1}y)^{\omega+1} = (x^{\omega-1}y)^{\omega} \text{ for } x, y \text{ words of the same length }]\!]. \qquad (4)$$

That is, a certain logic fragment is given by certain equations. This is the kind of result where both sides of the equation are well within the natural scope of Stone duality, namely the dual space of a Boolean algebra corresponding to a certain logic fragment is given as a quotient of a bigger space (we will explain in Section 2.5 how equations relate to quotient spaces). What we are looking for is a result of the form

$$\mathrm{FO}[\mathcal{N}] = [\![\, \mathcal{E} \,]\!]$$

for some set $\mathcal{E}$ of appropriate equations, and the strategy for discovering and proving such a result should be the same as for (4). Note that while the proof of (2) depends on [Furst et al. 1984], language theorists can prove (4) directly.

To form the class $\mathrm{FO}[\mathrm{Reg}]$, one starts with the numerical predicates Reg and applies inductively layers of quantifiers (and Boolean closure). Obtaining a result such as (4) relies on an understanding of the effect on recognisers of closing under a layer of quantifiers and then on understanding how this affects the equational characterisation of the corresponding class. In the regular setting both of these steps are well understood. The effect on recognisers of closing under a layer of quantifiers is governed by various semidirect product constructions such as the block product (see [Tesson and Thérien 2007] for a nice introduction). The effect on, first the recognising space, and then the equations, of taking block products has been worked out in a number of special cases within the regular setting over the past 30 to 40 years and the papers [Almeida and Weil 1995] and [Almeida and Weil 1998], respectively, give general tools for solving these problems within the regular setting. Thus our task is to discover the appropriate generalisations of block product (and some results have been achieved in [Behle et al. 2011; Gehrke et al. 2016; Gehrke et al. 2017]), and then to generalise the results of [Almeida and Weil 1995] and [Almeida and Weil 1998] (at least in particular cases).

Even if all this works out, there is still the issue whether such equations can be used to separate the corresponding classes. The reason for optimism here is that, as mentioned above, one just needs to find *one* language and *one* equation that do the job (in Section 3, we give a 'proof of concept' example, and show in particular that PARITY does not satisfy one of the equations characterising the logic fragment treated there, see Example 3.11). We can now sum up what our goal is.

Research goal

We want to develop a topo-algebraic theory beyond regular languages with the ultimate goal of obtaining new tools and separation results for Boolean circuit classes. The first major goal in this direction is the challenge of giving a duality theoretic proof of an analogue of (4) *without intersecting with* Reg and using this result to give a new proof of the Furst, Saxe, and Sipser result.

Plan of the column

In Section 2 we provide an introduction to Stone duality tailored to the study of Boolean algebras of languages. After a historical overview of Stone duality and its applications in theoretical computer science, we start in Section 2.2 with the duality for finite Boolean algebras (and a slight generalisation of this non-topological duality). As examples, we treat

- Classical Propositional Logic (CPL) on a finite set of variables, showing how duality yields/is tantamount to the familiar truth-value semantics for CPL;
- The syntactic monoid of a regular language, which may be seen as the dual space of the Boolean algebra closed under algebraic quotient operations that it generates.

In Section 2.3 we introduce the full-fledged Stone duality for arbitrary Boolean algebras. We introduce the notion of syntactic space of a non-regular language, which is a so-called Boolean space with an internal monoid and use Stone duality to generalise the notion of recognition from regular languages to non-regular ones. As examples, we treat

- Classical Propositional Logic (CPL) on an infinite set of variables, showing how the Stone dual space is the well-known Cantor space;
- The syntactic space of the non-regular language MAJORITY;
- The dual space of the powerset $\mathcal{P}(A^*)$ of a finitely generated free monoid equipped with the biaction of A^* by quotienting operations. This so-called Boolean space with an internal monoid plays the same role for not-necessarily-regular languages as the free profinite monoid over A plays in the algebraic theory of regular languages. In particular, the equations used for separation of classes of languages are between elements of this space.

In Section 2.4 we give an alternative view of Boolean spaces as profinite sets. In the case of profinite monoids, this leads to the fact that one can study profinite monoids by means of systems of finite ones, such as pseudo-varieties of finite monoids. Here we generalise this notion, which leads to the notion of typed monoid and pseudo-varieties of typed stamps.

In Section 2.5, we consider Stone duality for subalgebras and quotient spaces and show how this leads both to the notion of profinite equations in algebraic automata theory and a similar notion of β-equations which is pertinent to the study of not-necessarily-regular languages. As examples we treat:

- The meaning and intuition behind the profinite equations for $\mathrm{AC}^0 \cap \mathrm{Reg}$ given in (2);
- A full proof of a simple equational characterisation of the languages recognised by the syntactic space of the language MAJORITY via its syntactic morphism relative to the monoid $(\mathbb{Z}, +)$, which is the (classical) syntactic monoid of this language.

Section 3 treats a first example of the theory in action, namely that of $FO_1[\mathcal{N}_0, \mathcal{N}_1^{uni}]$, which was the subject of [Gehrke et al. 2016]. However here we show how this example is a blueprint for the road to follow in treating more complex classes. Finally, Section 4 gives some perspectives on the future research we envisage as well as its goals.

2. STONE DUALITY

2.1. Historical background

Dualities between algebraic and topological structure are pervasive in mathematics, and toggling back and forth between algebraic and spacial reasoning has often been associated with important breakthroughs. Our objective here is to outline some ideas for how Stone duality may be applied in the theory of Boolean circuit complexity classes.

In 1936, M. H. Stone initiated duality theory in logic by presenting a dual category equivalence between the category of Boolean algebras and the category of compact Hausdorff spaces having a basis of clopen sets, so-called Boolean spaces [Stone 1936]. Stone's duality and its variants are central in making the link between syntactical and semantic approaches to logic. Also in theoretical computer science this link is central as the two sides correspond to specification languages and to spaces of computational states. The ability to translate faithfully between these two worlds has often proved itself to be a powerful theoretical tool as well as a handle for making practical problems decidable. A prime example is Abramsky's seminal work [Abramsky 1991] linking program logic and domain theory via Stone duality. Other examples include Esakia's duality [Bezhanishvili *(Ed.)* 2014] for Heyting algebras and the corresponding frame semantics for intuitionistic logic, Goldblatt's seminal work [Goldblatt 1989] identifying extended Stone duality as the setting for completeness issues for Kripke semantics in modal logic, and Ghilardi's work in modal and intuitionistic logic on unification [Ghilardi 2004] and normal forms [Ghilardi 1995]. Applications of Stone duality in logic and computer science generally need more than just basic Stone duality. For example, Abramsky's work needs Stone or Priestley duality for distributive lattices and applications in modal logic require a duality for Boolean algebras or distributive lattices endowed with additional operations. Dualities for additional operations originate with Jónsson and Tarski who studied duality in the form of canonical extensions. Duality for additional operations was later reformulated in purely duality theoretic terms in [Goldblatt 1989] in the setting of Priestley duality. Stone and Priestley duality for Boolean algebras and distributive lattices with various kinds of additional operations are often referred to as *extended* duality. In computer science applications in semantics, additional operations (or connectives) play an important role and extended duality has been extensively developed both in the form of canonical extension theory [Gehrke and Jónsson 2004; Gehrke et al. 2005], and in coalgebraic form [Venema 2006].

In contrast, Stone duality has not played a role in more algorithmic areas of theoretical computer science until recently. Profinite topology is a central tool in the algebraic theory of automata [Almeida 2005] and, as was observed as early as 1937 by Birkhoff, profinite topological algebras are based on Boolean spaces. However, the connection was not used until much more recently, first, in an isolated case by Pippenger [Pippenger 1997], and then more structurally starting in [Gehrke et al. 2008; Gehrke et al. 2010]. In Chapter 8 of their 2009 book, Rhodes and Steinberg introduced a bialgebraic and duality-theoretic approach to profinite semigroups [Rhodes and Steinberg 2009]. This point of view identifies deep connections with classical algebra.

2.2. Discrete duality

It is easiest to understand Stone duality by first understanding the non-topological part. At the finite level, there is no need for topology, and Stone duality is simply the fact that the category of finite Boolean algebras is equivalent to the *opposite* of the category of finite sets. In symbols

$$\mathsf{BA}_f \cong {\mathsf{Set}_f}^{op}.$$

Given a finite Boolean algebra B, the corresponding set is $At(B)$, the set of atoms of B. A non-zero element $a \in B$ is an *atom* provided, for any $b \in B$, either $a \wedge b = a$ or $a \wedge b = 0$. That is, on the induced order on the Boolean algebra, a is immediately above the bottom element. Conversely, given a finite set X, the powerset $\mathcal{P}(X)$ is a finite Boolean algebra. Further, going back and forth gives back an isomorphic copy of the original object. That is

$$B \cong \mathcal{P}(At(B)) \quad \text{and} \quad X \cong At(\mathcal{P}(X)).$$

The fact that BA_f is equivalent to the opposite of Set_f has to do with maps. Let A and B be finite Boolean algebras and X and Y their duals. Given a homomorphism $h\colon A \to B$ and a set map $f\colon Y \to X$, these are dual to each other provided

$$\forall a \in A\ \forall y \in Y \qquad y \leqslant h(a) \iff f(y) \leqslant a.$$

In particular, this means that, starting from a set map $f\colon Y \to X$, the dual homomorphism is the preimage map on the powersets, $f^{-1}\colon \mathcal{P}(X) \to \mathcal{P}(Y)$.

This non-topological duality does not work for all Boolean algebras, but it does work for all *complete and atomic* Boolean algebras (CABA). These are Boolean algebras in which all suprema and infima exist, and in which the atoms separate the elements. The same definitions as given above actually show that

$$\mathsf{CABA} \cong \mathsf{Set}^{op}.$$

We provide a few examples.

Example 2.1 (*Classical propositional logic on finitely many primitive propositions*). Let P be a finite set of primitive propositional variables. Then, up to logical equivalence, the propositional formulas in these variables form a Boolean algebra, CPL_P. The atoms of this Boolean algebra are all the complete conjunctions of literals. That is, they are in one-to-one correspondence with valuations, $v\colon P \to 2$, where p occurs positively if $v(p){=}1$ and negatively otherwise. That is, $At(CPL_P) \cong 2^P$. Disjunctive normal form tells us that each element of CPL_P can be written uniquely as a disjunction of complete conjunctions. That is, there is a one-to-one correspondence between the elements of CPL_P and the subsets of 2^P. This is simply a case of the Stone duality

$$CPL_P \cong \mathcal{P}(At(CPL_P)) \cong \mathcal{P}(2^P) \cong 2^{2^P}.$$

Example 2.2 (*The syntactic monoid of a regular language*). In this example we show how the syntactic monoid of a regular language may be obtained by discrete duality. The argument is given in a form which generalises to the non-regular setting, see [Gehrke et al. 2010; Gehrke et al. 2016] and especially [Gehrke et al. 2017], where this example is treated in the introduction.

Let L be a regular language over a finite alphabet A. Consider the Boolean subalgebra $\mathcal{B}(L)$ of $\mathcal{P}(A^*)$ generated by the quotients of L, i.e. by the sets

$$w^{-1}Lv^{-1} = \{u \in A^* \mid wuv \in L\}$$

for $w, v \in A^*$. Since the language L is regular, it is the set of all words recognised by some finite state automaton (Q, A, δ, I, F). Notice that the language $w^{-1}Lv^{-1}$ is recognised by (Q, A, δ, I_w, F_v), where I_w is obtained by moving forward along transitions from an initial state by any path labelled by w, while F_v is obtained by moving backwards along transitions from a final state by any path labelled by v. As a consequence, the seemingly infinite generating set for $\mathcal{B}(L)$ is actually finite, and so is the Boolean algebra $\mathcal{B}(L)$.

By the discrete duality, the embedding $\mathcal{B}(L) \hookrightarrow \mathcal{P}(A^*)$ is dual to a surjective map $A^* \twoheadrightarrow At(\mathcal{B}(L))$. That is, the subalgebra $\mathcal{B}(L)$ corresponds dually to an equivalence relation on $A^* \times A^*$ given by the equivalence classes

$$[u] = \bigcap_{wuv \in L} w^{-1}Lv^{-1} \cap \bigcap_{wuv \notin L} (w^{-1}Lv^{-1})^c$$

for $u \in A^*$, which are also the atoms of $\mathcal{B}(L)$. These are the equivalence classes of the Myhill *syntactic congruence* of L, $\sim_L$, and thus the elements of the *syntactic monoid* M_L of L.

However, this does not account for the monoid structure on M_L via duality, just for its set of elements. In order to explain the monoid structure via duality, notice first that the left quotient operation by a word, $S \mapsto w^{-1}S$, on $\mathcal{P}(A^*)$ is a homomorphism of complete and atomic Boolean algebras, which is dual to the left action of w on A^* given by left concatenation of w since

$$\forall S \in \mathcal{P}(A^*)\ \forall u \in A^* \qquad u \in w^{-1}S \quad \iff \quad wu \in S.$$

In diagrammatic form, we have the following duality of maps for each $w \in A^*$:

$$\begin{tikzcd}
\mathcal{P}(A^*) \arrow[r, "\Lambda_w"] & \mathcal{P}(A^*) & & A^* \arrow[r, "l_w"] & A^*. \\
S \arrow[r, mapsto] & w^{-1}S & \arrow[u, phantom, "|"] & v \arrow[r, mapsto] & wv
\end{tikzcd}$$

Also, the fact that

$$A^* \times A^* \to A^*, (w, u) \mapsto wu$$

is a left action of the monoid A^* on itself implies, by duality, that

$$A^* \times \mathcal{P}(A^*) \to \mathcal{P}(A^*), (w, S) \mapsto w^{-1}S$$

is a right action of the monoid A^* on the Boolean algebra $\mathcal{P}(A^*)$. Of course the same holds for the right quotients and the right action of A^* on itself, and the compatibility of the two actions on A^* (i.e. $l_w(r_v(u)) = wuv = r_v(l_w(u))$) yields the compatibility of the actions by quotienting on $\mathcal{P}(A^*)$. Now recall that the Boolean algebra $\mathcal{B}(L)$ is closed under the quotient operations, and thus, we have commuting squares as the left one in the following diagram

$$\begin{tikzcd}
\mathcal{B}(L) \arrow[r, hook] \arrow[d, dashed, "\Lambda_w"'] & \mathcal{P}(A^*) \arrow[d, "\Lambda_w"] \\
\mathcal{B}(L) \arrow[r, hook] & \mathcal{P}(A^*)
\end{tikzcd}$$

$$\begin{tikzcd}
A^* \arrow[r, two heads] \arrow[d, "l_w"'] & M_L \arrow[d, dashed, "\lambda_w"] \\
A^* \arrow[r, two heads] & M_L
\end{tikzcd}$$

Thus one obtains a left action of A^* on M_L. By a similar argument, the closure of $\mathcal{B}(L)$ under the right quotient operations yields a right action of A^* on M_L. And the compatibility of the two actions follows from the compatibility of the actions on $\mathcal{B}(L)$. Finally, the fact that the embedding $\mathcal{B}(L) \hookrightarrow \mathcal{P}(A^*)$ is a morphism for the biaction of A^* implies by duality that the quotient map $A^* \twoheadrightarrow M_L$ is a morphism for the biaction of A^*. One can show that this is equivalent to saying that M_L carries a monoid operation making the quotient map $A^* \twoheadrightarrow M_L$ a monoid morphism (one defines $x \cdot y = \lambda_w(y)$ where $x = [w]$).

2.3. Stone duality for Boolean algebras

All finite Boolean algebras are atomic as are some infinite ones, powersets for example. However, there are infinite Boolean algebras that do not have enough atoms to separate their elements. In fact there are atomless Boolean algebras (e.g., CPL_P for infinite P, see Example 2.5 below).

Thus in order to obtain representations of arbitrary Boolean algebras in powersets, we need to generalise the concept of atom. Just as ideals generalise divisors in rings, so ultrafilters generalise atoms in Boolean algebras.

Definition 2.3. A subset F of a Boolean algebra A is a *filter* provided

— F is an up-set, i.e., $a \in F$ and $a \leqslant b$ implies $b \in F$;
— F is non-empty, or equivalently, $1 \in F$;
— F is closed under finite meets i.e., $a, b \in F$ implies $a \wedge b \in F$.

A filter is said to be *proper* provided $F \neq A$, or equivalently, $0 \notin F$. Further, a proper filter F is an *ultrafilter* provided

— For all $a \in A$ either a or $\neg a$ belongs to F.

We denote by $Ult(A)$ the set of all ultrafilters of A.

Ultrafilters are in one-to-one correspondence with the set $\mathsf{BA}(A, 2)$ of Boolean algebra homomorphisms from A into the two-element Boolean algebra, 2.

PROPOSITION 2.4. *Let A be a Boolean algebra and $F \subseteq A$. The following conditions are equivalent:*

(*1*) *F is an ultrafilter;*
(*2*) *The characteristic function $\chi_F : A \to 2$ is a homomorphism of Boolean algebras.*

Example 2.5 (*Classical propositional logic on countably many propositional variables*). We claim that CPL_P for P an infinite set of primitive propositional variables has no atoms at all: Given any consistent propositional formula φ, and any primitive propositional variable p that does not occur in φ, the formula $\varphi \wedge p$ is still consistent but is not equivalent to φ. This shows that no formula φ can be an atom.

Let's explore then what the ultrafilters of this Boolean algebra, or equivalently, the homomorphisms from CPL_P into 2 are. Any function $v\colon P \to 2$ extends to a homomorphism $\tilde{v}\colon CLP_P \to 2$ (by structural induction, or equivalently, since CPL_P is the free Boolean algebra over P). Thus, as in the case of P finite, the set of ultrafilters corresponds to the set of all valuations $v\colon P \to 2$.

Further, we see easily that CPL_P can be embedded in 2^P: Any two formulas φ, ψ use only a finite subset P' of the propositional variables and they can be separated by a complete conjunction relative to P', and thus by a valuation $v\colon P' \to 2$. It follows that φ and ψ can be separated by any valuation $v'\colon P \to 2$ extending v.

Thus the map

$$CLP_P \to \mathcal{P}(2^P), \varphi \mapsto \widehat{\varphi} = \{v\colon P \to 2 \mid \left(\bigwedge_{\substack{p \in var(\varphi)\\ v(p)=1}} p\right) \wedge \left(\bigwedge_{\substack{p \in var(\varphi)\\ v(p)=0}} \neg p\right) \leqslant \varphi\},$$

where $var(\varphi)$ denotes the set of propositional variables that occur in φ, is an embedding of Boolean algebras. On the other hand, it is clear that this map cannot be surjective since the cardinality of CLP_P is equal to the cardinality of P, while even 2^P and thus certainly also $\mathcal{P}(2^P)$ have strictly larger cardinalities.

However, the sets

$$\widehat{p} = \{v \in 2^P \mid v(p) = 1\} \quad \text{and} \quad \widehat{\neg p} = \{v \in 2^P \mid v(p) = 0\}$$

are the subbasic open sets of the product topology on 2^P when we equip 2 with the discrete topology, and the set $\{\widehat{\varphi} \mid \varphi \in CPL_P\}$ is the closure of this subbasis under finite intersections and unions. It is therefore a basis for the product topology. The ensuing topological space is the well-known *Cantor space*, and it is not hard to see that the $\widehat{\varphi}$ are precisely the *clopen* (simultaneously closed and open) subsets of this space.

Definition 2.6. A *Boolean Stone space* (or just Boolean space) is a topological space that is compact Hausdorff and that has a basis of clopen subsets.[2] Given a Boolean space X we denote by $Clop(X)$ the Boolean algebra of clopen subsets of X equipped with the set-theoretic operations.
Given a Boolean algebra B, its *Stone dual space*, denoted $S(B)$, is the space of ultrafilters of B (or homomorphisms into 2) equipped with the topology generated by the sets

$$\widehat{a} = \{\mu \in Ult(B) \mid a \in \mu\} \cong \{h : B \to 2 \mid h(a) = 1\}$$

where a ranges over the elements of B.

THEOREM 2.7 (STONE DUALITY). *Let B be a Boolean algebra. Then $S(B)$ is a Boolean space and the map*

$$B \to \mathcal{P}(S(B)), \quad a \mapsto \widehat{a}$$

[2] The property of having a basis consisting of clopen subsets is also known as the property of being *zero dimensional*.

is an embedding of Boolean algebras whose image is the Boolean algebra $Clop(S(B))$. Further, if $h\colon A \to B$ is a homomorphism of Boolean algebras, then

$$S(h)\colon S(B) \to S(A),\ \mu \mapsto h^{-1}(\mu) \quad \textit{(or in terms of homomorphisms } (f\colon B \to 2) \mapsto (f \circ h : A \to 2))$$

is a continuous map with the property that $S(h)^{-1}(\widehat{a}) = \widehat{h(a)}$.

Let X be a Boolean space. Then $Clop(X)$ is a Boolean algebra and the map

$$X \to S(Clop(X)), x \mapsto \mu_x = \{U \in Clop(X) \mid x \in U\}$$

is a homeomorphism. Further, if $f\colon X \to Y$ is a continuous function between Boolean spaces, then

$$Clop(f)\colon Clop(Y) \to Clop(X),\ V \mapsto f^{-1}(V)$$

is a Boolean algebra homomorphism with the property that $Clop(f)^{-1}(\mu_x) = \mu_{f(x)}$.

In Example 2.2 we saw that the syntactic monoid of a regular language L may be seen as the dual space of the Boolean algebra $\mathcal{B}(L)$ equipped with the biaction of A^* via quotient operations. Now that we have dual spaces for infinite Boolean algebras as well, we can extend the concept.

Definition 2.8. Let A be a finite alphabet and $L \subseteq A^*$ any language over A. The *syntactic space* of L is the dual space of the Boolean algebra

$$\mathcal{B}(L) = \langle w^{-1}Lv^{-1} \mid w, v \in A^* \rangle_{\mathsf{BA}}$$

equipped with the biaction of A^* by quotient operations and the *syntactic morphism* is the dual of the embedding $\mathcal{B}(L) \hookrightarrow \mathcal{P}(A^*)$.

As we will see, the dual space of such a Boolean algebra with a biaction may be seen as what was called a Boolean space with an internal monoid in [Gehrke et al. 2016]. Before getting to that, we consider an example.

Example 2.9 (*The syntactic space of* MAJORITY). Let $A = \{a, b\}$. Consider the language

$$L = \{w \in A^* \mid w \text{ contains more } a\text{'s than } b\text{'s}\} = \text{MAJORITY}.$$

Consider the unique monoid homomorphism given by

$$\eta\colon A^* \twoheadrightarrow \mathbb{Z}, a \mapsto 1, b \mapsto -1.$$

It is not hard to see that, for all $u \in A^*$, we have that $u^{-1}L = Lu^{-1}$ and

$$L = \eta^{-1}(\mathbb{Z}^+) \quad \text{ and } \quad Lu^{-1} = \eta^{-1}(\mathbb{Z}^+ - \eta(u)).$$

Thus the Boolean algebra $\mathcal{B} = \mathcal{B}(L)$ is isomorphic (via the inverse of the mapping η^{-1}) to the Boolean subalgebra of $\mathcal{P}(\mathbb{Z})$ generated by the cosets under addition of $\mathbb{Z}^+ = \{n \in \mathbb{Z} \mid n > 0\}$. Notice also, that $Lu^{-1} = \eta^{-1}(\mathbb{Z}^+ - \eta(u))$ shows that the action of A^* on $\mathcal{B}$ factors through the action of $\mathbb{Z}$ on its powerset. Accordingly, we make the identification

$$\mathcal{B} = \langle \mathbb{Z}^+ - k \mid k \in \mathbb{Z} \rangle_{\mathsf{BA}}$$

and consider the (bi)action of $\mathbb{Z}$ on this Boolean algebra.

It is not difficult to see that $\mathcal{B}$ consists of all subsets of $\mathbb{Z}$ having finite or cofinite intersection with each of $\mathbb{Z}^+$ or $\mathbb{Z}^-$. We would like to compute the dual space of $\mathcal{B}$. The points $k \in \mathbb{Z}$ all give rise to points of the dual via

$$k \mapsto \{K \in \mathcal{B} \mid k \in K\}.$$

In this particular case, one can show that there are only two additional ultrafilters of $\mathcal{B}$, namely

$$+\infty = \{K \in \mathcal{B} \mid K \triangle \mathbb{Z}^+ \text{ is a finite set}\} \quad \text{and} \quad -\infty = \{K \in \mathcal{B} \mid K \triangle \mathbb{Z}^- \text{ is a finite set}\}.$$

Thus the dual space of $\mathcal{B}$ is based on the set $X = \mathbb{Z} \cup \{-\infty, +\infty\}$ and the sets $\widehat{K}$ for $K \in \mathcal{B}$ are:

(1) all finite subsets of $\mathbb{Z}$ and their complements in X;
(2) all subsets containing $+\infty$ and not containing $-\infty$ which have cofinite intersection with $\mathbb{Z}^+$ and finite intersection with $\mathbb{Z}^-$ — and their complements in X.

The topology of X is then obtained by closing under arbitrary unions. Thus the opens of X are the following sets:

(1) all subsets of $\mathbb{Z}$;
(2) all subsets containing $+\infty$ which contain all but finitely many of the elements of $\mathbb{Z}^+$;
(3) all subsets containing $-\infty$ which contain all but finitely many of the elements of $\mathbb{Z}^-$.

Using this, one may then verify that the sets $\widehat{K}$ for $K \in \mathcal{B}$ are exactly the clopens of X. Finally we note that $\mathbb{Z}$ is a dense subspace of X and that the components of the biaction of $\mathbb{Z}$ on itself extend continuously to X by setting $k + x = x + k = x$ for all $k \in \mathbb{Z}$ and $x \in \{+\infty, -\infty\}$.

We review what has happened in the above example. The language $L =$ MAJORITY is not regular and thus its syntactic monoid, $\mathbb{Z}$, is infinite. The syntactic morphism, $\eta\colon A^* \to \mathbb{Z}$, recognises uncountably many languages in $\mathcal{P}(A^*)$ (all the pre-images of subsets of $\mathbb{Z}$). However, $\mathcal{B}(L)$ is only a countable Boolean algebra of fairly well-behaved languages. By making a 'two-point compactification' X of $\mathbb{Z}$, we are able to embed $\mathcal{B}(L)$ in $\mathcal{P}(X)$ so that we can characterise the image in topological terms as the clopen subsets. The fact that $\mathbb{Z}$ is a monoid, is replaced by the fact that X contains a monoid as a dense subset and that this 'internal monoid' acts with continuous components on X (see Definition 2.11 below).

Example 2.10 (*The dual space of* $\mathcal{P}(A^*)$ *equipped with the quotienting biaction of* A^*). Let A be a finite alphabet. Since the Boolean algebras we are interested in are subalgebras of $\mathcal{P}(A^*)$, it is important to understand Stone duality in this case. By Definition 2.6, the dual space of $\mathcal{P}(A^*)$ is the space of all ultrafilters of $\mathcal{P}(A^*)$ equipped with the topology generated by the sets $\widehat{L} = \{\mu \in Ult(\mathcal{P}(A^*)) \mid L \in \mu\}$ for $L \in \mathcal{P}(A^*)$.

As in the previous example, there are some easy-to-access ultrafilters, namely the ones given by the elements of A^*: for each $u \in A^*$, we have a *principal ultrafilter*:

$$\mu_u = \{L \subseteq A^* \mid u \in L\}.$$

In general, for a powerset Boolean algebra, the only ultrafilters which are constructively available are the principal ones. However, a standard application of the Axiom of Choice, shows that *every proper filter of a Boolean algebra extends to an ultrafilter*. Alternatively, one can take this as an axiom (which is non-constructive but weaker than AC).

This very same space comes up in a different setting in topology, namely that of *Stone-Čech compactifications*. This is a free compactification. That is, given a space X (that is nice enough to have a compactification), its Stone-Čech compactification is an embedding of X into a compact Hausdorff space βX so that any continuous function $f\colon X \to Y$ into a compact Hausdorff space Y extends uniquely to a continuous function

$\beta f\colon \beta X \to Y$. It is then a fact that $\beta(A^*)$, where we consider A^* as a space with the discrete topology, is homeomorphic to the dual space of $\mathcal{P}(A^*)$. For this reason we denote this space by $\beta(A^*)$, and we often use its universal property. Note that the embedding $A^* \hookrightarrow \beta(A^*)$ required by the definition of the Stone-Čech compactification is given by the inclusion of A^* in its dual space as the principal ultrafilters, $u \mapsto \mu_u$.

One may observe that A^* sits densely in $\beta(A^*)$: If $\widehat{K}$ is a non-empty basic (cl)open, then $K \subseteq A^*$ must be non-empty (as all ultrafilters are proper) and thus there is a word $u \in K$. It follows that $K \in \mu_u$ and thus that $\mu_u \in \widehat{K}$. Also, the monoid A^* which sits as a dense subset of $\beta(A^*)$ has a biaction on the space with continuous components. To see this, let $u \in A^*$, then left concatenation by u may be seen as a map

$$A^* \to \beta(A^*), w \mapsto \mu_{uw}$$

and, for the discrete topology on A^*, it is continuous. Thus it has a unique extension

$$\lambda_u\colon \beta(A^*) \to \beta(A^*).$$

Exploiting the uniqueness, one may show that this is a left action of A^* on $\beta(A^*)$. Similarly we get a right action and one can show that the two are compatible. In the sequel, we will consider A^* as a subset of $\beta(A^*)$ and write things like $\lambda_u\colon \beta(A^*) \to \beta(A^*)$ is given by $w \mapsto uw$.

Definition 2.11 (*Boolean space with an internal monoid*). A *Boolean space with an internal monoid* is a pair (X, M) where X is a Boolean space and M is a dense subset of X and a monoid so that each left component $l_m\colon M \to M, m' \mapsto mm'$ and each right component $r_m\colon M \to M, m' \mapsto m'm$ has a continuous extension $\lambda_m\colon X \to X$ and $\rho_m\colon X \to X$, respectively.

A morphism from a Boolean space with an internal monoid (X, M) to another such, (Y, N), is a continuous map $f\colon X \to Y$, so that $f(m) \in N$ for each $m \in M$ and the ensuing map $f|\colon M \to N$ is a monoid morphism.

All finite monoids M are Boolean spaces with internal monoids when we take $X = M$ and *any* finite Boolean space with an internal monoid is of that form. As we have shown in Example 2.10, the pair $(\beta(A^*), A^*)$ is a Boolean space with an internal monoid. A topological version of the argument in Example 2.2 implies that the dual space of any Boolean subalgebra of $\mathcal{P}(A^*)$ closed under the quotienting action of A^* on $\mathcal{P}(A^*)$ is a Boolean space with internal monoid quotient of $(\beta(A^*), A^*)$, see [Gehrke et al. 2016, Section 3] for more details. We introduce some nomenclature and record the result here.

Definition 2.12 (*Recognition*). Let A be a finite alphabet and $L \subseteq A^*$. Recall that $\widehat{L} \subseteq \beta(A^*)$ is the clopen corresponding to L. A morphism of Boolean spaces with internal monoids $\phi\colon (\beta(A^*), A^*) \to (X, M)$ is said to *recognise* L provided there is a clopen $U \subseteq X$ so that $\phi^{-1}(U) = \widehat{L}$. Note that this is equivalent to $L = A^* \cap \phi^{-1}(U)$. Further we say that (X, M) recognises L provided there is a morphism of Boolean spaces with internal monoids $\phi\colon (\beta(A^*), A^*) \to (X, M)$ which does. When we restrict to finite Boolean spaces with internal monoids we recover the classical notion of recognition.

THEOREM 2.13. *Let A be a finite alphabet and $\mathcal{B}$ a Boolean subalgebra of $\mathcal{P}(A^*)$ closed under the quotienting action of A^* on $\mathcal{P}(A^*)$. Then the Stone dual of $\mathcal{B}$ with the action of A^* is a Boolean space with an internal monoid (X, M) and the Stone dual of the embedding $\mathcal{B} \hookrightarrow \mathcal{P}(A^*)$ is a surjective morphism of Boolean spaces with internal monoids $\phi\colon (\beta(A^*), A^*) \to (X, M)$ which recognises precisely the languages in $\mathcal{B}$. In particular, the syntactic space of any language $L \subseteq A^*$ is a Boolean space with an* *internal monoid which recognises L, and the syntactic morphism factors through any other morphism of Boolean spaces with internal monoids that recognises L.*

Conversely if $\phi\colon (\beta(A^*), A^*) \to (X, M)$ is any morphism of Boolean spaces with internal monoids, then the set of all languages over A recognised by ϕ forms a Boolean subalgebra of A^* closed under the quotienting action of A^* on $\mathcal{P}(A^*)$.

For the reader that is not familiar with the classical theory, we note that, in the special case of Theorem 2.13 where $\mathcal{B}$ consists entirely of regular languages, then the operation on the internal monoid M extends to a jointly (in both coordinates at once, i.e., in the product topology) continuous monoid operation on the dual space X, and this profinite monoid is the dual of $\mathcal{B}$. In fact, in [Gehrke et al. 2010] it was shown that this happens if and only if $\mathcal{B}$ consists entirely of regular languages.

Definition 2.12 and Theorem 2.13 tell us that the basic tools of algebraic automata theory, namely a rich enough class of recognisers with a nice universal property, exist also in the non-regular setting. This result was essentially the content of the paper [Gehrke et al. 2010] but the formulation given here is as in [Gehrke et al. 2016]. As we will see in Section 2.5, the equations that we want to use for characterising language classes come about as generators for the equivalence relation on $\beta(A^*)$ one has to mod out by to get the Boolean space with an internal monoid dual to the language class in question.

2.4. Boolean spaces as profinite sets: typed monoids

Boolean spaces are exactly the *profinite sets*. That is, if one has a system of finite sets $\{X_i\}_{i\in I}$ where I is a directed set (that is, $i, j \in I$ implies there exists $k \in I$ with $i \leqslant k$ and $j \leqslant k$) and a system of functions $\{f_{ji}\colon X_j \to X_i \mid i \leqslant j\}$ so that $f_{ki} = f_{ji} \circ f_{kj}$ whenever $i \leqslant j \leqslant k$, then the *limit* of this system is a Boolean space — and every Boolean space may be obtained in this way.

We elaborate a bit on this. Given a directed system $(\{X_i\}_{i\in I}, \{f_{ji}\}_{i\leqslant j})$, the inverse limit (denoted $\lim_{i\in I} X_i$), as a set, is the subset of all those sequences $\overline{x} \in \Pi_{i\in I} X_i$ so that $f_{ji}(x_j) = x_i$ whenever $i \leqslant j$. The restrictions of the projection maps $\pi_j\colon \lim_{i\in I} X_i \to X_j$ witness the limit in the sense that they commute with the connecting maps of the system, that is, $f_{ji} \circ \pi_j = \pi_i$ whenever $i \leqslant j$. The limit as a topological space (where we think of the finite sets as equipped with the discrete topology), is based on this same set and equipped with the topology generated by the basis

$$\{\pi_i^{-1}(S) \mid i \in I \text{ and } S \subseteq X_i\}$$

(note that this is also the usual subbasis for the product topology of $\Pi_{i\in I} X_i$). The directedness of I and the compositionality of the connecting maps is used to show that this is a basis and not just a subbasis. One can show that compact Hausdorff spaces are closed under inverse limits within topological spaces, and the profinite limits are zero dimensional since each $\pi_i^{-1}(S)$ is clopen since $\pi_i^{-1}(X_i \backslash S)$ is open as well. Thus profinite sets, that is, inverse limits of finite sets, are Boolean spaces. Conversely, given a Boolean space X and any partition $\mathcal{P} = \{U_1, \ldots, U_n\}$ consisting of clopens, the corresponding quotient map is a continuous map to a finite (Boolean) space. Since any two such partitions have a common refinement, which, again, consists of clopens, this system of finite continuous quotients of X, ordered by $\mathcal{P} \leqslant \mathcal{Q}$ provided $\mathcal{Q}$ refines $\mathcal{P}$, is an inverse limit system. Further, one can show that the limit of this system is (homeomorphic to) X.

This leads to a notion of finite recogniser, which was actually introduced independently of duality considerations in [Krebs et al. 2007] as typed monoids.

Definition 2.14. A *typed monoid* is a tuple (M, p, X), where X is a finite set, M is a monoid and $p\colon M \twoheadrightarrow X$ is a surjective set function. (Up to isomorphism) both the finite set X and the map p are determined by the partition $\mathcal{P} = \{p^{-1}(x) \mid x \in X\}$ induced by

p. Thus we will mainly work with typed monoids $(M, \mathcal{P})$, where M is a monoid and $\mathcal{P}$ is a finite partition of M. We say that a monoid morphism $\psi \colon A^* \to M$ into the monoid component of a typed monoid $(M, p, X) \cong (M, \mathcal{P})$ *recognises* a language $L \subseteq A^*$ if and only if there is a subset $S \subseteq X (\cong \mathcal{P})$ with $L = (p \circ \psi)^{-1}(S) = \psi^{-1}(\bigcup_{P \in S} P)$. As in the regular case, we say that (M, p, X) recognises $L \subseteq A^*$ provided there is a monoid morphism $\psi \colon A^* \to M$ which recognises L.

Example 2.15 (*The syntactic typed monoid for* MAJORITY). The notion of typed monoid may be seen as arising from separating the two features of a finite monoid as recogniser: namely recognition by a monoid and recognition by a finite set. Since both notions have a 'best' solution there is also a best typed monoid recognising a language, given simply by the partition L, L^c of the syntactic monoid $M_L = A^*/\sim_L$ of L. In the case of MAJORITY, this is the typed monoid $(\mathbb{Z}, \{\mathbb{Z}^+, \mathbb{Z}^- \cup \{0\}\})$. Via the syntactic morphism η, it just recognises the four languages in the Boolean algebra generated by L. In order to get the syntactic space of L we need to consider all the finite Boolean subalgebras of $\mathcal{B}(L)$, or at least an increasing chain of these. That is, if we have a directed collection $\{\mathcal{B}_i\}_{i \in I}$ of subalgebras of $\mathcal{B}(L)$ whose union is $\mathcal{B}(L)$, then $\{(\mathbb{Z}, At(\mathcal{B}_i))\}_{i \in I}$ with the connecting maps $At(\mathcal{B}_j) \twoheadrightarrow At(\mathcal{B}_i)$ dual to $\mathcal{B}_i \hookrightarrow \mathcal{B}_j$ whenever $\mathcal{B}_i \subseteq \mathcal{B}_j$ is an inverse limit system whose limit is the syntactic space of L.

As long as we are only interested in Boolean algebras of languages which are closed under the quotienting action of A^* on its powerset, the Boolean spaces with internal monoids suffice. However, since for non-regular languages, these are necessarily infinite spaces, the typed monoids, as finite approximants, allow an approach closer to the one that is familiar to automata theorists. We will see more on the use of typed monoids in Section 3.

2.5. Duality between subalgebras and quotients: Equations

The tool for separation results or in fact, in the regular setting, characterisation results are so-called equations. From the point of view of duality, these arise from the fact that the dual of a Boolean *sub*algebra is a *quotient* space, and thus given by *equating* elements of the bigger space.

That is, if $\mathcal{B}$ is a Boolean subalgebra of $\mathrm{Reg}(A^*)$, then the dual space X of $\mathcal{B}$ is a quotient of the dual of $\mathrm{Reg}(A^*)$. Since the dual of $\mathrm{Reg}(A^*)$ is the free profinite monoid, $\widehat{A^*}$, it follows that X is obtained as $\widehat{A^*}/\mathcal{E}$ where $\mathcal{E}$ is a set of pairs of profinite words, i.e., elements of $\widehat{A^*}$.

In the case of a Boolean subalgebra of $\mathcal{P}(A^*)$, not contained in the regular languages, the dual will not be a quotient of $\widehat{A^*}$ but of the dual of $\mathcal{P}(A^*)$, which is the Stone-Čech compactification $\beta(A^*)$. Accordingly, these Boolean algebras will be characterised by sets of pairs of ultrafilters $\mu \approx \nu$ with $\mu, \nu \in \beta(A^*)$.

This would not be all that helpful if we really needed to compute the entire equivalence relation $\mathcal{E}$. However, duality gives us a means of specifying $\mathcal{E}$ from a (possibly) much smaller set of pairs.

Definition 2.16. Let X be a Boolean space and $\mathcal{E} \subseteq X \times X$. We call $\mathcal{E}$ a *Boolean equivalence relation* provided the quotient space $X/\mathcal{E}$ is again a Boolean space. A quotient space of a Hausdorff space is again Hausdorff if and only if the equivalence relation is closed in the product topology. The compactness comes for free. In order for $X/\mathcal{E}$ to be zero dimensional, for any two non-equivalent points $x, y \in X$, there must be a clopen $U \subseteq X$ which separates them *and* which is saturated with respect to $\mathcal{E}$.

The last condition required for being a Boolean equivalence relation is in general difficult to achieve as it isn't prima facie a closure property. However, the following theorem shows that it is, making 'the least Boolean equivalence relation containing a set $E \subseteq X \times X$' well defined.

THEOREM 2.17 (STONE DUALITY FOR SUBALGEBRAS OF BOOLEAN ALGEBRAS). *Let B be a Boolean algebra, X the dual space of B. The assignments*

$$E \mapsto A_E = \{a \in B \mid \forall (x,y) \in E \; (x \in \widehat{a} \iff y \in \widehat{a})\}$$

for $E \subseteq X \times X$ and

$$S \mapsto \approx_S = \{(x,y) \in X \times X \mid \forall a \in S \; (x \in \widehat{a} \iff y \in \widehat{a})\}$$

for $S \subseteq B$ establish a Galois connection whose Galois closed sets are the Boolean equivalence relations and the Boolean subalgebras, respectively.

Definition 2.18. Let B be a Boolean algebra, X its dual space, and $x, y \in X$. An element $a \in B$ *satisfies* the equation $x \approx y$, and we write $a \vDash x \approx y$, provided

$$x \in \widehat{a} \iff y \in \widehat{a}.$$

COROLLARY 2.19. *Let B be a Boolean algebra, X the dual space of B. Every set of equations over X determines a Boolean subalgebra of B, and every Boolean subalgebra of B is given by a set of equations over X.*

Example 2.20 (*The meaning of the profinite equations for* $\mathrm{FO}[\mathcal{N}] \cap \mathrm{Reg}$). Given an element m of a finite semigroup or monoid M, consider the cyclic subsemigroup, $\langle m \rangle$, generated by m. If k and l are the least positive integers such that $m^k = m^{k+l}$ then it is not difficult to see that $\{m^k, m^{k+1}, \ldots, m^{k+l-1}\}$ is isomorphic to the group of integers modulo l. This subgroup of M has an identity element, which is then an idempotent element of M. In addition, it is not too hard to see that it is the *only* idempotent in $\langle m \rangle$. By a Cayley-type argument, one can show that, for any element $m \in M$, where M has n elements, this unique idempotent in the cyclic semigroup generated by m must be equal to $m^{n!}$. This argument generalises to compact topological semigroups: any element x of a compact topological semigroup has a unique idempotent in the closure of the cyclic semigroup generated by x and this element, which is denoted by x^ω may be obtained as $\lim_{n\to\infty} x^{n!}$. In particular, the sequence $\{x^{n!}\}$ is convergent.

Let us first look at the equation $y^\omega \approx y^{\omega+1}$. This is the equation for aperiodic monoids, that is, monoids that do not contain any non-trivial groups. This equation characterises $FO[<]$ and the corresponding languages are the star-free regular languages. This is the content of the seminal results of Schützenberger [Schützenberger 1965] and McNaughton and Papert [McNaughton and Papert 1971]. Now, one can show that a language satisfies the equation $y^\omega \approx y^{\omega+1}$ if and only if its syntactic monoid does, and the latter is clearly something one can check in a finite monoid and this is what makes membership in $FO[<]$ and star-freeness decidable.

Informally speaking, the equation $y^\omega \approx y^{\omega+1}$ says that if we repeat the word y often enough, it behaves the same as if we repeat it even one extra time. Formally, a regular language L satisfies the equation provided there exists an n such that $y^{n!} \approx y^{n!+1}$ holds. From this description we get that, in a fixed language L, we cannot count repetitions beyond some constant threshold. It also means that we cannot count modulo any number: Let $y = a$, then $y^{n!}$ will have an even number of a's, and $y^{n!+1}$ has an odd number of a's. So we cannot count modulo two. Since the equations satisfied by a pseudo-variety are closed under multiplication by words on both sides, we also have the equations $yy^{n!} \approx yy^{n!+1}$ which is the same as $y^{n!+1} \approx y^{n!+2}$, hence we get $y^{n!} \approx y^{n!+1} \approx y^{n!+2}$. This yields that we cannot count modulo three. Similarly this equation prevents us from being able to count modulo any number, or beyond any threshold as $y^{n!} \approx y^{n!+m}$ for any number $m \in \mathbb{N}$.

Now consider the equation $(z^{\omega-1}x)^\omega \approx (z^{\omega-1}x)^{\omega+1}$ from (2), which characterises the regular languages in AC^0. This is basically the equation $y^\omega \approx y^{\omega+1}$ where y is restricted to be of the form $z^{\omega-1}x$.

Given a stamp $\phi\colon A^* \to M$, where M is a finite monoid, one can show that ϕ satisfies the equation $(z^{\omega-1}x)^\omega \approx (z^{\omega-1}x)^{\omega+1}$ if and only if the so-called stable subsemigroup of ϕ is aperiodic. To obtain the stable subsemigroup of ϕ, we have to consider the powerset monoid, $\mathcal{P}(M)$. Notice that this is again a semigroup (even a monoid) with the pointwise product. The image $\phi[A]$ of the alphabet is an element of the finite semigroup $\mathcal{P}(M)$. As such it has a unique idempotent power, call it S. Since S is idempotent it is a subsemigroup of M. This is the stable subsemigroup of ϕ. Clearly, again, this property can be checked for a stamp $\phi\colon A^* \to M$.

While the equation $y^\omega \approx y^{\omega+1}$ holds in $FO[<]$, it does not hold in $\mathrm{FO}[\mathcal{N}] \cap \mathrm{Reg}$. For example we can distinguish $a^{n!}$ from $a^{n!+1}$ by the language $L = \{w \mid |w| \equiv 0 \mod 2\}$. This is because we have all nullary predicates in the logic class. Beyond that we can distinguish whether an a is in an even or in an odd position, so even the profinite terms $z^\omega x z^{\omega+1}$ and $z^{\omega+1}xz^\omega$ are not equated. This equation again holds in $FO[<]$, as can be seen by applying $z^\omega \approx z^{\omega+1}$. Here the x is at an even position in the first case and at an odd position in the second case.

We know by Furst, Saxe, Sipser that $\mathrm{FO}[\mathcal{N}] \cap \mathrm{Reg}$ cannot count modulo two. So in order to create profinite words that can be identified, we want to create two instances of words that have a different number of a's but the positions where the relevant a's appear are positions that behave the same, i.e., are the same number modulo any natural number and are larger than any constant. Additionally the two words we pick should have word lengths that behave the same.

So we want to pick two (or actually more) positions for the a's that are at the same position modulo any natural number. There will be no word $z \in A^*$ such that the positions of the a's in aza is zero modulo all natural numbers. But inside a profinite word we can use $a(\lim_{n\to\infty} b^{n!-1})a$ and the two positions of a have the desired property. Now we want these positions to be larger than any constant so we can pick $(\lim_{n\to\infty} b^{n!})a(\lim_{n\to\infty} b^{n!-1})a$. Whether we pick $b^{n!}$ or $b^{n!-1}$ as a prefix does not matter as this only ensures that both positions of the a's are larger than any constant.

More generally we get $(z^{\omega-1}x)^\omega \approx (z^{\omega-1}x)^{\omega+1}$, where z, x need to have the same word length. So this equation says that we cannot count modulo or beyond a threshold inside an undistinguishable equivalence class of positions (assuming both words have the same length, i.e., both are larger than any constant and have the same length modulo any natural number).

The previous example discusses the meaning and consequences of an equational characterisation, but it does not provide a proof (which is much too substantial to discuss here). In the following example, we prove that a certain characterisation by equations holds in a very simple case in order to illustrate the technical content of such a statement. A more substantial but still relatively simple case is treated in Section 3.

Example 2.21 (*Equations for* MAJORITY). We will show that, relative to $\beta(\mathbb{Z})$, the dual space X of $\mathcal{B}(L)$, where L is the language MAJORITY, is characterised by the equations

$$\{\mu + 1 \approx \mu \mid \mu \in \beta(\mathbb{Z})\backslash\mathbb{Z}\}.$$

Before proving this, we make a few comments. First note that $\beta(\mathbb{Z})$ is an uncountable space. As a topological space, it is homeomorphic to $\beta(\mathbb{N})$, being the Stone-Čech compactification of a countable discrete space. However, as a Boolean space with an internal monoid it is of course different as the internal monoids and thus also the actions are different. The action of $\mathbb{Z}$ on $\beta(\mathbb{Z})$ is given by

$$S \in \lambda_k(\mu) \iff S - k \in \mu$$

and we will write $\mu + k$ for $\lambda_k(\mu)$ (this looks like a right action but since $\mathbb{Z}$ is commutative the right and left actions agree). Also, recall that the syntactic space of the language MAJORITY is the 'two point compactification' $X = \mathbb{Z} \cup \{-\infty, +\infty\}$ as described in Example 2.9. It is not difficult to see that the quotient map is:

$$\beta(\mathbb{Z}) \to X, \mu \mapsto \begin{cases} k & \textit{provided } \{k\} \in \mu \\ +\infty & \textit{provided } \mathbb{Z}^+ \in \mu \in \beta(\mathbb{Z})\backslash\mathbb{Z} \\ -\infty & \textit{provided } \mathbb{Z}^- \in \mu \in \beta(\mathbb{Z})\backslash\mathbb{Z} \end{cases}$$

These three conditions are exhaustive as the first accounts for all the principal ultrafilters. The rest are the ones in the so-called remainder $\beta(\mathbb{Z})\backslash\mathbb{Z}$, also referred to as *free ultrafilters*. Such a filter cannot contain any finite subset of $\mathbb{Z}$. Finally, since $\mathbb{Z} = \mathbb{Z}^- \cup \{0\} \cup \mathbb{Z}^+$ it follows that, in any free ultrafilter we must have (exactly one of) $\mathbb{Z}^+ \in \mu$ or $\mathbb{Z}^- \in \mu$.

Thus we see that, in the equivalence relation $\mathcal{E} \subseteq \beta(\mathbb{Z}) \times \beta(\mathbb{Z})$ corresponding to X, uncountably many free ultrafilters are in the equivalence class corresponding to each of $+\infty$ and $-\infty$.

Now we prove that the 'few' equations given above suffice to characterise the quotient space X (there are uncountably many equations in the set but only a few ultrafilters are stipulated to be equated with any one ultrafilter).

We want to show that, for $K \subseteq \mathbb{Z}$ and $\mu \in \beta(\mathbb{Z})\backslash\mathbb{Z}$ we have

$$(K \in \mu \iff K \in \mu + 1) \qquad \iff \qquad K \in \mathcal{B}(L)$$

where L is MAJORITY. Since $K \in \mu+1$ if and only if $K-1 \in \mu$, it follows that K satisfies the equation $\mu \approx \mu + 1$ if and only if $K \in \mu$ is equivalent to $K - 1 \in \mu$. Further, it is an easy verification, see e.g. [Gehrke et al. 2016, Proposition 1.8.], that for any ultrafilter μ of a Boolean algebra and any two elements b and b' of the algebra, we have

$$(b \in \mu \iff b' \in \mu) \qquad \iff \qquad b \triangle b' \notin \mu.$$

Thus we need to prove, for $K \subseteq \mathbb{Z}$ and $\mu \in \beta(\mathbb{Z})\backslash\mathbb{Z}$,

$$K \triangle (K - 1) \notin \mu \qquad \iff \qquad K \in \mathcal{B}(L).$$

First notice that for each $k \in \mathbb{Z}$ we have $\mathbb{Z}^+{-}k \subseteq \mathbb{Z}^+{-}k{-}1$ and therefore the symmetric difference is $(\mathbb{Z}^+{-}k)\triangle(\mathbb{Z}^+{-}k{-}1) = (\mathbb{Z}^+{-}k{-}1) - (\mathbb{Z}^+{-}k) = \{-k\}$. Thus for any free ultrafilter μ, we have that $(\mathbb{Z}^+{-}k)\triangle(\mathbb{Z}^+{-}k{-}1) \notin \mu$. We have shown that each generator of $\mathcal{B}(L)$ satisfies the equation $\mu + 1 \approx \mu$, and therefore it follows by Theorem 2.17 that every element of $\mathcal{B}(L)$ satisfies the equation. This proves the soundness of the equations.

For the converse, suppose $K \notin \mathcal{B}(L)$. Then one of $K \cap \mathbb{Z}^+$ and $K \cap \mathbb{Z}^-$ is neither finite nor cofinite. It follows that there exists an infinite set $S \subseteq \mathbb{Z}$ so that $S \subseteq K$ but $(S{+}1) \cap K = \emptyset$. Since S is infinite, there is a free ultrafilter μ of $\mathbb{Z}$ which contains S (to see this, note that $\{T \subseteq \mathbb{Z} \mid T \cap S \text{ is cofinite}\}$ is a proper filter of $\mathcal{P}(\mathbb{Z})$ and thus extends to an ultrafilter. Finally, this ultrafilter cannot contain any singleton set and must thus be free). Since $(S{+}1) \cap K = \emptyset$, for each $m \in S$, $m + 1 \notin K$, or equivalently, $m \notin K - 1$. That is, $S \cap (K{-}1) = \emptyset$. Thus $S \subseteq K \triangle (K{-}1)$ and thus $K \triangle (K{-}1) \in \mu$.

3. A SIMPLE APPLICATION

In this section we will examine the logic class $FO_1[\mathcal{N}_0, \mathcal{N}_1^{uni}]$. This class consists of languages of words $w = w_1 \dots w_k$ which, viewed as relational structures on initial segments $\{1,\ldots,k\}$ of $\mathbb{N}$, satisfy first-order sentences of quantifier depth one using letter predicates, arbitrary non-uniform nullary predicates, i.e., all numerical predicates P that depend only on the length of the word, and arbitrary uniform unary predicates, i.e., all numerical predicates $P(x)$ that accept certain positions x independent of the the specific word or its length, see [Gehrke et al. 2016, Section 2.2] for a detailed description of the connection between the logic fragment and the Boolean algebra of languages.

Both the nullary and the unary numerical predicates in question can be represented by subsets of the natural numbers: we identify a non-uniform nullary predicate P with the subset

$$\{n \in \mathbb{N} \mid w \models P \text{ for some (and then for all) } w \text{ with length } n\}.$$

We can identify a uniform unary predicate $P(x)$ with the subset

$$\{n \in \mathbb{N} \mid w_{x=n} \models P(x) \text{ for some (and then for all) } w \text{ with length } \geqslant n\}$$

where $w_{x=n}$ denotes the model obtained by adding to w an interpretation under which x is sent to the n^{th} position of w. It is an easy verification that the Boolean algebra $FO_1[\mathcal{N}_0,\mathcal{N}_1^{uni}]$ is generated by the languages $\mathrm{Mod}(P)$ and $\mathrm{Mod}(\forall x(\mathbf{a}(x) \implies P(x)))$, i.e., the languages

$$L_P = \{w \in A^* \mid \text{length}(w) \in P\} \quad \text{and} \quad L_{a,P} = \{w \in A^* \mid w_i{=}a \;\Rightarrow\; i \in P\},$$

where P ranges over the subsets of $\mathbb{N}$ and $a \in A$. It is quite simple to analyse this logic class by combinatorial methods, so this is a good class to pick as an example to demonstrate the duality approach. It is also the first layer of the inductive construction of $FO[\mathcal{N}] = \mathrm{AC}^0$. The material in this section is a slightly reformulated presentation of part of the content of [Gehrke et al. 2016]. This reformulation is closer to the formulation of the result in [Czarnetzki and Krebs 2016], which proves a result that is a generalisation of [Gehrke et al. 2016].

Just as $FO[\mathcal{N}] \cap \mathrm{Reg}$ is an lp-variety of stamps rather than of finite monoids, so $FO_1[\mathcal{N}_0,\mathcal{N}_1^{uni}]$ is an lp-variety of 'typed stamps', that is, pairs consisting of a typed monoid $(M,\mathcal{P})$ and a surjective monoid morphism $\phi\colon A^* \twoheadrightarrow M$. We denote such a *typed stamp* by $(A^*,\phi,M,\mathcal{P})$.

Remark 3.1. In [Behle et al. 2011; Behle et al. 2013] the notion of a typed monoid was extended to objects of the form $(M,\mathcal{B},\mathcal{U})$, where M is a monoid and $\mathcal{B}$ is a finite Boolean algebra of subsets of M, and U is a finite subset of M. The correspondence between the two notions is provided by the discrete duality

$$\iota\colon \mathcal{B} \hookrightarrow \mathcal{P}(M) \qquad \leftrightsquigarrow \qquad \iota^{-1}\colon M \twoheadrightarrow At(\mathcal{B}).$$

That is, the typed stamp in our sense corresponding to $(M,\mathcal{B},U)$ is $(U^*,\phi,M,At(\mathcal{B}))$, where ϕ is the monoid morphism extending the identity map, and conversely, given a typed stamp (A^*,ϕ,M,p,X), the triple $(M,\mathcal{B}(X),\phi[A])$ where

$$\mathcal{B}(X) = \{p^{-1}(P) \mid P \subseteq X\}$$

is a typed monoid in the sense of [Behle et al. 2011].

The lp in lp-variety stands for length preserving. A homomorphism $\phi\colon A^* \to B^*$ between free finitely generated monoids is said to be *length preserving* provided it sends letters to letters. That is, it is the (unique) extension of a map $A \to B$ rather than just of a map $A \to B^*$.

Definition 3.2 (*lp-variety*). An lp-variety of languages is an assignment, for each finite alphabet A, of a set $\mathcal{V}(A)$ of languages over A so that the following two properties are satisfied:

(1) For each finite alphabet A, $\mathcal{V}(A)$ is a Boolean subalgebra of $\mathcal{P}(A^*)$ which is closed under the quotienting biaction of A^* on $\mathcal{P}(A^*)$;
(2) If $\phi\colon A^* \to B^*$ is a length preserving homomorphism between free finitely generated monoids and $L \in \mathcal{V}(B)$, then $\phi^{-1}(L) \in \mathcal{V}(A)$. That is, ϕ^{-1} restricts to a Boolean algebra homomorphism $\phi^{-1}\colon \mathcal{V}(B) \to \mathcal{V}(A)$.

We will not give the definition of an lp-variety of typed stamps, but, as in the classical setting, it is simply the class of all typed stamps making up the (fullest possible) inverse limit systems (as discussed in Section 2.4) of the Boolean spaces with internal monoids dual to the $\mathcal{V}(A)$'s.

3.1. The variety $\underline{\mathbb{N}}$

In what follows $\mathbb{N}$ is the usual additive monoid of natural numbers. Note that it is also isomorphic to A^* when A has only one element and for this reason we will write $\mathbb{N}$ instead of A^* in this case.

Definition 3.3 ($\underline{\mathbb{N}}$). Let $\underline{\mathbb{N}}$ be the smallest pseudo-variety of typed stamps containing $(\mathbb{N}, \mathrm{id}, \mathbb{N}, \{P, P^c\})$ for all subsets $P \subseteq \mathbb{N}$ where $\mathrm{id}\colon \mathbb{N} \to \mathbb{N}$ is the identity isomorphism.

The definition of $\underline{\mathbb{N}}$ gives us a set of languages in $\mathbb{N}$, that is, in A^* where A is the one element alphabet. Also, since $(\mathbb{N}, \mathrm{id}, \mathbb{N}, \{P, P^c\})$ recognises the languages $P, P^c, \emptyset$, and $\mathbb{N}$, and we take these typed monoids for every $P \in \mathcal{P}(\mathbb{N})$, the set of languages recognised by $\underline{\mathbb{N}}$ in $\mathbb{N}$ itself will simply be the full powerset $\mathcal{P}(\mathbb{N})$. Now, for any other alphabet A, there is exactly one length preserving morphism $\ell\colon A^* \to \mathbb{N}$, namely the one that sends each letter to 1. That is, for each $w \in A^*$, we have $\ell(w) = \text{length}(w)$. Thus we see that $L_P = \{w \in A^* \mid \ell(w) \in P\}$ must be among the languages recognised by $\underline{\mathbb{N}}$. Also, it is not difficult to see that we do not obtain anything more. Thus we have:

LEMMA 3.4. *The languages recognised by $\underline{\mathbb{N}}$ are exactly the languages $L_P = \{w \in A^* \mid \ell(w) \in P\}$ for some subset $P \subseteq \mathbb{N}$.*

Hence it seems that $\underline{\mathbb{N}}$ is tailored to recognise exactly the non-uniform nullary predicates of our logic. But even more is true, when used in the right-hand side of the wreath product, the typed monoid will be applied to a prefix of a word, and will hence accept exactly a certain set of positions, like the uniform unary predicates do.

3.2. The variety $\mathbf{J}_1 * \underline{\mathbb{N}}$

We extend the notion of a wreath product from finite monoids to typed stamps. Given two monoids N and M, we define $N * M$ to be the monoid based on the set $N^M \times M$ with the multiplication defined by $(f_1, m_1) \cdot (f_2, m_2) = (f, m_1 m_2)$ where $f(m) = f_1(m) f_2(m m_1)$ for all $m \in M$.

Definition 3.5 (*Wreath product*). Given two typed stamps $(A^*, \phi, M, \mathcal{P})$ and $((A \times \mathcal{P})^*, \psi, N, \mathcal{Q})$, we define the wreath product $((A \times \mathcal{P})^*, \psi, N, \mathcal{Q}) * (A^*, \phi, M, \mathcal{P})$ to be the typed stamp

$$(A^*, \theta, S, \mathcal{R}),$$

where

(1) $S \subseteq N * M$ is the monoid generated by the elements of the form $(f_a, \phi(a))$ for $a \in A$, where $f_a(m) = \psi(a, [m])$ and $[m]$ is the unique element of $\mathcal{P}$ containing m;
(2) $\theta\colon A^* \to S$ is defined by $\theta(a) = (f_a, \phi(a))$;

(3) $\mathcal{R} = \{\{(f,m) \mid f(1) \in Q, m \in P\} \mid Q \in \mathcal{Q}, P \in \mathcal{P}\}$.

As in the case of finite monoids we can apply the wreath product to varieties in the following sense. Here $\mathbf{J}_1$ denotes the pseudo-variety of all finite semi-lattices.

Definition 3.6 ($\mathbf{J}_1 * \underline{\mathbb{N}}$). Let $\mathbf{J_1} * \underline{\mathbb{N}}$ be the smallest lp-variety of stamps containing the typed stamps obtained by taking wreath products of stamps in $\mathbf{J}_1$ with stamps in $\underline{\mathbb{N}}$.

Finally we are able to characterise the logic class of our example in terms of typed stamps.

LEMMA 3.7. *A language is in* $\mathrm{FO}_1[\mathcal{N}_0, \mathcal{N}_1^{uni}]$ *iff it is recognised by* $\mathbf{J}_1 * \underline{\mathbb{N}}$.

PROOF. One can show that the typed stamps of the form $((A \times \mathcal{P})^*, \psi, U_1, \{\{0\},\{1\}\}) * (A^*, \phi, \mathbb{N}, \mathcal{P})$, where U_1 is the monoid on $\{0,1\}$ with minimum and $(A^*, \phi, \mathbb{N}, \mathcal{P}) \in \underline{\mathbb{N}}$, generate the variety. Thus it is sufficient to show that these can be defined in $\mathrm{FO}_1[\mathcal{N}_0, \mathcal{N}_1^{uni}]$.

Let $(A^*, \theta, S, \mathcal{R}) = ((A \times \mathcal{P})^*, \psi, U_1, \{\{0\},\{1\}\}) * (A^*, \phi, \mathbb{N}, \mathcal{P})$. Since $S \subset U_1 * \mathbb{N}$, we will denote the elements of S as pairs $(f,m) \in U_1^{\mathbb{N}} \times \mathbb{N}$. We will write the monoid $\mathbb{N}$ additively, hence the multiplication is $+$ and the neutral element is 0, whereas we denote the multiplication in U_1 by $\cdot$ and the neutral element by 1.

The elements of $\mathcal{R}$ are of the form $\{(f,m) \mid f(0) = v, m \in P\}$ for $P \in \mathcal{P}$ and $v \in \{0,1\}$. For each $P \in \mathcal{P}$ and $v \in \{0,1\}$, we give a formula that defines exactly the language $L_{P,v} = \theta^{-1}(\{(f,m) \mid f(0) = v, m \in P\})$.

Given a word $w \in A^*$ we can compute $\theta(w) = (f_w, m_w)$. By definition $m_w = |w|$, and $f_w(0) = f_{w_1}(0) \cdot f_{w_2}(1) \cdot f_{w_3}(2) \cdot \ldots \cdot f_{w_{|w|}}(|w|-1)$. Hence $f_w(0) = 0$ iff there is an i such that $f_{w_i}(i-1) = 0$.

Hence the language $L_{P,0}$ is defined by the formula $|w| \in P \wedge \bigvee_{(a',P') \in J} \exists x \;\; P'(x) \wedge a'(x)$, where $J = \{(a',P') \in A \times \mathcal{P} \mid \forall m \in P' : f_a(m) = 0\}$. Similarly we can define $L_{P,1}$ by $|w| \in P \wedge \bigvee_{(a',P') \in J} \neg\exists x \;\; P'(x) \wedge a'(x)$, where $J = \{(a',P') \in A \times \mathcal{P} \mid \forall m \in P' : f_a(m) = 0\}$.

This proves one direction. For the other direction we use the characterisation of the logic class in [Gehrke et al. 2016] where it was shown that the languages defined in this logic class are Boolean combinations of languages $L_Q = \{w \mid |w| \in Q\}$ and $L_{a,Q} = \{w \mid \forall j \;\; a(j) \implies j \in Q\}$ for $Q \subseteq \mathbb{N}$.

A language of the first type is recognised by the typed stamp $(A^*, \phi, \mathbb{N}, \{Q, \mathbb{N} \setminus Q\})$ as $\phi^{-1}(Q)$.

A language of the second type is a bit more complicated. It is recognised by the typed stamp $(A^*, \theta, U_1 * \mathbb{N}, \mathcal{R}) = ((A \times \mathcal{P})^*, \psi, U_1, \{\{0\},\{1\}\}) * (A^*, \phi, \mathbb{N}, \{Q, \mathbb{N} \setminus Q\})$, where $f_a(j) = 1$ if $j \in Q$ and $f_a(j) = 0$ otherwise, $f_b(j) = 1$ for all $a \neq b \in A, j \in \mathbb{N}$ as the $\theta^{-1}(\{(f,m) \mid f(0) = 0\})$. Computing the value of $f_w(0)$, where $(f_w, m_w) = \theta(w)$, will show that $L_{a,Q}$ is the language that is recognised in this way. □

3.3. Guided guess of the equations

As explained in the introduction, classical algebraic automata theory is a mature and sophisticated theory with very powerful tools and results developed over the past 40 to 50 years. The idea then for the non-regular setting is to let us be guided by corresponding classical results. For pseudo-varieties $\mathbf{V}$ of finite monoids, the results in [Almeida and Weil 1998] yield the following equations for $\mathbf{J}_1 * \mathbf{V}$:

— $xyz \approx xzy$ for all $x, y, z \in \widehat{A^*}$ if $x \approx xy \approx xz$ are equations for $\mathbf{V}$;
— $xy \approx xy^2$ for all $x, y \in \widehat{A^*}$ if $x \approx xy$ is an equation for $\mathbf{V}$.

If a Boolean algebra closed under quotients satisfies $x \approx y$, then it satisfies $xz \approx yz$ for all $z \in A^*$. In the regular case, this extends to $xz \approx yz$ for all $z \in \widehat{A^*}$. This is because if prefixes behave equivalently for all words up to a fixed, but arbitrarily large, size, then by using the pumping lemma for regular languages this extends to all profinite words. However, in the non-regular case we do not have such a pumping lemma available. So instead of allowing any pair x, y in the first equation we want to swap only one letter. In fact, for regular languages recognised by $\mathbf{J}_1 * \underline{\mathbb{N}}$, the equations

$$xayb \approx xbya \tag{5}$$

where $x \in \widehat{A^*} \backslash A^*$ and $\mathrm{length}(x) = \mathrm{length}(xay)$ suffice for the first batch of equations.

Thus we want a non-regular version of the equations (5). In $\beta(A^*)$, we do not have a continuous monoid operation, but we can get around this problem. To this end, let $a, b \in A$ and consider the following functions:

$$\begin{tikzcd} A^* \times A^* \arrow[rr, shift left, "\ell_1"] \arrow[rr, shift right, "\ell_2"'] \arrow[d, shift right, "f_{ab}"'] \arrow[d, shift left, "f_{ba}"] & & \mathbb{N} \\ A^* & & \end{tikzcd}$$

where

$$f_{ab}(u, v) = uavb \quad \text{and} \quad f_{ba}(u, v) = ubva$$

$$\ell_1(u, v) = \mathrm{length}(u) \quad \text{and} \quad \ell_2(u, v) = \mathrm{length}(uav).$$

Then [Gehrke et al. 2016, Theorem 3.2] states (in a slightly different but equivalent form) that

THEOREM 3.8. *If a language $L \in \mathcal{P}(A^*)$ is recognised by $\mathbf{J}_1 * \underline{\mathbb{N}}$ then it satisfies the equations*

$$\beta f_{ab}(\gamma) \approx \beta f_{ba}(\gamma)$$

for all $a, b \in A$ and all $\gamma \in \beta(A^* \times A^*)$ such that $\beta\ell_1(\gamma) = \beta\ell_2(\gamma)$.

Similarly, one may show that, for the regular languages recognised by $\mathbf{J}_1 * \underline{\mathbb{N}}$, the equations in the second batch, of the form $xy \approx xy^2$, are equivalent to equations of the form $x^2y \approx xy^2$, and then again to the equations $xayazb \approx xaybzb$ where $x \in \widehat{A^*} \backslash A^*$ and $\mathrm{length}(x) = \mathrm{length}(xay) = \mathrm{length}(xayaz)$. Here again we obtain corresponding β-equations by considering, for any two letters $a, b \in A$ the functions

$$g_{aab} \colon (A^*)^3 \to A^*,\ (u, v, w) \mapsto uavawb$$

$$g_{abb} \colon (A^*)^3 \to A^*,\ (u, v, w) \mapsto uavbwb$$

and $k_i \colon (A^*)^3 \to \mathbb{N}$, $i = 1, 2, 3$, given by

$$k_1(u, v, w) = \mathrm{length}(u)\,, \quad k_2(u, v, w) = \mathrm{length}(uv){+}1 \quad \text{and} \quad k_3(u, v, w) = \mathrm{length}(uvw){+}2.$$

One can then prove, c.f. [Gehrke et al. 2016, Theorem 3.3]

THEOREM 3.9. *If a language $L \in \mathcal{P}(A^*)$ is recognised by $\mathbf{J}_1 * \underline{\mathbb{N}}$ then it satisfies the equations*

$$\beta g_{aab}(\gamma) \approx \beta g_{abb}(\gamma)$$

for all $a, b \in A$ and all $\gamma \in \beta((A^*)^3)$ such that $\beta k_1(\gamma) = \beta k_2(\gamma) = \beta k_3(\gamma)$.

And in fact the completeness result of [Almeida and Weil 1998] lifts in this case to the not-necessarily-regular setting as was proved in [Gehrke et al. 2016, Theorem 4.7]:

THEOREM 3.10. *A language $L \in \mathcal{P}(A^*)$ is recognised by $\mathbf{J}_1 * \underline{\mathbb{N}}$ if and only if it satisfies the equations*

$$\beta f_{ab}(\gamma) \approx \beta f_{ba}(\gamma) \quad \textit{and} \quad \beta g_{aab}(\mu) \approx \beta g_{abb}(\mu)$$

for all $a, b \in A$, all $\gamma \in \beta((A^*)^2)$ such that $\beta\ell_1(\gamma) = \beta\ell_2(\gamma)$, and all $\mu \in \beta((A^*)^3)$ such that $\beta k_1(\mu) = \beta k_2(\mu) = \beta k_3(\mu)$.

Just like the corresponding equations for the regular languages, these equations give us clear intuitive information about the expressive power of the pseudo-variety $\mathbf{J}_1 * \underline{\mathbb{N}}$. To this end, one should think of $\beta f_{ab}(\gamma)$ as a *generalised word of the form $uavb$* and of $\beta f_{ba}(\gamma)$ as a *generalised word of the form $ubva$*, and similarly for $\beta g_{aab}(\mu)$ and $\beta g_{abb}(\mu)$. Then we may interpret the two batches of equations as follows:

— $\mathbf{J}_1 * \underline{\mathbb{N}}$ cannot distinguish the order of letters in equivalent positions;
— $\mathbf{J}_1 * \underline{\mathbb{N}}$ cannot count the number of occurrences of a letter in equivalent positions.

For the second, note that we really mean 'cannot count beyond 1', since $\mathbf{J}_1 * \underline{\mathbb{N}}$ *can* tell which letters occur in a set of equivalent positions (e.g. $\beta f_{ab}(\gamma) \approx \beta f_{aa}(\gamma)$ is not a valid equation for $\mathbf{J}_1 * \underline{\mathbb{N}}$). That is, it can tell the difference between 0 and at least 1 occurrence of a letter, but no more.

Example 3.11 (PARITY *does not satisfy one of the equations for $FO_1[\mathcal{N}_0, \mathcal{N}_1^{uni}]$*). $FO_1[\mathcal{N}_0, \mathcal{N}_1^{uni}]$ is a very small fragment of $FO[\mathcal{N}] = \mathrm{AC}^0$, so by [Furst et al. 1984] we know that PARITY does not belong to $FO_1[\mathcal{N}_0, \mathcal{N}_1^{uni}]$. Here we show this directly by finding an equation among the ones given in Theorem 3.10 which does not hold of PARITY. To this end, note that PARITY is a commutative language, so it definitely satisfies all the equations of the first batch. The second batch, which says that we cannot count beyond 'at least one' seems contradictory to PARITY. We show that it is.

Let L denote the language PARITY and set

$$T = \{(a^i, a^j, b^k) \mid i, j, k \in \mathbb{N}\} \subseteq (A^*)^3.$$

We pick this subset T because it will be easier to control the preimage of L on T than it is on $(A^*)^3$ but T is still infinite and thus 'big enough' to be in an ultrafilter $\mu \in \beta((A^*)^3)$ satisfying $\beta k_1(\mu) = \beta k_2(\mu) = \beta k_3(\mu)$. Indeed, this is what we prove first.

To this end, let α be any free ultrafilter of $\beta(\mathbb{N})$ and consider the set

$$\mathcal{G} = \{S \cap \bigcap_{j=1}^{3} k_j^{-1}(P) \mid P \in \alpha\}.$$

We show that the elements of $\mathcal{G} \subseteq \mathcal{P}((A^*)^3)$ are all non-empty, thus, as $\mathcal{G}$ is closed under finite intersections, it will follow that there is an ultrafilter $\mu \in \beta((A^*)^3)$ that extends $\mathcal{G}$. Let $P \in \alpha$. Since α is a free ultrafilter, $P \subseteq \mathbb{N}$ is infinite. Let $n \in P$. Since P is infinite, there exists $n' \in P$ with $n + 1 < n'$. Similarly, there is $n'' \in P$ with $n + n' + 2 < n''$. Now it follows that $s = (a^n, a^{n'-n-1}, b^{n''-n'-n-2}) \in S$ and $k_1(s) = n \in P$, $k_2(s) = n' \in P$, $k_3(s) = n'' \in P$. That is, $s \in S \cap \bigcap_{j=1}^{3} k_j^{-1}(P)$. Accordingly, pick an ultrafilter $\mu \in \beta((A^*)^3)$ that extends $\mathcal{G}$. We claim that $\beta k_j(\mu) = \alpha$ for $j = 1, 2$, and 3. To see this, let $P \in \alpha$ then by construction $k_j^{-1}(P) \in \mu$ and therefore $P \in \beta k_j(\mu)$. Thus $\alpha \subseteq \beta k_j(\mu)$, but distinct ultrafilters are non-comparable, so $\alpha = \beta k_j(\mu)$ as stipulated. We claim that

$$\text{PARITY} \not\models \beta g_{aab}(\mu) \approx \beta g_{abb}(\mu).$$

This is substantiated by the following string of equivalences

$$\begin{aligned} g_{aab}^{-1}(L) \in \mu &\iff S \cap g_{aab}^{-1}(L) \in \mu \\ &\iff \{(a^i, a^j, b^k) \mid i + j + 2 \text{ is odd}\} \in \mu \\ &\iff \{(a^i, a^j, b^k) \mid i + j + 2 \text{ is even}\} \notin \mu \\ &\iff \{(a^i, a^j, b^k) \mid i + j + 1 \text{ is odd}\} \notin \mu \\ &\iff S \cap g_{abb}^{-1}(L) \notin \mu \\ &\iff g_{abb}^{-1}(L) \notin \mu. \end{aligned}$$

This last example thus gives an example of how one may use equations to separate language classes (in this case $FO_1[\mathcal{N}_0, \mathcal{N}_1^{uni}]$ and $\mathrm{FO} + \mathrm{MOD}[\mathcal{N}]$).

4. FUTURE GOALS

Our aim is to understand the computational power of AC^0. While we have a clear understanding for some languages, there still are very simple languages, 'just outside' the regular language class, for which it is not known whether or not they belong to AC^0. For example, for the linear context free language described by the grammar $S \to acSb|aScb|\epsilon$, we do not know of any method for deciding whether it is in AC^0 or not.

On the way to reach this goal other very challenging problems show up. What are the regular languages in linear AC^0, where linear AC^0 has the same power as $\mathrm{FO}_2[\mathcal{N}]$?

Even more challenging are some open problems, that do not seem so hard at a first glance, yet they have resisted for a long time. So it would be interesting to understand what makes these problems hard:

Consider a circuit family where all gates are modulo 6 gates, i.e., the gate is true provided the number of inputs evaluating to true is 0 modulo 6. We limit this family to depth 3 and polynomial size. It is open whether 1^* can be computed by this class. So the class seems to be rather weak, yet it is not known how to separate this class from NP. Hence we want to compute equations for this class, and even if we do not get all equations there is some hope for obtaining equations that rule out at least one NP-hard problem.

All these problems have in common that their hardness stems from the fact that they have powerful numerical predicates. So a different class to look at would be $\mathrm{MAJ}[<]$, which is equivalent to $\mathrm{FO}[<]-$uniform TC^0. There is a strong belief that this class cannot compute all regular languages, but this also remains open.

A simpler class to look at is $\mathrm{MAJ}_2[<]$ which is known not to contain all regular languages, but still it would be a worth-while goal to find equations for this class.

In order to understand all these classes better we would like to borrow techniques from the world of regular languages such as the block product principle, the derived category theorem, and many more. It is a promising direction to try to lift all these principles beyond regular languages and apply them.

5. ACKNOWLEDGEMENTS

We would like to thank Neil Immerman for the opportunity to write this column as well as for his enthusiastic support and his very helpful comments on an early draft. Further, we would like to acknowledge the important role in formulating and developing the ideas laid out in this column of our colleagues and students working with us on this project: Jean-Éric Pin, Howard Straubing, Célia Borlido, Silke Czarnetzki, Daniela Petrişan, and Luca Reggio.

REFERENCES

Samson Abramsky. 1991. Domain theory in logical form. *Annals of Pure and Applied Logic* 51, (1-2) (1991), 1–77.

Jorge Almeida. 2005. Profinite semigroups and applications. In *Structural theory of automata, semigroups, and universal algebra*. Springer, 1–45.

Jorge Almeida and Pascal Weil. 1995. Free profinite semigroups over semidirect products. *Izvestiya VUZ Matematika* 39, 1–3 (1995), 3–31. English version, Russian Mathem. (Iz. VUZ.) 39 (1995) 1–28.

Jorge Almeida and Pascal Weil. 1998. Profinite categories and semidirect products. *J. Pure Appl. Algebra* 123, 1-3 (1998), 1–50.

David A. Mix Barrington, Kevin Compton, Howard Straubing, and Denis Thérien. 1992. Regular languages in NC^1. *J. Comput. System Sci.* 44, 3 (1992), 478–499.

Christoph Behle, Andreas Krebs, and Mark Mercer. 2013. Linear circuits, two-variable logic and weakly blocked monoids. *Theor. Comput. Sci.* 501 (2013), 20–33. DOI:http://dx.doi.org/10.1016/j.tcs.2013.07.005

Christoph Behle, Andreas Krebs, and Stephanie Reifferscheid. 2011. Typed Monoids - An Eilenberg-Like Theorem for Non Regular Languages. In *Algebraic Informatics - 4th International Conference, CAI 2011, Linz, Austria, June 21-24, 2011. Proceedings*. Springer, 97–114. DOI:http://dx.doi.org/10.1007/978-3-642-21493-6_6

Christoph Behle and Klaus-Jrn Lange. 2006. FO[¡]-Uniformity. In *21st Annual IEEE Conference on Computational Complexity (CCC 2006)*. IEEE Computer Society, 183–189.

Guram Bezhanishvili *(Ed.)*. 2014. *Leo Esakia on duality in modal and intuitionistic logic*. Outstanding contributions to logic, Vol. 4. Springer. 334 pages.

Silke Czarnetzki and Andreas Krebs. 2016. Using Duality in Circuit Complexity. In *Language and Automata Theory and Applications - 10th International Conference, LATA 2016, Prague, Czech Republic, March 14-18, 2016, Proceedings*. 283–294. DOI:http://dx.doi.org/10.1007/978-3-319-30000-9_22

Samuel Eilenberg. 1976. *Automata, languages, and machines*. Vol. B. Academic Press.

Merrick Furst, James B. Saxe, and Michael Sipser. 1984. Parity, circuits, and the polynomial hierarchy. *Mathematical Systems Theory* 17 (1984), 13–27.

Mai Gehrke, Serge Grigorieff, and Jean-Éric Pin. 2008. Duality and Equational Theory of Regular Languages. In *ICALP 2008, Part II*, Aceto et al. (Ed.). Springer, Berlin Heidelberg, 246–257. DOI:http://dx.doi.org/10.1007/978-3-540-70583-3_21

Mai Gehrke, Serge Grigorieff, and Jean-Éric Pin. 2010. A Topological Approach to Recognition. In *ICALP 2010, Part II*. Springer-Verlag, Berlin, Heidelberg, 151–162. http://dl.acm.org/citation.cfm?id=1880999.1881016

Mai Gehrke and Bjarni Jónsson. 2004. Bounded distributive lattices expansions. *Math. Scand.* 94, 2 (2004), 13–45.

Mai Gehrke, Andreas Krebs, and Jean-Éric Pin. 2016. Ultrafilters on Words for a Fragment of Logic. *Theor. Comput. Sci.* 610, PA (Jan. 2016), 37–58. DOI:http://dx.doi.org/10.1016/j.tcs.2015.08.007

Mai Gehrke, Hideo Nagahashi, and Yde Venema. 2005. A Sahlqvist Theorem for distributive modal logics. *Annals of Pure and Applied Logic* 131, 1–3 (2005), 65–102.

Mai Gehrke, Daniela Petrişan, and Luca Reggio. 2016. The Schützenberger Product for Syntactic Spaces. In *ICALP 2016*. Schloss Dagstuhl, 112:1–112:14. DOI:http://dx.doi.org/10.4230/LIPIcs.ICALP.2016.112

Mai Gehrke, Daniela Petrişan, and Luca Reggio. 2017. Quantifiers on languages and codensity monads. (2017). https://arxiv.org/abs/1702.08841 To appear in LICS.

Silvio Ghilardi. 1995. An Algebraic Theory of Normal Forms. *Annals of Pure and Applied Logic* 71, 3 (1995), 189–245.

Silvio Ghilardi. 2004. Unification, finite duality and projectivity in varieties of Heyting algebras. *Annals of Pure and Applied Logic* 127, 1-3 (2004), 99–115.

Robert Goldblatt. 1989. Varieties of complex algebras. *Annals of Pure and Applied Logic* 44 (1989), 173–242.

Neil Immerman. 1999. *Descriptive Complexity*. Springer-Verlag, New York.

Andreas Krebs, Klaus-Jörn Lange, and Stephanie Reifferscheid. 2007. Characterizing TC^0 in Terms of Infinite Groups. *Theory Comput. Syst.* 40, 4 (2007), 303–325. DOI:http://dx.doi.org/10.1007/s00224-006-1310-2

Michal Kunc. 2003. Equational description of pseudovarieties of homomorphisms. *Theoret. Informatics Appl.* 37 (2003), 243–254.

Robert McNaughton and Seymour Papert. 1971. *Counter-free Automata*. MIT Press.

Jean-Éric Pin. 2009. Profinite Methods in Automata Theory. In *STACS 2009*, Albers and Marion (Eds.), Vol. 3. Schloss Dagstuhl, 31–50. http://drops.dagstuhl.de/opus/volltexte/2009/1856

Jean-Éric Pin and Howard Straubing. 2005. Some results on $\mathcal{C}$-varieties. *Theoret. Informatics Appl.* 39 (2005), 239–262.

Nicholas Pippenger. 1997. Regular Languages and Stone Duality. *Theory Comput. Syst.* 30, 2 (1997), 121–134.

Jan Reiterman. 1982. The Birkhoff theorem for finite algebras. *Algebra Universalis* 14, 1 (1982), 1–10.

John Rhodes and Benjamin Steinberg. 2009. *The q-theory of finite semigroups*. Springer Verlag. 666 pages.

Marcel-Paul Schützenberger. 1965. On finite monoids having only trivial subgroups. *Inform. Control* 8 (1965), 190–194.

Marshall H. Stone. 1936. The theory of representations for Boolean algebras. *TAMS* 40 (1936), 37–111.

Howard Straubing. 1994. *Finite Automata, Formal Logic, and Circuit Complexity*. Birkhauser.

Pascal Tesson and Denis Thérien. 2007. Logic meets algebra: the case of regular languages. *Log. Methods Comput. Sci.* 3, 1 (2007), 1:4, 37.

Yde Venema. 2006. Algebras and coalgebras. In *Handbook of Modal Logic*, Blackburn et al (Ed.). Elsevier, 331–426.

Pascal Weil. 2002. Profinite methods in semigroup theory. *Int. J. Alg. Comput.* 12 (2002), 137–178.

Ryan Williams. 2014. Nonuniform ACC Circuit Lower Bounds. *J. ACM* 61, 1, Article 2 (Jan. 2014), 32 pages. DOI:http://dx.doi.org/10.1145/2559903

VERIFICATION COLUMN

NEHA RUNGTA, Amazon Web Services, Inc.
rungta@amazon.com

The growing popularity of verification competitions such as VerifyThis Verification Competition and the Competition on Software Verification (SV-COMP), both, incentivize as well as reward tool developers. VerifyThis 2016 was held in Eindhoven colocated with the European Joint Conferences on Theory and Practice of Software (ETAPS) 2016. The competition presents challenges to evaluate a broad range of tools such as Danfy, KeY, VeriFast, CIVL, among others. The first article in the newsletter "CIVL Solutions to VerifyThis 2016 Challenges" by Stephen F. Siegel at University of Delaware presents the solutions to the challenges using The Concurrency Intermediate Verification Language (CIVL) verifier. CIVL supports of a variety of techniques based on symbolic execution and model checking to verify sequential and concurrent programs. The article presents an overview of the framework, a description of the challenges, and how CIVL performs on the benchmarks. The second article "Security by Compilation: An Automated Approach to Comprehensive Side-channel resistance" by Chao Wang from University of Southern California and Patrick Schaumont from Virginia Tech leverage formal verification and program synthesis to detect side challenge attacks on mobile devices. The approach is evaluated on implementations of MAC-Keccak, AES and other cryptographic algorithms. This article bridges the security and verification forums of the newsletter.

I sincerely thank all the contributing authors Stephen Siegel, Chao Wang, and Patrick Schaumont for sharing their work with the SIGLOG and the larger logic, formal methods, and verification community.

CIVL Solutions to VerifyThis 2016 Challenges

Stephen F. Siegel, University of Delaware

The VerifyThis 2016 program verification competition was held in April, 2016. The competition presented three verification challenges: the first dealt with matrix multiplication, including the correctness of Strassen's algorithm; the second dealt with Morris' tree traversal algorithm, and the third required verification of a multithreaded tree barrier. In this paper I present solutions using the CIVL (Concurrency Intermediate Verification Language) verifier. CIVL is able to verify each program within small but non-trivial bounds. The solutions are relatively simple, and are presented in full.

1. INTRODUCTION

This paper presents solutions to the VerifyThis 2016 challenges using the CIVL verifier. The fifth event in an annual series, the competition took place on April 2, 2016, as part of the European Joint Conferences on Theory and Practice of Software (ETAPS). The competition proper took place in one day, and consisted of three challenges, each lasting 90 minutes. Each challenge consisted of a natural language description of a problem, usually with some pseudocode, and a description of properties expected to hold. Working in teams of one or two members, participants attempted to implement, specify, and formally verify the system described, using any languages and verification frameworks of their choosing. The definition of *formal verification* is broad and the competition included deductive verification approaches as well as bounded model checking. The organizers evaluated each solution for correctness, completeness, and elegance. The 14 teams used 9 different tools: CIVL, Dafny, Why3, KIV, KeY, VerCors, Viper, mCRL2, and VeriFast. A second day was used to present and discuss solutions. The complete challenge statements and other information are available on the competition web site [Huisman et al. 2016a]. A post-competition report summarizes the results [Huisman et al. 2016b].

After the competition, participants and others could submit completed and cleaned up versions of their solutions to a public Wiki accessible from the competition web page. The solutions shown here are the revised versions. In each case, I point out any significant differences from my original solutions.

CIVL. The CIVL verifier [Siegel et al. 2015; CIVL 2017] uses symbolic execution and model checking techniques to verify sequential or parallel C programs. The parallel programs may use MPI [Message-Passing Interface Forum 2015], OpenMP [OpenMP 2017], CUDA [NVIDIA 2017], or Pthreads [IEEE 2004], or even combinations of those APIs. The framework is built around an intermediate language called CIVL-C. CIVL-C is an extension of (sequential) C which includes a number of primitives facilitating specification as well as basic concurrency primitives. The CIVL framework consumes a C program using those concurrency dialects and transforms it into a pure CIVL-C program which can be consumed by the verifier. Users can also write in CIVL-C directly, and this was the approach I took in the competition.

The CIVL verifier checks a number of safety properties, including absence of assertion violations, deadlocks, division by 0, illegal pointer dereferences, out-of-bound array indexes, memory leaks, improper uses of malloc/free, and reads of uninitialized memory. The assertion language is richer than regular C—e.g., it includes first-order quantifiers and predicates for deep equality of objects.

Typically, CIVL is used to verify properties within some bounded region of the input space. Bounds are usually placed on the size of input data structures, on parameters that control the number of loop iterations, and on the number of processes or threads. Within these bounds, properties are verified exhaustively: for all possible values of the inputs, for all thread interleavings, for all choices available to the concurrency APIs (such as wildcard matchings in MPI), and so on.

The effectiveness of this verification approach relies on the *small scope hypothesis*—the belief that defects in a parameterized system almost always manifest themselves in some instance of the system with small parameter values. For example, if a program to multiply matrices behaves correctly on all matrices with size $n \leq 10$, most people would take this as strong evidence that the program behaves correctly for all n. Of course, it is *possible* to construct a solution which fails only for $n = 11$—but this tends not to occur "in nature."[1]

All other things being equal, bounded verification is not as good as approaches based on deductive reasoning which can prove claims without bounds. However, deductive approaches generally require significantly more effort and skill on the part of the user. I hope this paper adds to our understanding of this tradeoff. In particular, I believe the solutions to be sufficiently simple that a programmer of moderate skill, with some additional training, could use CIVL in a similar way.

Repeatability. All of the solutions, original and revised, as well as a Makefile and the CIVL output, can be downloaded from http://vsl.cis.udel.edu/verifythis2016. CIVL is open source software released under the GNU Public Licence. The results reported here were obtained using CIVL 1.7, the same version used in the competition, and which is available at http://vsl.cis.udel.edu/lib/sw/civl/1.7/latest/release/.[2] The CIVL verifier is a Java program and should run on any Java 8 virtual machine. Its only dependencies are three automated theorem provers, which should be installed and in the user's `PATH` the first time CIVL is run. The results reported here were obtained using Z3 [de Moura and Bjørner 2008] version 4.5.1, CVC4 [Barrett et al. 2011] version 1.4, and CVC3 [Barrett and Tinelli 2007] version 2.4.1, run on a 2014 MacBook Air with a 1.7 GHz Intel Core i7 CPU and 8GB RAM, the same laptop used in the competition.

2. OVERVIEW OF CIVL

2.1. Architecture

The CIVL framework comprises four components:

(1) SARL: the Symbolic Algebra and Reasoning Library, used to create and manipulate symbolic expressions, and to prove the validity of formulas. SARL uses external automated provers such as Z3 and CVC4;
(2) ABC: the ANTLR-based C front end, used to parse and represent C programs, including those using OpenMP, CUDA-C, and CIVL-C; produces a CIVL-C Abstract Syntax Tree;
(3) GMC: Generic Model Checking framework, implements standard model checking algorithms, such as depth-first search of a state-transition system; and

[1] There is an implicit assumption in the small scope hypothesis that "magic numbers" such as "11" do not occur in the program code. Such numbers should be replaced by parameters.

[2] The SHA1 checksum of `civl-1.7_3157.jar` is `c1e7d31663b8588459c9275ff81bbdf4ace160fc`.

(4) CIVL proper, which uses the above 3 components. Translates a CIVL-C AST into a lower-level model which defines a state-transition system in which states map program variables to symbolic expressions. Uses GMC to perform reachability analysis of that system, and SARL to check assertions.

2.2. SARL and Symbolic arithmetic

SARL provides services usually associated with two different fields: symbolic algebra and automated theorem proving. The language supported by SARL is essentially a typed ("many-sorted") first-order logic. The types include the real numbers, integers, booleans, and characters, as well as tuple, array, and function types. The terms in this logic are *symbolic expressions* and formulas are symbolic expressions of boolean type. Variables are *symbolic constants*. The function and predicate symbols are *symbolic operators*. There are methods to build expressions, to ask if a formula is valid, to get various kinds of information about an expression, to perform substitution, to simplify an expression or compute an interval approximation of a numeric expression under a given context, and so on. SARL solves many validity queries itself through its simplification process, but if that process does not suffice it invokes a sequence of external provers.

Additional features of SARL include the following:

- There are *complete* and *incomplete* array types. Both specify an element type E, but a complete type additionally specifies a length l, which is a symbolic expression of integer type. The domain of the complete type consists of all sequences of E of length l. The incomplete type's domain consists of all finite sequences of E; it is a supertype of every complete type with element type E. This differs from other systems in which arrays are infinite. SARL's approach is convenient for representing arrays in languages such as C.
- Operators that are commutative and associative (including addition, multiplication, *logical and*, and *logical or*) consume a finite sequence of arguments of any length.
- Two additional types are the *Herbrand integers* and *Herbrand reals*. These are like the usual integers and reals, but numeric operations involving them are treated as uninterpreted functions. These could be used for "bit-precise" reasoning, though currently CIVL does not use them.
- SARL also supports operations on bit vectors, which are represented as arrays of booleans.

Like most computer algebra systems, SARL attempts to transform symbolic expressions into a standard form[3] and to simplify expressions. There are multiple reasons for doing so: (1) a standard form means two equivalent expressions are likely to be represented in the same way, which speeds up equality testing, (2) in SARL, the Flyweight pattern is used on symbolic expressions, so using a standard form reduces the total number of expressions and therefore the memory footprint, (3) simpler expressions tend to be smaller, which further decreases the memory footprint, and (4) the speed and precision of most algorithms consuming symbolic expressions decreases with the size and complexity of the expressions, so simpler expressions generally make these algorithms more effective.

In the SARL semantic model, operations do not have to be total, i.e., there can exist values on which a symbolic expression is *undefined*. We write $t_1 \rightsquigarrow t_2$ if t_2 evaluates to the same value as t_1 in every valuation for which t_1 is defined. For example, if X is a

[3] I use *standard form* instead of *canonical form*, because the latter technically means a unique representative of an equivalence class. Transforming expressions to a true canonical form is often prohibitively expensive and not even possible for certain classes of expressions; cf. [Caviness 1970].

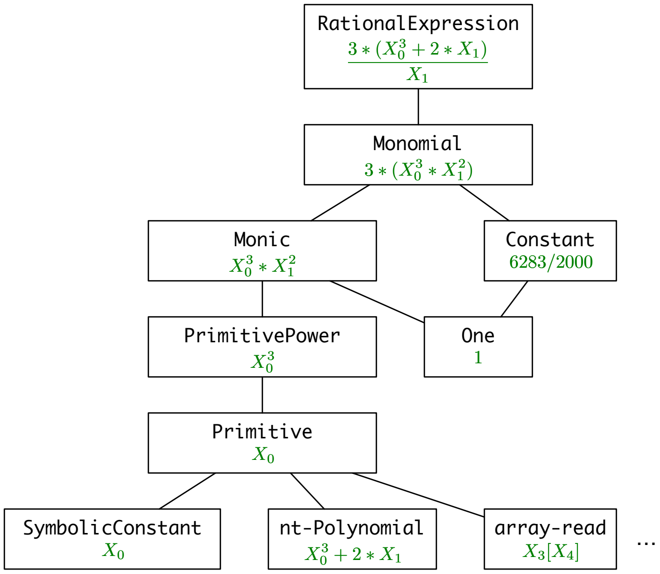


Fig. 1. Classes of expressions used to define the standard form for symbolic expressions of real type, with examples. An edge indicates that the bottom class is a sub-class of the class above.

symbolic constant of real type, $X/X \rightsquigarrow 1$, but $1 \not\rightsquigarrow X/X$, as X/X is not defined at 0. For most applications, t_1 can be safely replaced by t_2.

Every SARL method that returns a symbolic expression guarantees that expression will be in *standard form*. The form requires a total order on symbolic expressions: the symbolic constants and operators are ordered, and these extend to a total order on all expressions. For example, the standard form for boolean expressions is conjunctive normal form, in which the operands of the *and* and *or* operators occur in increasing order.

For expressions of real or integer type, the situation is more complicated. The specification of the standard real form, for example, requires defining several classes of expressions. The class hierarchy is depicted in Figure 1 and the definitions are as follows:

— The real *primitives* are a class of expressions that play the role of "variables" in polynomial forms. Most non-constant real expressions in which the operator is not addition, multiplication, subtraction, or division are primitives. This includes real symbolic constants, array reads for which the element type is real, function applications for which the return type is real, tuple selections for which the selected component is real, and power expressions in which the exponent is not a concrete integer. There is another type of real primitive, the *nt-polynomial*, defined below.
— A *real constant* is a concrete rational number. These are represented as quotients of (unbounded) integers with greatest common divisor 1, and for which the denominator is positive.
— A *primitive power* is a primitive or a power expression in which the base is a primitive and the exponent is a concrete positive integer greater than or equal to 2.
— A *monic* is the constant 1, a primitive power, or a product of two or more primitive powers with distinct primitives. The factors of a monic are ordered by their primitives.

— A *monomial* is a constant, a monic, or the product of a non-0 constant and a monic that is not one.
— An *nt-polynomial* (“non-trivial” polynomial) is a sum of two or more monomials, all of which have distinct monics. The terms occur in order of increasing monic and the leading coefficient is 1. Furthermore, there is no primitive which occurs as a factor in every term.

A monomial is *not* an nt-polynomial. Instead, all nt-polynomials are primitives, so can be used as “variables” in more complicated monics. The reason for this approach, as opposed to the more natural approach of declaring monomials to be polynomials, is that *expansion* is an expensive operation. Rather than requiring every operation to expand polynomials, expansion can be carried out only when necessary. For example, $(X_0 + X_1)^{100}$ is in standard form: it is a primitive power in which the primitive base is the nt-polynomial $X_0 + X_1$.

Finally, the standard form for any real expression is the *rational expression*:

— A *rational expression* is a monomial, or the quotient of a monomial and a monic which is not 1. The monic of the numerator and the denominator can have no primitive factor in common.

The following are the standard forms for numeric comparison operators:

— $0 < m$
— $0 \leq m$
— $m < 0$
— $m \leq 0$
— $0 = p$
— $0 \neq p$

where p is a primitive and m is a monic in which all factors are primitives (i.e., the maximum power of each factor is 1).

Standard properties of the real numbers allow every real or comparison expression t_1 to be transformed to a standard form t_2 such that $t_1 \rightsquigarrow t_2$. I omit the details, but give a few examples:

(1) $2 * X_0 + X_1$ is not in standard form. It is not an nt-polynomial because the leading coefficient is not 1. Its standard form is $2 * (X_0 + (1/2) * X_1)$, which is a monomial in which the constant is 2 and the monic is the nt-polynomial $X_0 + (1/2) * X_1$.
(2) $\dfrac{X_0 + X_1}{2 * (X_0 + X_2)}$ is not in standard form because the denominator is not a monic. Its standard form is $\dfrac{(1/2) * (X_0 + X_1)}{X_0 + X_2}$.
(3) $(X_0{+}X_1)^2$ is in standard form, as is the equivalent expression $X_0^2{+}2{*}(X_0{*}X_1){+}X_1^2$.
(4) $0 = X_0 * X_1^2$ is not in standard form. Its standard form is $0 = X_0 \vee 0 = X_1$.
(5) $0 < X_0^2 * X_1^3$ is not in standard form. Its standard form is $0 \neq X_0 \wedge 0 < X_1$.
(6) $0 \leq X_0^6$ is not in standard form. Its standard form is *true*.

2.3. Symbolic execution

I briefly review the main concepts from symbolic execution in a simple context; refer to [Siegel and Zirkel 2011] for details.

2.3.1. States. A program defines some set V of typed variables. A *concrete state* of the program consists of a value for the *program counter*, which specifies the current control location of the program, and a *valuation*, which is a type-respecting map from

V to some set Val of *values*. For example, a program with two integer variables u and v has a concrete state

$$\langle l_0; \mathtt{u} \mapsto 6, \mathtt{v} \mapsto 2\rangle \tag{1}$$

in which control is at some program location l_0, u has the value 6, and v has value 2.

A *symbolic state* consists of a program counter; a boolean symbolic expression called the *path condition*; and a *symbolic valuation*, which is a type-respecting map from V to the set of symbolic expressions. A symbolic state represents a (possibly infinite) set of concrete states—all concrete states obtained by substituting concrete values for the symbolic constants in such a way that the path condition evaluates to *true*. The program mentioned above has a symbolic state

$$\langle l_0; X_0 > X_1; \mathtt{u} \mapsto 2 * X_0, \mathtt{v} \mapsto X_1\rangle$$

in which control is at l_0, the path condition is $X_0 > X_1$, u is assigned the symbolic expression $2 * X_0$, and v is assigned X_1. The set of concrete states represented by this symbolic state contains concrete state (1), as the assignment $X_0 = 3, X_1 = 2$ causes the path condition to evaluate to *true*.

2.3.2. Symbolic execution for program verification. Symbolic execution entails a reachability analysis of a state-transition sytem in which the states are symbolic states and transitions correspond to program statements. Each program statement must be interpreted on the symbolic level in a way that is compatible with its concrete semantics. For the most part, this is straightforward: e.g., an assignment `x=e;` is executed by evaluating e symbolically, assigning the resulting symbolic expression to x in the new state, and updating the program counter appropriately.

At a branch on a boolean expression c, two outgoing transitions are enabled: one for the case where c is *true* and one for the case where c is *false*. To execute the *true* transition, c is added to the path condition, i.e., the new value of the path condition is the conjunction of the old value and the result of evaluating c. For the *false* transition, $\neg c$ is added to the path condition. Hence the path condition keeps track of all the choices made at branch points along an execution path.

Assertions are verified by checking the validity of the asserted formula ψ under the assumption that the path condition ϕ holds, i.e., by asking whether $\phi \Rightarrow \psi$ is valid. This validity implies that the assertion holds for all concrete states represented by the symbolic state. If validity can be established for all reachable symbolic states, the assertion holds on all reachable concrete states [Siegel and Zirkel 2011, Theorem 8].

2.3.3. State simplification. Two symbolic states are *equivalent* if they represent the same set of concrete states. As with symbolic expressions, it is desirable to transform symbolic states into an equivalent standard and/or simpler form. It can reduce the number of symbolic states searched, because a state is more likely to be recognized as equivalent to a state seen before. It can also improve the effectiveness of the validity checking.

The first step is to use standard form for all symbolic expressions occurring in the state; as discussed above, this is always the case when using SARL. However, simplification of symbolic states can go much further. The techniques used by SARL include the following:

(1) An *interval analysis* is applied to all monics in the path condition. This information is then used to simplify occurrences of that monic throughout the state. In some case this analysis obtains a single concrete value for the monic.
(2) When a concrete value is obtained for a symbolic constant, the value is substituted for the symbol everywhere in the state, and the symbol is completely eliminated.
(3) The set of all numeric equalities derived from the path condition is treated as a linear system of equations in which the "variables" are the monics. Gaussian

elimination is applied to this system to obtain further concrete values for monics and to eliminate some monics if they can be expressed as linear combinations of the remaining monics.

(4) When a sequence of array-write operations over an array is *complete* (i.e., there is a write to every cell), the entire expression is replaced by a concrete array.

(5) Since symbolic constants can appear and disappear from the state (e.g., due to dynamic memory allocation, or control leaving a scope), at certain points they are renamed in a canonical way (e.g., X_0, X_1, $\ldots$).

The state simplification process is one of the most expensive operations performed by the CIVL verifier. However, it is quite effective and often the vast majority of validity queries are resolved by simplification alone. This greatly reduces the number of calls to external provers.

3. CHALLENGE 1: MATRIX MULTIPLICATION

The first challenge involved matrix multiplication of square matrices. There were three tasks to the challenge.

3.1. Task 1: Specification and verification of a “naïve” algorithm

Task 1 presented pseudocode for a “naive” version of matrix multiplication:

```
int[][] matrixMultiply(int[][] A, int[][] B) {
  int n = A.length;
  // initialise C
  int[][] C = new int[n][n];
  for (int i = 0; i < n; i++) {
    for (int k = 0; k < n; k++) {
      for (int j = 0; j < n; j++) {
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }
  return C;
}
```

The task was to “[p]rovide a specification to describe the behaviour of this algorithm, and prove that it correctly implements its specification.”

My solution is given in Figure 2. The function `matrixMultiply` in the solution is nearly identical to the pseudocode, with some small changes, e.g., a pointer to the output matrix has been made a parameter to the function, in a typical C style. The remaining code sets up a general environment, provides some auxiliary functions for the specification, and implements a driver with an assertion to test the correctness of the output.

Several of the global variable declarations use the CIVL-C `$input` type qualifier.[4] An input variable represents a program input. The variable is initialized using the following protocol: first, if a value for that variable is specified on the command line, that value is used. For example,

```
civl verify -inputBOUND=10 mmp1.cvl
```

[4] Like all CIVL-C language primitives not in standard C, the name of this primitive starts with `$`. Also, C and CIVL-C keywords appear in `blue` text in this article.

```
 1 #include <civlc.cvh>
 2 $input int BOUND; // upper bound on N
 3 $input int N; // the size of the matrices
 4 $assume(1<=N && N<=BOUND);
 5 $input float A0[N][N], B0[N][N], C0[N][N]; // arbitrary input matrices
 6 void matrixMultiply(int n, float C[][], float A[][], float B[][]) {
 7   for (int i=0; i<n; i++)
 8     for (int j=0; j<n; j++)
 9       C[i][j] = 0.0;
10   for (int i=0; i<n; i++)
11     for (int k=0; k<n; k++)
12       for (int j=0; j<n; j++)
13         C[i][j] += A[i][k] * B[k][j];
14 }
15 float dot(int n, float u[], float v[]) {
16   float sum = 0;
17   for (int i=0; i<n; i++) sum += u[i]*v[i];
18   return sum;
19 }
20 // gets the index-th column of matrix mat:
21 float * column(int n, float result[], float mat[][], int index) {
22   for (int i=0; i<n; i++) result[i] = mat[i][index];
23   return &result[0];
24 }
25 int main() {
26   float actual[N][N], tmp[N];
27   matrixMultiply(N, actual, A0, B0);
28   for (int i=0; i<N; i++)
29     for (int j=0; j<N; j++)
30       $assert(dot(N, A0[i], column(N, tmp, B0, j))  == actual[i][j]);
31 }
```

Fig. 2. Challenge 1, part 1: "naïve" matrix multiplication (mmp1.cvl)

specifies a value of 10 for variable BOUND. If no value is specified on the command line but an initializer is present, the initializer is used; this is a good way to provide a default value. Finally, if neither the command line value nor an initializer is present, the variable is assigned an arbitrary, unconstrained value of its type; for symbolic execution, this is accomplished by assigning the variable a fresh symbolic constant. All input variables are read-only.

In the solution, BOUND is used in an assumption to place an upper bound on N. Hence the command above will verify the program for all matrices of size 10 or smaller. If no concrete value of BOUND is provided, the verifier will run forever (or until it runs out of memory).

The idea for the specification is that the (i, j)-th entry of the product should be the dot product of the i-th row of A and the j-th column of B. To do this I implemented a function to extract a column from a matrix and a function to compute the dot product of two vectors. The assertion checks this for all (i, j). There is no difficulty in verifying this: all of the assertions follow immediately from the standard form. One can insert a printf statement between lines 29 and 30 to see the symbolic expressions; the output begins

```
C[0][0] =
  X_A0[0][1]*X_B0[1][0]+X_A0[0][2]*X_B0[2][0]+X_A0[0][3]*X_B0[3][0]+
  X_A0[0][4]*X_B0[4][0]+X_A0[0][5]*X_B0[5][0]+X_A0[0][6]*X_B0[6][0]+
  X_A0[0][7]*X_B0[7][0]+X_A0[0][8]*X_B0[8][0]+X_A0[0][9]*X_B0[9][0]+
  X_A0[0][0]*X_B0[0][0]
```

The final output gives the result and several statistics:

```
CIVL v1.7 of 2016-03-31 -- http://vsl.cis.udel.edu/civl
```

```
... declaration of inputs and definition of matrixMultiply from Figure 2 ...
void main() {
  $atomic {
    float T1[N][N], T2[N][N], R1[N][N], R2[N][N];
    matrixMultiply(N, T1, A0, B0); // T1 <- A0*B0
    matrixMultiply(N, R1, T1, C0); // R1 <- T1*C0
    matrixMultiply(N, T2, B0, C0); // T2 <- B0*C0
    matrixMultiply(N, R2, A0, T2); // R2 <- A0*T2
    $assert($equals(&R1, &R2)); // check deep equality of two objects
  }
}
```

Fig. 3. Challenge 1, part 2: associativity of matrix multiplication (`mmp2.cvl`)

```
=== Command ===
civl verify -inputBOUND=10 mmp1.cvl

=== Stats ===
   time (s)            : 4.74
   memory (bytes)      : 163053568
   max process count   : 1
   states              : 35127
   states saved        : 11747
   state matches       : 0
   transitions         : 35126
   trace steps         : 11746
   valid calls         : 190978
   provers             : cvc4, z3, cvc3
   prover calls        : 37

=== Result ===
The standard properties hold for all executions.
```

The output indicates that the verification time is just under 5 seconds. The "standard properties" include all of the safety properties listed in Section 1 and in particular indicate that the assertion can never be violated.

3.2. Task 2: Verifying associativity

The second task was to "[s]how that matrix multiplication is associative, i.e., the order in which matrices are multiplied can be disregarded: $A(BC) = (AB)C$. To show this, you should write a program that performs the two different computations, and then prove that the result of the two computations is always the same."

The solution is given in Figure 3 and solves the problem exactly as specified. Verification time is 10 seconds for $\texttt{BOUND} = 10$. The solution uses a special CIVL-C function `$equals`, which accepts two pointers to C objects and tests for "deep equality." The definition is what you would expect: two arrays are "equal" if they have the same length and corresponding elements are "equal", two floats are "equal" if they are the same rational number, etc. Again, the assertion is completely resolved by the standard form for nt-polynomials, in particular the unique ordering for arguments to $+$ and $*$.

The body of the `main` function is placed in an `$atomic` block. This primitive is typically used in concurrent programs and prevents other processes from executing while one process is in the atomic region. A side-effect of CIVL's `$atomic` is that it reduces the number of state canonicalizations, which can make a significant performance difference, even in sequential programs. I have used it in the revised solutions because it decreases the verification time.

3.3. Task 3: Equivalence of the naïve and Strassen algorithms for matrix multiplication

The third challenge dealt with Strassen's matrix multiplication algorithm, which is real-equivalent to the naive algorithm but requires fewer multiplications and has slightly smaller asymptotic time complexity. The challenge statement referred participants to [Wikipedia 2016] for general information on the algorithm, and to [Thoma 2013] for code implementing the algorithm in several languages. The challenge was to prove the equivalence of the naive and Strassen's algorithm for arbitrary square matrices with size a power of 2.

My solution is given in Figure 4. Verification time is 4 seconds for `BOUND` $= 8$, and 93 seconds for `BOUND` $= 16$. Function `strassenR` is a direct transliteration of the Java code from the given web page. The environment creates arbitrary input matrices as before, but restricts to sizes which are a power of 2. `LEAF_SIZE` is another parameter used in [Thoma 2013]; it is the threshold below which ordinary matrix multiplication is used. It wasn't immediately clear to me what the appropriate values for this parameter were, but verification succeeded using the range $2..$`N`, and I have kept this range in the revision. A lower bound of 1 will also work, though verification time goes up to 8s for `BOUND` $= 8$ and 375s for `BOUND` $= 16$.

The solution uses CIVL's `$elaborate` function, which consumes a positive integer n. Semantically, this function is a no-op, but it signals the verifier to use a different search strategy. The verifier generates a sequence of transitions, all departing from the current state. The first assumes $n = 0$, the next $n = 1$, and so on. The verifier must be able to determine a concrete upper bound on n from the current path condition for this to succeed. The effect is to eliminate a symbolic constant from the state by exploring separate cases for all of its possible concrete values. This can increase the number of states, but simplify the symbolic expressions and prover queries. It is difficult to predict whether this strategy will be a net win, so CIVL leaves it up to the user. In this case it decreases verification time significantly.

3.4. Discussion

The differences between the revised solutions presented here and my original submission are relatively minor. Besides formatting and stylistic changes, the original solution was in one file, which I broke up into three separate files here in order to get more precise timings. I also added some atomic statements for better performance.

I missed the fact that the problem called for matrices with integer entries, and used `floats` instead. It would be trivial to change the `floats` to `ints`. However this brings up an important issue, which is the difference between floating-point and real number semantics. CIVL currently interprets all floating point operations as real arithmetic, which is why it declared these programs correct. With floating-point semantics, the associativity of matrix multiplication (task 2) fails to hold, since floating-point multiplication is not associative. Similarly, Strassen's algorithm (task 3) is not floating-point-equivalent to the naive algorithm. These discrepancies are not due to bugs—it is well known that Strassen's algorithm, for example, has floating-point characteristics that differ from that of the naive algorithm, but it is expected that it—or any "correct" matrix multiplication algorithm—be real-equivalent to the naive one. Clearly, floating-point semantics are useful for specifying some properties of numerical programs, and real semantics are useful for specifying others. We need tools that can handle both.

The specification in task 1 was essentially accomplished using more C code. This worked, but arguably the new C code is at least as complex as the original code being verified. While there can be advantages to using the programming language as the specification mechanism, in this case I feel a better solution could be constructed if there were additional specification primitives, in particular a "sum" operator, as in

```
 ... declaration of inputs and definition of matrixMultiply from Figure 2 ...
$input int LEAF_SIZE; // threshold below which ordinary matrix multiplication is used
$assume (2 <= LEAF_SIZE && LEAF_SIZE <= N);
void add(int n, float C[][], float A[][], float B[][]) {
  for (int i=0; i<n; i++) for (int j=0; j<n; j++) C[i][j] = A[i][j] + B[i][j];
}
void subtract(int n, float C[][], float A[][], float B[][]) {
  for (int i=0; i<n; i++) for (int j=0; j<n; j++) C[i][j] = A[i][j] - B[i][j];
}
// Strassen algorithm from https://martin-thoma.com/strassen-algorithm-in-python-java-cpp/
void strassenR(int n, float C[][], float A[][], float B[][]) { // requires n is power of 2
  if (n <= LEAF_SIZE) {
    matrixMultiply(n, C, A, B);
  } else {
    int m = n/2;
    float a11[m][m], a12[m][m], a21[m][m], a22[m][m];
    float b11[m][m], b12[m][m], b21[m][m], b22[m][m];
    float aResult[m][m], bResult[m][m];
    for (int i=0; i<m; i++)
      for (int j=0; j<m; j++) {
        a11[i][j] = A[i][j];      a12[i][j] = A[i][j + m];
        a21[i][j] = A[i+m][j];    a22[i][j] = A[i + m][j + m];
        b11[i][j] = B[i][j];      b12[i][j] = B[i][j+m];
        b21[i][j] = B[i+m][j];    b22[i][j] = B[i+m][j+m];
      }
    add(m, aResult, a11, a22); add(m, bResult, b11, b22);
    float p1[m][m]; strassenR(m, p1, aResult, bResult); add(m, aResult, a21, a22);
    float p2[m][m]; strassenR(m, p2, aResult, b11); subtract(m, bResult, b12, b22);
    float p3[m][m]; strassenR(m, p3, a11, bResult); subtract(m, bResult, b21, b11);
    float p4[m][m]; strassenR(m, p4, a22, bResult); add(m, aResult, a11, a12);
    float p5[m][m]; strassenR(m, p5, aResult, b22); subtract(m, aResult, a21, a11);
    add(m, bResult, b11, b12);
    float p6[m][m]; strassenR(m, p6, aResult, bResult); subtract(m, aResult, a12, a22);
    add(m, bResult, b21, b22);
    float p7[m][m]; strassenR(m, p7, aResult, bResult);
    float c12[m][m]; add(m, c12, p3, p5);
    float c21[m][m]; add(m, c21, p2, p4); add(m, aResult, p1, p4);
    add(m, bResult, aResult, p7);
    float c11[m][m]; subtract(m, c11, bResult, p5); add(m, aResult, p1, p3);
    add(m, bResult, aResult, p6);
    float c22[m][m]; subtract(m, c22, bResult, p2);
    // Grouping the results obtained in a single matrix:
    for (int i=0; i<m; i++)
      for (int j=0; j<m; j++) {
        C[i][j]   = c11[i][j]; C[i][j+m]   = c12[i][j];
        C[i+m][j] = c21[i][j]; C[i+m][j+m] = c22[i][j];
      }
  }
}
_Bool isPowerOf2(int n) {
  while (n>1) {
    if (n%2 != 0) return $false;
    n = n/2;
  }
  return $true;
}
int main() {
  $atomic {
    $elaborate(N); // hint to verifier: iterate over concrete values of this variable
    $assume(isPowerOf2(N));
    float R1[N][N], R2[N][N];
    matrixMultiply(N, R1, A0, B0);
    strassenR(N, R2, A0, B0);
    $assert($equals(&R1, &R2));
  }
}
```

Fig. 4. Challenge 1, part 3: equivalence of Strassen's algorithm (mmp3.cvl)

$\sum_{i=a}^{b}$. (The ACSL specification language [ACSL 2016] specifies such an operator.) One could even imagine adding a number of standard operators from functional programming languages, like *fold*, *reduce*, and *filter*. These could lead to shorter and more intuitive specifications. On the other hand, the use of CIVL-C's `$equals` primitive was extremely useful for comparing two matrices, avoiding the need for iteration over every entry.

The careful reader may have noticed a little twist in the naive algorithm: the last two loops have been transposed from their usual order. At the discussion session, it was clear this complicated deductive approaches—in particular, the construction of appropriate loop invariants. In contrast, it makes no difference at all to the symbolic execution approach used here. I don't believe I even noticed this twist until after the competition.

4. CHALLENGE 2: BINARY TREE TRAVERSAL

The second challenge dealt with a binary tree

```
class Tree {
  Tree left, right, parent;
  bool mark;
}
```

with the properties

— "following a child pointer (left or right) and then following a parent pointer brings us to the original node"
— "the parent pointer of the root is null"
— "It has at least one node, and each node has 0 or 2 children"
— "We do not know the initial value of the mark fields"

and gave an algorithm [Morris 1979] for traversing the nodes in linear time but constant space:

```
void markTree(Tree root) {
  Tree x, y;
  x = root;
  do {
    x.mark = true;
    if (x.left == null && x.right == null) {
      y = x.parent;
    } else {
      y = x.left;
      x.left = x.right;
      x.right = x.parent;
      x.parent = y;
    }
    x = y;
  } while (x != null);
}
```

The tasks are to prove:

(1) "upon termination of the algorithm, all mark fields are set"
(2) "the tree shape does not change"
(3) "the code does not crash"
(4) "the code terminates"
(5) "the nodes are visited in depth-first order" (bonus).

```
#include <civlc.cvh>
#include <stdbool.h>
#include <stdlib.h>
$input int DB; // depth bound
typedef struct _tree { struct _tree *left, *right, *parent; _Bool mark; } *Tree;
void markTree(Tree root) { // mark every node, using only constant space
  Tree x=root, y;
  do {
    x->mark = true;
    if (x->left == NULL && x->right == NULL) { y = x->parent; }
    else { y = x->left; x->left = x->right; x->right = x->parent; x->parent = y; }
    x = y;
  } while(x != NULL);
}
Tree makeTree(Tree left, Tree right, _Bool mark) { // makes a new tree
  Tree result = (Tree)malloc(sizeof(struct _tree));
  result->left = left; result->right = right; result->mark = mark; result->parent = NULL;
  if (left != NULL) left->parent = result;
  if (right != NULL) right->parent = result;
  return result;
}
Tree makeArbitrary(int depth) { // construct an arbitrary tree with height < "depth"
  if (depth == 0) return NULL;
  // nondeterministic choice: 0: return tree with 0 nodes, 1: return tree with 2 nodes
  if ($choose_int(2) == 0) return makeTree(NULL, NULL, false);
  return makeTree(makeArbitrary(depth - 1), makeArbitrary(depth - 1), false);
}
Tree copyTree(Tree root) { // deep copy a tree, ignoring marks
  if (root == NULL) return NULL;
  return makeTree(copyTree(root->left), copyTree(root->right), false);
}
void freeTree(Tree root) { // free memory allocated by making a tree
  if (root != NULL) { freeTree(root->left); freeTree(root->right); free(root); }
}
_Bool allMarked(Tree t) { // checks every node in Tree t is marked
  if (t != NULL) {
    if (!(t->mark)) return false;
    if (!allMarked(t->left)) return false;
    if (!allMarked(t->right)) return false;
  }
  return true;
}
_Bool wellFormed(Tree t) { // all non-leaf nodes u: u==u->left->parent==u->right->parent
  if (t->left != NULL) {if (t->left->parent!=t || !wellFormed(t->left)) return false;}
  if (t->right != NULL) {if (t->right->parent!=t || !wellFormed(t->right)) return false;}
  return true;
}
_Bool isomorphic(Tree t1, Tree t2) { // given two well-formed trees, are they isomorphic?
  if (t1 == NULL) return t2 == NULL;
  if (t2 == NULL) return false;
  if (!isomorphic(t1->left, t2->left)) return false;
  return isomorphic(t1->right, t2->right);
}
int main() {
  $atomic {
    Tree t1 = makeArbitrary(DB), t2 = copyTree(t1); // create arbitrary tree, save a copy
    $assume(t1 != NULL);
    markTree(t1); // markTree should mark every node and not change the shape
    $assert(allMarked(t1)); $assert(t1->parent == NULL); $assert(wellFormed(t1));
    $assert(isomorphic(t1, t2)); // check that the shape of t1 has not changed
    freeTree(t1); freeTree(t2);
  }
}
```

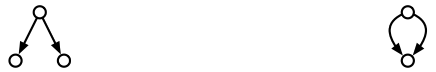
Fig. 5. Challenge 2: binary tree traversal (tree.cvl)

My solution is shown in Figure 5. The program will explore all trees for which the number of edges along any path from the root to a leaf is less than DB. Verification takes about 3s for DB $= 4$ and 77s for DB $= 5$.

Function markTree is virtually identical to the code given in the challenge. To this I added a constructor makeTree, and a function makeArbitrary which uses nondeterministic choice to create an arbitrary tree satisfying the assumptions given in the challenge. The function $choose_int consumes an integer n and returns an integer in $0..n-1$; when verifying a program, all possible choices are explored. The remaining functions are used to specify the desired properties of the algorithm.

Function allMarked checks that all the mark fields are set. It recurses over all nodes and returns false if any unmarked node is found.

To check that the shape of t1 does not change, a deep copy t2 of the original tree is made using copyTree. This is compared with t1 after markTree(t1). The function isomorphic consumes two tree-or-nulls and checks that either both are null or the corresponding children are isomorphic. In my original solution, I thought this sufficed, but the organizers pointed out it would declare the following two structures to have the same shape:



In the revision, I have corrected this by checking that in t1, the root has null parent, and for any node u, the parent of a child of u is u. (These necessarily hold for t2.) This implies that every node is a child of at most one node, and the root is a child of no node. It follows that isomorphic establishes a structure-preserving bijection between the nodes of the two trees.

The only way that the code of markTree could crash is from an invalid or null pointer dereference. The CIVL verifier checks every dereference for these anomalies, so the code cannot crash for any tree within the specified bound.

CIVL does not have a specific mechanism to verify termination or other liveness properties. It does verify absence of deadlocks, but a program can fail to terminate without deadlocking because of an infinite loop. There are, however, a few “tricks” that can sometimes be used to verify termination. The first is to run the verifier with the flag -saveStates=false. This instructs the verifier not to save seen states as it searches the state space. If there is a cycle in the state space, this will cause the verifier to run forever. Hence if the verifier returns and reports all properties hold (in particular, absence of deadlock), every execution must terminate. Doing this for tree.cvl with DB $= 4$ leads the verifier to return after 47s. This represents a significant slow-down, due to the large number of states that are traversed multiple times, but establishes termination for trees within this bound. I did not try this for DB $= 5$.

The second trick is to modify the program by adding counters to loops. I inserted int count=0; after line 7 and count++; after 8. If the do..while loop could loop forever, the verifier would never return (even with saving of states), because every iteration yields a new state. Running this modified program, the verifier returned for both DB $= 4$ and DB $= 5$ with no noticeable slow-down.

I never attempted the bonus task on depth-first order.

4.1. Structure of the state: recursion, pointers, and dynamically-allocated memory

As this example illustrates, CIVL has no problem verifying programs with recursive functions, dynamically allocated data structures, and pointers, as long as the call stack and heap cannot grow without bound. A CIVL state has a complex and dynamic structure, as illustrated in Figure 6. New *dynamic scopes* are added whenever control enters

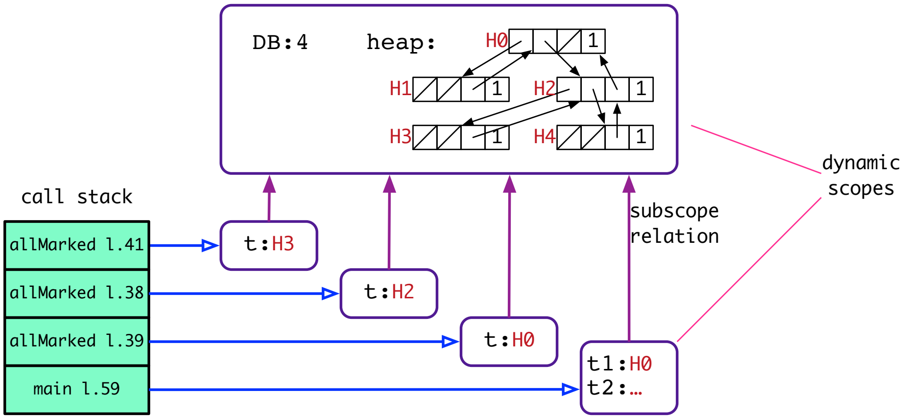


Fig. 6. Structure of a state for `tree.cvl`

a static scope, and are removed when control exits the static scope. Each dynamic scope stores the (symbolic) values of variables declared in that scope. In the pictured state, the large dynamic scope at the top is the global scope, which also contains the heap. Because of recursion, there can be multiple instances of a single static scope: three instantiations of the function scope for `allMarked` in this example. The call stack consists of a sequence of frames, each of which includes a reference to a dynamic scope and a program counter value.

A pointer is represented as a triple (d, i, r), where d is the ID number of a dynamic scope, i is the ID of a variable within that scope, and r is a *reference expression*. A reference expression is a symbolic expression that specifies a "path" to an internal point of a compound value, such as "array element 5 of tuple component 3 of tuple component 0." These expressions are supported by SARL, which provides methods to build and manipulate them, and a method to "apply" a reference expression to a symbolic expression of the appropriate compound type in order to extract the specified subexpression. This last method is used to implement pointer dereferencing. CIVL also supports most, but not all, forms of pointer arithmetic permitted under the C Standard.

5. CHALLENGE 3: TREE BARRIER

The third challenge was a concurrency problem: verification of a tree barrier. The number N of threads is fixed. There is also a fixed binary tree in which the nodes correspond to threads:

```
class Node {
  final Node left, right;
  final Node parent;
  boolean sense;
  int version;
  ... methods ...
}
```

This node class has a *barrier* method which is called by each thread to create a global synchronization point:

```
void barrier()
      requires !sense
      ensures  !sense
{
      // synchronization phase
```

```
        if (left != null) while (!left.sense) { }
        if (right != null) while (!right.sense) { }
        sense = true;  // assume this statement and the next to execute
        version++;     // simultaneously (that is, in one step)
        // wake-up phase
        if (parent == null) sense = false;
        while(sense) { }
        if (left != null) left.sense = false
        if (right != null) right.sense = false
  }
```

When a thread enters the barrier, it waits for a signal from its children indicating that they and their descendants have entered the barrier. Then the thread signals its parent. When the root receives this signal, it knows all threads have arrived, and the signal then travels down the tree, freeing threads to leave the barrier.

The challenge continues: "Assume a state in which `sense` is *false* and `version` is zero in all nodes. Assume further that no thread is currently executing `barrier()` and that threads invoke `barrier()` only on their nodes. The number of threads (and, thus, the number of nodes in the tree) is constant, but unknown" and specifies two tasks: to prove

(1) the following invariant holds in all states: If `n.sense` is *true* for any node `n` then `m.sense` is *true* for all nodes `m` in the subtree rooted in `n`.
(2) for any call `n.barrier()`, if the call terminates then there was a state during the execution of the method where all nodes had the same version.

My solution is given in Figure 7. Verification time is 13s for $\mathtt{N} = 4$ and 173s for $\mathtt{N} = 5$.

The solution uses several CIVL-C concurrency primitives. The `$spawn` expression (line 63) wraps a function call, and creates a new *process* (or thread) to execute the call. The new process has its own call stack and the single frame on that stack will point to a new dynamic scope corresponding to the function body—the same scope that would be created by an ordinary function call. Evaluation of the `$spawn` expression returns immediately with a reference to the new process, an object of type `$proc`. Figure 8 shows the structure of a typical state in `barrier.cvl`.

In the solution, I have added a field `p` to the node structure (line 6). The value returned by `$spawn` is assigned to that field in the node corresponding to the new process. This is used later as an argument to the `$wait` function (line 64), which blocks until the referenced process terminates. This is the only place where the field is used, so the modification to the node structure could have been avoided by storing the `$proc` values in a local array instead. Even better: CIVL-C provides the `$parfor` primitive, which captures the common pattern of lines 63–64; those lines can be replaced with

```
                $parfor (int i : 0 .. N-1) thread(theNodes[i]);
```

In addition to `$wait`, synchronization takes place though *guarded commands*, which have the form `$when (`*expr*`)` *stmt*. This command blocks until *expr* evaluates to *true*, then executes *stmt*; the first atomic sub-step of *stmt* executes atomically with the testing of the guard expression. This basic concurrency primitive can be used to implement semaphores, locks, and other higher-level concurrency constructs. For the guarded commands in the solution, the *stmt* is just a no-op. These are used in place of the pseudocode's busy-wait loops.

Another CIVL-C feature not in standard C is the ability to nest function definitions. In the solution, function `barrier` is placed inside function `thread`. The body of `thread` (line 39) may invoke `barrier`, but code outside of `thread` may not. Moreover, `barrier` may read and modify the local variables of `thread`, e.g., `myNode`. These internal func-

```
#include <civlc.cvh>
#include <stdlib.h>
#include <stdbool.h>
$input int N; // the number of nodes (tried with N=1,2,3,4,5)
typedef struct _node { // a node in a binary tree
  $proc p; // the process to which this node is associated
  struct _node *left, *right *parent; // left child, right child, and parent
  _Bool sense; // am I in the barrier?
  int version; // number of times I have gone through barrier
} * Node;
Node theNodes[N]; // array storing all nodes created so far
int count = 0; // number of nodes created so far
void checkDescendantsSensesTrue(Node u) { // check all senses in u and descendants are true
  if (u != NULL) {
    $assert(u->sense);
    checkDescendantsSensesTrue(u->left); checkDescendantsSensesTrue(u->right);
  }
}
void checkAncestorSensesFalse(Node u) { // check all senses in u and ancestors are false
  while (u != NULL) { $assert(!u->sense); u = u->parent; }
}
void thread(Node myNode) { // the function each thread will run
  void barrier() { // the barrier function
    if (myNode->left != NULL) $when (myNode->left->sense); // wait for left child's sense
    if (myNode->right != NULL) $when (myNode->right->sense);
    $atom {
      myNode->sense = true;
      checkDescendantsSensesTrue(myNode); // check invariant
      myNode->version++;
      if (myNode->parent == NULL) // root: check everyone has same version
        for (int i=0; i<N; i++) $assert(theNodes[i]->version == myNode->version);
    }
    if (myNode->parent == NULL) myNode->sense = false;
    $when (!myNode->sense); // wait until my sense is false
    $atom { checkAncestorSensesFalse(myNode); } // check invariant
    if (myNode->left != NULL) myNode->left->sense = false;
    if (myNode->right != NULL) myNode->right->sense = false;
  }
  for (int i=0; i<3; i++) barrier(); // driver: run around the barrier 3 times...
}
Node makeTree(Node left, Node right) { // make a tree from given children
  Node result = (Node)malloc(sizeof(struct _node));
  result->left = left; result->right = right; result->sense = false; result->version = 0;
  if (left != NULL) left->parent = result;
  if (right != NULL) right->parent = result;
  result->parent = NULL;
  return result;
}
Node makeArbitraryTree(int numNodes) { // create an arbitrary tree with numNodes nodes
  if (numNodes == 0) return NULL;
  int leftSize = $choose_int(numNodes);
  Node leftTree = makeArbitraryTree(leftSize);
  Node rightTree = makeArbitraryTree(numNodes - leftSize - 1);
  Node result = makeTree(leftTree, rightTree);
  theNodes[count] = result; count++;
  return result;
}
void freeTree(Node tree) { // free all nodes in the tree
  if (tree != NULL) { freeTree(tree->left); freeTree(tree->right); free(tree); }
}
int main() {
  Node theTree = makeArbitraryTree(N);
  $atomic { for (int i=0; i<N; i++) theNodes[i]->p = $spawn thread(theNodes[i]); }
  for (int i=0; i<N; i++) $wait(theNodes[i]->p);
  freeTree(theTree);
}
```

Fig. 7. Challenge 3: tree barrier (barrier.cvl)

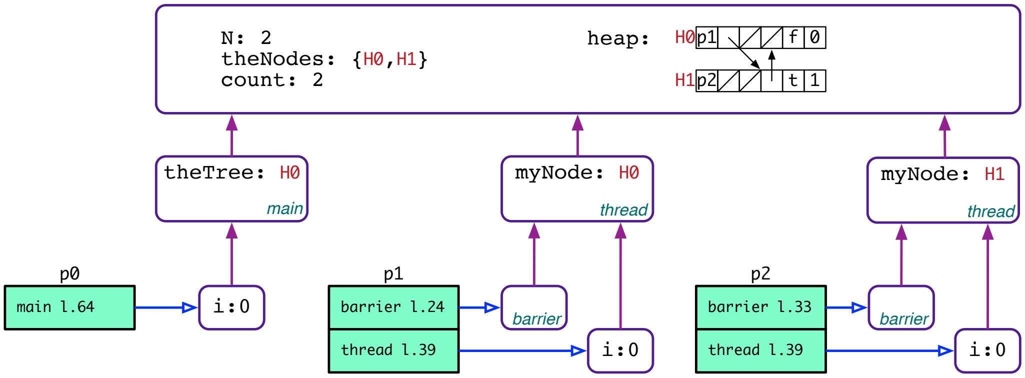


Fig. 8. A state of `barrier.cvl`

tions can be spawned just like any other function. While this feature is not used in the challenges, it is used extensively in other applications of CIVL, specifically to represent so-called “hybrid” parallel programs that use multiple levels of concurrency, such as multi-threaded MPI programs, or CUDA-C programs.

The `thread` function invokes the barrier three times. This use of 3 is arbitrary; it should be another parameter to the model. Some bound must be used, because the `version` field is incremented with each call and can therefore increase without bound—i.e., there would be an infinite number of states.[5]

The exact number of nodes `N` is specified as an input, and an arbitrary binary tree with `N` nodes is constructed. Verification time for $\texttt{N} = 4$ is 15s; $\texttt{N} = 5$ takes 203s.

CIVL checks automatically that the program does not deadlock, though the challenge did not specifically ask this.

The first task is to prove an invariant—a property that holds at every state. CIVL does not provide a way to specify invariants, but I was able to approximate this using assertions. The claim is that at any state, for any node n, if n.`sense` holds then m.`sense` holds for all descendants m of n. At the point just after `sense` is set to *true*, I inserted a function that checks that all descendants of `myNode` have *true* sense. Moreover, at the point just after waiting for `sense` to become *false*, I inserted a function that checks that all ancestors of `myNode` have *false* sense (this function was added in the revision). Putting these together, one has something very close to the requested invariant.

The second task is to show there is a state at which all nodes have the same `version`. It seemed to me that such a state would occur just as the root increments its `version`. I inserted code at this point to check this assertion. However, in the original statement of the challenge, the comment about the two statements `sense=true` and `version++` executing simultaneously did not appear, and my draft solution therefore did not include those two statements within an `$atomic` block. CIVL reported a violation of the assertion.

One of the purported advantages of model checking is its ability to generate a counterexample execution *trace* when a property is violated. This was certainly the case here. I have also found that the ability to produce *minimal* counterexamples is extremely useful for defect understanding. So I set $\texttt{N} = 2$ and ran the verifier with the `-min` flag, which instructs the verifier to find a path of minimal length to a violating state. The file `barrier_bad.cvl`, included in the experimental archive, demonstrates

[5] In other barrier examples distributed with CIVL, the driver is actually an infinite loop, yet the verifier still converges because the number of states is finite, so the loop will eventually reach states seen before.

this strategy. The minimal counterexample consists of 49 trace steps, the first 36 of which deal with the initialization phase of the program. After finding the counterexample, the trace can be *replayed* using the command

```
civl replay -showTransitions barrier_bad.cvl
```

to show the step-by-step sequence leading to the violation. An excerpt of the final part of the trace follows:

```
Step 43: Executed by p1 from State 43:
  53->54: (*(&<d0>heap.malloc0[0][0])).sense=true ...
Step 47: Executed by p2 from State 47:
  53->54: (*(&<d0>heap.malloc0[1][0])).sense=true ...
Step 48: Executed by p2 from State 48:
  54->55: ENTER_ATOMIC [_atomic_lock_var:=p2, p2.atomicCount:=1]
  55->56: (*(&<d0>heap.malloc0[1][0])).version=0+1
  56->57: TRUE_BRANCH_IF (guard: (void*)0==((struct _node)*)0)
  57->58: i=0 at barrier_bad.cvl:38.6-12 "int i=0"
  58->59: LOOP_BODY_ENTER (guard: 0<2) at barrier_bad.cvl:38.15-17 "i<N"
Error 0:
CIVL execution violation in p2 (kind: ASSERTION_VIOLATION, certainty: PROVEABLE)
at barrier_bad.cvl:38.25-72 "$assert(theNodes[i]->version  ... )":
Assertion: ((*((theNodes)[i])).5==(*(myNode)).5)
        -> 0==1
        -> false
Step 49: Trace ends after 49 trace steps.
Violation(s) found.
```

In this trace, p1 sets its sense to *true*, then the root process p2 intervenes, enters the barrier, sets its sense to *true*, and increments its version from 0 to 1. At this point, p2 checks the assertion and discovers that p1's value of 0 differs from its value of 1.

By examining this trace, it became clear that the two statements should happen atomically. I notified the organizers, as did one other participant (Bart Jacobs), and the organizers quickly fixed the statement. I was not able to complete the second task in time, but did so shortly after the competition. With the two statements inside an $atomic block, CIVL was able to verify the complete program for $\mathtt{N} \leq 5$.

6. CONCLUSION

I have presented CIVL solutions to the VerifyThis 2016 challenges that address almost all of the tasks. In each case, CIVL was able to verify the specified properties within small but non-trivial bounds, e.g., 16×16 matrices for Strassen multiplication, binary trees with height at most 4 for Morris' tree traversal algorithm, and up to 5 threads for the tree barrier. The verification runtimes range from a few seconds to a few minutes.

I believe the solutions provide some evidence for the usability of CIVL by programmers of moderate skill. The solutions are short, and do not require much knowledge beyond basic C. A few new primitives and concepts are needed, but these are minimal and do not require great conceptual leaps. The bulk of the work involves setting up a general environment and driver, which is similar to the work involved in ordinary program testing.

Detailed comparisons with other tools are beyond the scope of this paper, but a look at some of the other solutions can give a feel for the differences between deductive approaches and that of CIVL.

CIVL was the only tool to solve the Strassen task during the competition. Afterwards, three participants used Why3 [Filliâtre and Paskevich 2013] to construct a complete solution to challenge 1. The solution includes an implementation and proof of a version of Strassen's algorithm for square matrices of any size, not just powers of 2; see [Clochard et al. 2016a] and [Clochard et al. 2016b]. The solution consists of over 1000 lines of WhyML code, including function contracts, predicate, type, and function definitions, invariants, axioms, lemmas, and theories—including an axiomatic theory of matrix arithmetic which could be re-used in other contexts. All of the verification conditions generated from this system are discharged by fully automatic theorem provers. This is a landmark achievement, but certainly required a level of sophistication beyond that of a typical programmer, as well as a good deal of effort.

Solutions to challenges 2 and 3 using VeriFast are also notable [Jacobs 2016]. These consist of Java code with annotations in a language based on separation logic. The annotations are used to construct a proof which is automatically checked by VeriFast. The solution to challenge 2 is complete (excluding the bonus task on depth-first order) and includes 92 non-whitespace annotation lines in addition to the implementation code. The solution to challenge 3 does not address the second task on the consistency of the version number, but is otherwise complete; it uses over 220 non-whitespace annotation lines.

The cognitive process is also different. Construction of a proof requires a deep understanding of the algorithm. This was evident in the post-competition discussion and in the solutions mentioned above. The user who succeeded in constructing a proof ended up understanding exactly *why* the Morris algorithm works, *why* the Strassen algorithm computes the product of the two matrices, and so on. In contrast, the user of CIVL can almost treat the algorithm as a black box. The exception is when something goes wrong, such as the missing atomicity requirement in challenge 3. In that case, CIVL's ability to generate a minimal counterexample proved very useful in understanding the defect.

Several improvements to CIVL would help it solve problems like those discussed here. These include:

— reporting the presence or absence of cycles in the state space
— additional specification primitives such as a `$sum` operator
— a way to specify invariants (properties expected to hold at every state), or even general temporal properties
— a mechanism to help generate arbitrary trees and other dynamic data structures
— the ability to specify loop invariants and to use them to verify programs without bounding input sizes.

The CIVL project is actively working on these and other improvements.

ACKNOWLEDGMENTS

I am grateful to Dr. Manchun Zheng and Ziqing Luo for their leading roles in the development of CIVL.

Funding for the CIVL project is provided by the U.S. National Science Foundation under awards CCF-1319571, CCF-1346769 and CCF-0953210.

This material is also based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under Award Number DE-SC0012566.

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

REFERENCES

ACSL 2016. ACSL: ANSI/ISO C Specification Language. http://frama-c.com/acsl.html. (2016). Accessed Aug. 25, 2016.

Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings (Lecture Notes in Computer Science)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.), Vol. 6806. Springer, 171–177. http://dl.acm.org/citation.cfm?id=2032305.2032319

Clark Barrett and Cesare Tinelli. 2007. CVC3. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings (Lecture Notes in Computer Science)*, Werner Damm and Holger Hermanns (Eds.), Vol. 4590. Springer, 298–302.

B. F. Caviness. 1970. On Canonical Forms and Simplification. *J. ACM* 17, 2 (April 1970), 385–396. DOI:http://dx.doi.org/10.1145/321574.321591

CIVL 2017. CIVL: Concurrency Intermediate Verification Language. https://vsl.cis.udel.edu/civl. (2017). Accessed Mar. 14, 2017.

Martin Clochard, Léon Gondelman, and Mário Pereira. 2016a. The Matrix Reproved (Verification Pearl). In *Verified Software. Theories, Tools, and Experiments: 8th International Conference, VSTTE 2016, Toronto, ON, Canada, July 17–18, 2016, Revised Selected Papers*, Sandrine Blazy and Marsha Chechik (Eds.). Springer, 107–118. DOI:http://dx.doi.org/10.1007/978-3-319-48869-1_8

Martin Clochard, Léon Gondelman, and Mário Pereira. 2016b. Solutions — VerifyThis 2016. http://toccata.lri.fr/gallery/verifythis2016.en.html. (2016). Accessed March 8, 2017.

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science)*, C. R. Ramakrishnan and Jakob Rehof (Eds.), Vol. 4963. Springer, 337–340. DOI:http://dx.doi.org/10.1007/978-3-540-78800-3_24

Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3: Where Programs Meet Provers. In *Proceedings of the 22nd European Conference on Programming Languages and Systems (ESOP'13)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer-Verlag, Berlin, Heidelberg, 125–128. DOI:http://dx.doi.org/10.1007/978-3-642-37036-6_8

Marieke Huisman, Rosemary Monahan, and Peter Müller. 2016a. VerifyThis Verification Competition to be held at ETAPS 2016, Saturday, 2 April 2016. http://etaps2016.verifythis.org. (2016). Accessed March 8, 2017.

Marieke Huisman, Rosemary Monahan, Peter Müller, and Erik Poll. 2016b. *VerifyThis 2016: A Program Verification Competition*. Technical Report TR-CTIT-16-07. Centre for Telematics and Information Technology, University of Twente, Enschede. http://eprints.eemcs.utwente.nl/27060/01/sttt-summary.pdf.

IEEE. 2004. IEEE POSIX 1003.1c Standard. http://www.unix.org/version3/ieee_std.html. (2004). Accessed Oct. 30, 2015.

Bart Jacobs. 2016. Partial Solutions to VerifyThis 2016 Challenges 2 and 3 with VeriFast. In *Proceedings of the 18th Workshop on Formal Techniques for Java-like Programs (FTfJP'16)*. ACM, New York, NY, USA, Article 7, 6 pages. DOI:http://dx.doi.org/10.1145/2955811.2955818

Message-Passing Interface Forum. 2015. MPI: A Message-Passing Interface Standard, Version 3.1. http://www.mpi-forum.org/docs/docs.html. (4 June 2015).

Joseph M. Morris. 1979. Traversing Binary Trees Simply and Cheaply. *Inf. Process. Lett.* 9, 5 (1979), 197–200. DOI:http://dx.doi.org/10.1016/0020-0190(79)90068-1

NVIDIA. 2017. CUDA Zone. https://developer.nvidia.com/cuda-zone. (2017). Accessed March 14, 2017.

OpenMP 2017. OpenMP web site. http://www.openmp.org. (2017). Accessed March 14, 2017.

Stephen F. Siegel, Manchun Zheng, Ziqing Luo, Timothy K. Zirkel, Andre V. Marianiello, John G. Edenhofner, Matthew B. Dwyer, and Michael S. Rogers. 2015. CIVL: The Concurrency Intermediate Verification Language. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. ACM, New York, Article 61, 12 pages. DOI:http://dx.doi.org/10.1145/2807591.2807635

Stephen F. Siegel and Timothy K. Zirkel. 2011. TASS: The Toolkit for Accurate Scientific Software. *Mathematics in Computer Science* 5, 4 (2011), 395–426.

Martin Thoma. 2013. Part II: The Strassen algorithm in Python, Java and C++. https://martin-thoma.com/strassen-algorithm-in-python-java-cpp/. (Jan. 2013).

Wikipedia. 2016. Strassen algorithm. https://en.wikipedia.org/wiki/Strassen_algorithm. (2016).

Security by Compilation: An Automated Approach to Comprehensive Side-channel Resistance

Chao Wang
University of Southern California

Patrick Schaumont
Virginia Tech

We explain how formal verification and program synthesis can be used to (1) detect side-channel leaks of software code running on portable devices, (2) prove the absence of side-channel leaks, and (3) transform software to eliminate such leaks. We use power side-channel leaks in cryptographic software as examples, but the underlying techniques are applicable to other types of side channels and software systems as well.

1. INTRODUCTION

Programmers often view the computing devices as blackboxes, but real computers leak information of the software they execute through various side channels, e.g., variations in power dissipation, radiation, execution time, and sound signature of the processor. Side-channel information may be exploited by adversaries. For example, while cryptographic algorithms may be secure against hundreds of years of brute-force attacks, their actual implementations may be broken in hours or even minutes in the presence of side-channel leaks.

Since the seminal work on differential power analysis [Kocher et al. 1999], many similar techniques have been developed, making side-channel analysis (SCA) a real threat to commercial hardware and software. Examples from the past few years include the use of SCA to extract secret keys from contactless smartcards [Kasper et al. 2011], keyless entry systems [Eisenbarth et al. 2008], secure memory modules [Balasch et al. 2012], and field programmable gate arrays (FPGAs) [Moradi et al. 2011; Skorobogatov and Woods 2012; Moradi et al. 2013]. Furthermore, new sources of side-channel leakage and methods to exploit them are discovered on a regular basis. For example, the *sound* made by a processor as it executes the public-key RSA algorithm was found to be a viable side channel to reveal the secret key [Genkin et al. 2014] — although a patch was quickly made and distributed, it is still worrisome that side-channel leakage can be found in such widely distributed production software.

A fundamental difficulty in mitigating side-channel leaks is that their physical sources are at the level of the processor hardware, an abstraction layer typically invisible to programmers. For example, data-dependent control flow in software will impact the power dissipation of the processor hardware, but for an average programmer, it may be difficult or even impossible to tell how much harmful side-channel leakage would occur. A similar problem exists with the prediction of execution time, another source of side-channel leakage, or the prediction of cache misses and their impact on the execution time. Average programmers tend to view the computing device as an abstract machine while ignoring architectural details of the processor (such as out-of-order execution) and storage (such as register allocation and cache), thus making side-channel leakage a concept that is hard to grasp for them.

Current implementation of side-channel resistant software in embedded computing applications relies on manual efforts of experts, but even for them, this process is complex and error-prone. Furthermore, there are no techniques available to verify these handcrafted implementations, let alone generating them automatically. Although automation is desirable, existing verification and synthesis techniques are not sufficient. The reason is because, first, side-channel resistance is a *non-functional* property, which cannot be handled by techniques developed for proving functional correctness [Clarke et al. 1999; McMillan 1994]. Furthermore, unlike the *non-interference* property in information-flow security [Sabelfeld and Myers 2003], side-channel resistance is a *statistical* property, which requires fundamentally new analysis techniques.

We outline the development of a new type of verification and program synthesis techniques to aid in the construction of side-channel resistant software for embedded computing applications, e.g., cryptographic software used in various cyber-physical systems (CPS) and the Internet of things (IoT) where physical security of the computing devices is a major concern. As shown in Fig. 1, the automated analysis and code transformation framework consists of techniques being developed along the following directions:

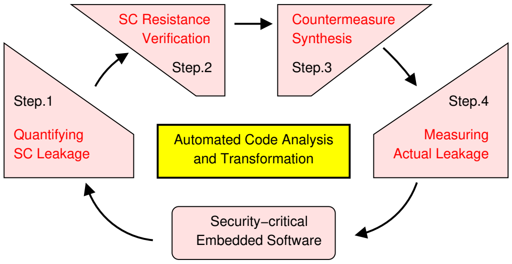

SC Resistance Verification
Step.2
Countermeasure Synthesis
Step.3
Step.1
Quantifying SC Leakage
Automated Code Analysis and Transformation
Step.4
Measuring Actual Leakage
Security–critical Embedded Software

Fig. 1. Automated approach to side-channel resistance.

- *Quantifying side-channel leaks*. First, we need to formally define what it means for a piece of software to be side-channel resistant on a given platform, and in case it is not side-channel resistant, how to quantify the amount of leakage.
- *Verifying side-channel resistance*. For existing and manually-secured software code, we need new verification techniques to formally prove that the implementation is indeed side-channel resistant.
- *Synthesizing countermeasures*. We also need program synthesis techniques for automatically generating functionally-equivalent, but side-channel resistant, software code to replace the original code. They must go beyond simple compiler transformations, to handle unknown vulnerabilities and generate new implementations.
- *Validation on real devices*. Finally, the resulting software code must be validated on real devices to ensure our modeling and synthesis accurately reflect side-channel leaks observed in the physical world.

In the remainder of this article, we use power side-channel leaks in cryptographic software as examples to illustrate our recent work on formally verifying side-channel resistance [Eldib et al. 2014b; Eldib et al. 2014c] as well as synthesizing countermeasures [Eldib and Wang 2014b]. Then, we discuss how to extend these techniques to handle other types of side channels and software systems.

2. PRELIMINARIES

We assume the software code implements a cryptographic function $c \leftarrow f(x,k)$, where x is the plaintext, c is the ciphertext, and k is the secret key. The goal of the adversary is to compute k based on knowledge of x and c as well as the information of internal computations leaked through side channels.

It is possible to implement $c \leftarrow f(x, k)$ in a manner such that the side-channel leakage remains harmless, e.g., using the idea of secret sharing [Chari et al. 1999]. In this approach, every internal variable v of the software program is split into $n + 1$ shares $v_0, v_2, \ldots, v_n$ such that $v = v_0 \oplus v_1 \oplus \ldots \oplus v_n$, where $\oplus$ is a suitable masking operator, e.g., the XOR operator in Boolean domain. Among these $n + 1$ shares, n are randomly chosen masks and the remaining one is computed as a matching share. Since every masked share v_i is statistically independent of the original v, leakage of individual shares or any combination of $\leq n$ shares will not reveal v.

Splitting variables into shares affects the internal operations of the program. Thus, we call the new program a *masked* program. Furthermore, the number of shares corresponds to the *order* of masking, e.g., in an *order-d* masking, every variable is split into $d + 1$ shares. If $f(x, k)$ were a linear function of k with respect to XOR, masking would be straightforward, because $f(x, k \oplus r) \oplus f(x, r) = f(x, k) \oplus f(x, r) \oplus f(x, r) = f(x, k)$. That is, we can mask the sensitive k by the XOR with a random variable r before the computation, and de-masking afterward by the XOR with $f(x, r)$. However, in practice, $f(x, k)$ is always a non-linear function, which means masking requires a complete rewriting of the software code, and the process is labor-intensive and error-prone.

Threat Model. We assume an adversary knows the value of the plaintext x, the ciphertext c, and side-channel information of at most d intermediate computation results; they correspond to variables in the program. Let $I_1, I_2, \ldots, I_d$ be the set of intermediate results. Furthermore, each $I_i(x, k, r)$ is a function in terms of x, k, and random variable r introduced to *mask* the sensitive k. Thus, the adversary does not have access to the value of r. However, if the side-channel leakage associated with I_i or any combination of $\leq d$ intermediate results is dependent of k, we say the implementation of $c \leftarrow f(x, k)$ is vulnerable to SCA based attacks.

A necessary condition for $f(x, k)$ to be side-channel resistant is that all intermediate computation results are either logically independent of k or logically dependent of (and thus masked by) some random variable r. The condition seems reasonable and can be easily checked [Bayrak et al. 2013]. However, it is a logical property (as opposed to statistical property)—we will show that the condition is not sufficient for ensuring side-channel resistance.

Leakage Model. A widely used power model is the *Hamming Weight (HW)* model, which relates variations in the power dissipation of the processor to values of its registers, which in turn hold variables used in the software program. More specifically, the power dissipation correlates to the number of logical-1 bits of intermediate computation results. We have shown in our work [Eldib et al. 2014c] that the HW model is sufficiently accurate for conducting DPA attacks on embedded systems. Sometimes, however, the *Hamming Distance (HD)* model needs to be used instead, to relate variations in power dissipation to differences between the register values and their initial states [Brier et al. 2004].

The example in Fig. 2 shows that, under the HW model, *logically dependent of some random variable* is not the same as *statistically-independent of the secret*. Here, `k` is the secret bit, `r1` and `r2` are the random bits, and `o1`, `o2`, `o3`, and `o4` are the masked intermediate results. According to the truth table on the right-hand side or functions on the left-hand side, all four intermediate results are logically dependent of `r1,r2` and thus are masked. However, the first three still leak secret information because they are not perfectly masked. Specifically, `o1` leaks information of `k` because, if it were logical 1, `k` would also be logical 1 regardless of the values of the random variables. `o2` leaks information of `k` because, if it were logical 0, `k` would also be logical 0. `o3` leaks information of `k` because, if it were logical 1 (or 0), there would be a 75% chance that `k`

```
o1 = x ∧ k ∧ (r1 ∧ r2)
o2 = x ∧ k ∨ (r1 ∧ r2)
o3 = x ∧ k ⊕ (r1 ∧ r2)
o4 = x ∧ k ⊕ (r1 ⊕ r2)
```

x	k	r1	r2	o1	o2	o3	o4
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	1	0
0	1	1	1	1	1	0	1

Fig. 2. Four masking schemes with different side-channel leakages and the corresponding truth table when $x = 0$. Although o1,o2,o3 are masked by random bits r1 and r2, they still leak secret information about k. In contrast, o4 does not have side-channel leakage.

is also logical 1 (or 0). In contrast, o4 does not leak information of k because, regardless of whether k is logical 1 (or 0), there is a 50% chance that o4 is logical 1 (or 0).

Perfect Masking. Following [Blömer et al. 2004], we define *perfect masking* for the implementation of $c \leftarrow f(x,k)$ as follows. Given a pair (x,k) of plaintext and secret key, together with d intermediate results $I_1(x,k,r), \ldots, I_d(x,k,r)$, where r is a random variable in the domain R, we say f is order-d *perfectly masked* if the joint distribution of $I_1, \ldots, I_d$ is independent of k. Otherwise, we say the implementation is vulnerable to *order-d* SCA-based attacks. The intermediate result o4 in Fig. 2, for example, is perfectly masked and thus is immune to first-order attacks.

3. VERIFYING THE SIDE-CHANNEL RESISTANCE

A verification procedure for deciding *if $f(x,k)$ is perfectly masked* works as follows. Initially, the input variables are annotated such that all plaintext bits in x are marked as public, all key bits in k are marked as secret, and all bits in r are marked as random. Then, for each intermediate result, denoted $I(x,k,r)$, the procedure checks if I is perfectly masked by r.

For ease of presentation, we assume $d = 1$. Thus, verifying if I is perfectly masked is the same as checking the *validity* of the following formula:

$$\forall x. \forall k. \forall k'. \left(\sum_{r \in R} I(x,k,r) = \sum_{r \in R} I(x,k',r) \right)$$

Here, x denotes the plaintext value, k and k' denote two values of the key, and r denotes the random variable. Thus, for each combination (x,k,k'),

— $\sum_{r \in R} I(x,k,r)$ denotes the number of values of r making I logical 1; and
— $\sum_{r \in R} I(x,k',r)$ denotes the number of values of r making I logical 1.

Assume that r is uniformly distributed in R, the above summations are probabilities of I being logical 1 under the plaintext value x and the two key values k and k'.

Given a cryptographic software program, we first obtain a branch-free representation by merging all if-else branches. Since there are typically no input-dependent loop bounds (otherwise, they may be timing side channels), we apply loop unrolling to obtain a loop-free program. Since all variables are bounded integers, we can model them as finite-length bit-vectors or even construct a purely Boolean program. Fig. 3 shows a masked version of $c \leftarrow (k1 \wedge k2)$, where $r1$ and $r2$ are two random bits. The corresponding de-masking function, which is not shown in the figure, would be $c \oplus (r1 \wedge r2)$. Due to the property of XOR, de-masking would produce the desired value $(k1 \wedge k2)$.

Thus, we can traverse the abstract syntax tree (AST) of the given program, and for each intermediate result I, check if I is perfectly masked. For ease of implementation, instead of checking the validity of the above universally-quantified formula, we use a

```
compute(bool k1, bool k2, bool r1, bool r2){
  bool n1, n2, n3, n4, n5, n6, n7, n8, c;
  n1 = k1 ⊕ r1;
  n2 = k2 ⊕ r2;
  n3 = n1 ∧ n2;
  n4 = k2 ⊕ r2;
  n5 = r1 ∧ n4;
  n6 = k1 ⊕ r1;
  n7 = r2 ∧ n6;
  n8 = n5 ⊕ n7;
  c = n3 ⊕ n8;
  return c;
}
```

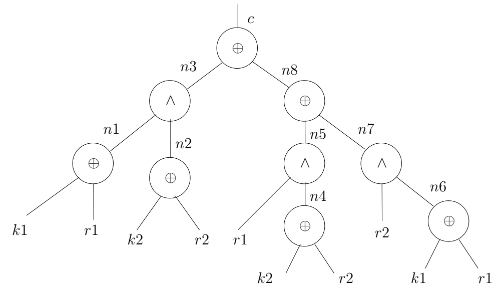


Fig. 3. Example Boolean program and its graphic representation ($\oplus$ denotes XOR; $\wedge$ denotes AND).

constraint solver to check the satisfiability of its negation, shown as follows:

$$\exists x.\exists k.\exists k' \; . \; \left(\sum_{r \in R} I(x,k,r) \neq \sum_{r \in R} I(x,k',r) \right)$$

If the formula is satisfiable, the solver will return a plaintext value x and two different key values (k, k') such that the probabilities of $I(x,k,r)$ and $I(x,k',r)$ being logical 1 differ. Therefore, some information of k is leaked. In contrast, if this formula is unsatisfiable, it means no such leak is possible.

Model Counting. Thus, the verification of side-channel resistance can be viewed as comparing the number of satisfying assignments of two closely-related formulas. This is the case not only for power side channels, but also for other types of side channels because, fundamentally, the attacks all rely on correlation-based statistic analysis. Consequently, unlike standard verification techniques, which rely on SAT and SMT solvers as the decision procedures, the new verification techniques need SAT$^{\#}$ and SMT$^{\#}$ solvers to support model-counting. Although model-counting solvers are not yet as mature as standard SAT and SMT solvers in terms of speed and scalability, they are catching up rapidly [Chakraborty et al. 2013; Chakraborty et al. 2014; Aydin et al. 2015; Fremont et al. 2017].

Without using specialized solvers, we can still solve the verification problem [Eldib et al. 2014b], albeit in a less efficient fashion. Fig. 4 is a pictorial illustration of our encoding for $I(k_1, k_2, r_1, r_2)$, where k_1, k_2 are key bits and r_1, r_2 are random bits. Each box in the figure denotes a copy of the input-output relation of I but with random bits customized to values 00, 01, 10, and 11, respectively. Furthermore, the first four boxes correspond to one set of key values, denoted k_1 and k_2, and the remaining four boxes correspond to another set of key values, denoted k_1' and k_2'. The summations add up the number of logical 1's,

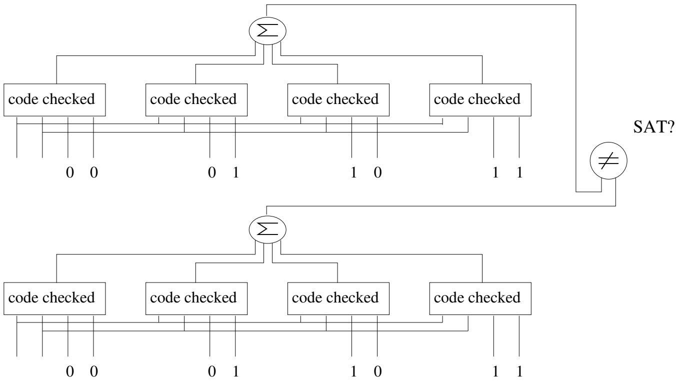


Fig. 4. Checking the statistical dependence of secret data $(k1, k2)$.

while the comparison on the right-hand side checks if the probabilities of $I(k_1, k_2, \ldots)$ and $I(k'_1, k'_2, \ldots)$ being logical 1 can differ.

Compositional Verification. Whether we use specialized solvers or standard solvers does not change the fact that, in the worst case, the number of satisfying assignments is exponential in the number of random bits in r. This may cause scalability problems. Fortunately, certain properties of masked programs allow us to apply compositional analysis. That is, instead of verifying the whole program, we partition the AST into small code regions, and apply the model-counting based analysis only to each individual code region, one at a time.

This is possible because a common strategy used by cryptographic system engineers is to create a chain of small modules, where the inputs of each module are masked before executing its logic and are de-masked afterward. To avoid having unmasked intermediate values, the inputs to the successor module are masked with fresh random variables before they are de-masked from the random variables used by the previous module. Due to the *associativity* of XOR ($\oplus$), reordering these masking and de-masking operations would not change the result. We have shown [Eldib et al. 2014b] that such property may be exploited for performance optimization in real applications.

Quantifying the Leakage. Our verification procedure so far only checks if a given program is perfectly masked. However, it cannot quantify the amount of leakage in programs that are not perfectly masked. To differentiate the strengths of masking schemes, e.g., `o1,o2,o3` in Fig. 2, we have extended the definition of perfect masking to quantify the amount of residual leakage [Eldib et al. 2014c]. That is, we define the *quantitative masking strength (QMS)* as the minimal value of $(1 - \Delta_{qms})$ such that,

$$|E(I_i \mid k = \kappa \wedge x = \chi) - E(I_i \mid k = \kappa' \wedge x = \chi)| \leq \Delta_{qms}$$

holds for all intermediate results $I_i(x, k, r)$, all plaintext values χ, and all key values κ and κ', where $\kappa \neq \kappa'$. Here, $E(I_i \mid k = \kappa \wedge x = \chi)$ can be viewed as the number of values of r under which $I_i(\kappa, \chi, r)$ evaluates to logical 1.

Consider the example in Fig. 2 again. We have

$$\begin{array}{ll}
\Delta_{qms}(o1) = 1/4 - 0/4 = 0.25 & \Delta_{qms}(\overline{o1}) = 4/4 - 3/4 = 0.25 \\
\Delta_{qms}(o2) = 4/4 - 1/4 = 0.75 & \Delta_{qms}(\overline{o2}) = 3/4 - 0/4 = 0.75 \\
\Delta_{qms}(o3) = 3/4 - 1/4 = 0.50 & \Delta_{qms}(\overline{o3}) = 3/4 - 1/4 = 0.50 \\
\Delta_{qms}(o4) = 2/4 - 2/4 = 0.00 & \Delta_{qms}(\overline{o4}) = 2/4 - 2/4 = 0.00
\end{array}$$

Intuitively, the numbers are consistent with the amount of power leakage. For example, the perfectly-masked `o4` has $\Delta_{qms} = 0.0$, which corresponds to $QMS = 1.0$. For each of the remaining three, the larger Δ_{qms}, the more information it leaks.

To decide whether a given software program meets the QMS requirement, we check if there exists any intermediate result $I(x, k, r)$ that satisfies the following formula:

$$\exists x, k, k' \ . \ \left(\sum_{r \in R} I(x, k, r) - \sum_{r \in R} I(x, k', r) \right) > \Delta_{qms} \ .$$

If this formula is satisfiable, there exist some values for x and (k, k') such that the difference between distributions of $I(x, k, r)$ and $I(x, k', r)$ is larger than the expected Δ_{qms}. On the other hand, if the above formula is unsatisfiable for all intermediate results of the program, the implementation meets the QMS requirement.

4. SYNTHESIZING SIDE-CHANNEL RESISTANT SOFTWARE

Given some not-yet-masked software code as input, we use *inductive program synthesis* to systematically search for an alternative, functionally-equivalent, but side-channel resistant implementation. Although recent years have seen a renewed interest

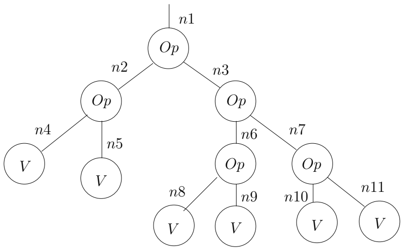


Fig. 6. A candidate program skeleton consisting of 11 parameterized AST nodes.

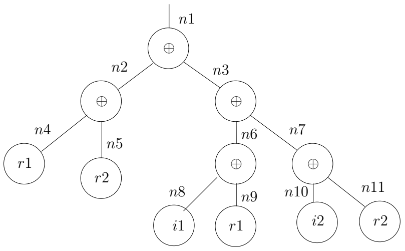


Fig. 7. The synthesized candidate program with instantiated Boolean masking.

in applying *inductive synthesis* to a wide variety of applications [Solar-Lezama et al. 2005; Jha et al. 2010; Gulwani 2011; Harris and Gulwani 2011; Harris et al. 2013; Alur et al. 2013; Eldib and Wang 2014a; Eldib et al. 2016], prior to our work, it has never been used to mitigate side-channel leaks.

Our synthesis procedure relies on a set of architectural parameters to estimate the leakage. For each side channel, we leverage a different type of code transformation, or countermeasure, to eliminate the leakage. Specifically, for instruction timing, the countermeasure would be CFG-balancing, which is to remove all branching conditions that are dependent on the sensitive data. For cache-memory timing, the countermeasure would be to remove the dependency between table lookups and the sensitive table content. For power side channel, the countermeasure would be masking, which removes the dependency between variations in power dissipation and the sensitive data.

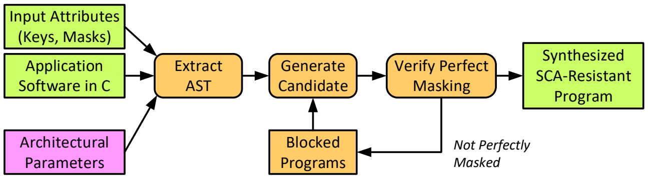


Fig. 5. Counterexample-guided inductive synthesis procedure.

The overall flow of our synthesis procedure is shown in Fig. 5. Given the application software together with a set of sensitive variables and architectural parameters, it first extracts an abstract syntax tree (AST) representation of the program. Then, it generates a *candidate program* that is functionally equivalent to the original program—the two programs produce the same output for the same input. Next, it verifies that the candidate program is free of side-channel leaks. If the verification succeeds, we are done. Otherwise, we block this candidate program and try again.

To generate the candidate program, we create a *skeleton* of the program's AST, which captures any syntactically correct program up to that size. For example, the skeleton AST of size 5 shown in Fig. 6 can represent any candidate program with up to five AST nodes: *Op* represents any of the predefined binary operators, $V|C$ means the node represents either a variable or a constant, and the root node represents the computation output, which must be functionally equivalent to the original program.

We use SMT solvers to search among the candidate programs. That is, to determine the node types, variable names, and constant values of the skeleton AST, we construct a formula Φ such that Φ is satisfiable if and only if the candidate program is functionally equivalent to the original program. If Φ is unsatisfiable, it means no solution exists; in this case, we increase the skeleton size and try again.

If Φ is satisfiable, we have found a candidate program, e.g., as in Fig. 7, which is an instantiation of the skeleton AST. The next step is to verify that it is free of side-channel leaks. Toward this end, we create another formula Ψ such that Ψ is satisfiable if and only if the candidate program has side-channel leaks. If Ψ is unsatisfiable, the candidate program is proved to be a valid solution and we are done. Otherwise, we block this candidate program and try again.

Fig. 8 shows a masked implementation of the χ-function of a reference implementation of MAC-Keccak, which is NIST's new SHA-3 crypto-hashing algorithm [NIST 2013]. The original code is on the left-hand side and the new code is on the right-hand side. We guarantee that all intermediate results in the new program are perfectly masked. That is, by assuming `r1, r2` and `r3` are uniformly distributed random variables, our method guarantees that the probability of each intermediate result being logical 1 (or 0) is independent of `i1, i2` and `i3`. As for the compactness of the implementation, we note that a countermeasure handcrafted by cryptographic experts has 14 operations [Bertoni et al. 2013], whereas our synthesized version only has 12 operations—it is more compact than the one handcrafted by experts.

```
Chi(bool i1, bool i2, bool i3) {
  bool n1, n2, n3;
  n3 = ¬i2;
  n2 = n3 ∧ i3;
  n1 = n2 ⊕ i1;
  return n1;
}
```

i1	i2	i3	n3	n2	n1
0	0	0	1	0	0
0	0	1	1	1	1
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	1	0	1
1	0	1	1	0	1
1	1	0	0	0	1
1	1	1	0	0	1

```
mChi(bool i1, bool i2, bool i3) {
  bool r1, r2, r3;  //random bits added
  bool b1, b2, b3, n1, n2, n3, n4, n5, n6, n7, n8, n9;
  b1 = i1 ⊕ r1;
  b2 = i2 ⊕ r2;
  b3 = i3 ⊕ r3;
  n9 = b3 ∧ r2;
  n8 = r3 ∧ r2;
  n7 = r3 ∨ b2;
  n6 = r1 ⊕ n9;
  n5 = n7 ⊕ n8;
  n4 = b2 ∨ b3;
  n3 = n5 ⊕ n6;
  n2 = n4 ⊕ b1;
  n1 = n2 ⊕ n3;
  return n1;
}
```

Fig. 8. The χ function in MAC-Keccak, its truth-table, and the synthesized $m\chi$ function ($\neg$ denotes NOT, $\wedge$ denotes AND, $\vee$ denotes OR, and $\oplus$ denotes XOR).

Compositional Synthesis. Again, the key is to exploit the unique characteristics of masked programs. Thus, we have developed a compositional synthesis procedure [Eldib and Wang 2014b], which applies computationally intensive analysis (e.g., model-counting) only to small code regions, one at a time, as opposed to the entire program. Compared to the application of standard synthesis techniques to the entire program, our compositional synthesis procedure is significantly more scalable.

5. VALIDATING SIDE-CHANNEL RESISTANCE ON REAL DEVICES

To confirm that our modeling and analysis of side-channel leaks at the source code level accurately reflect what is observed in the physical world, we conducted a set of SCA-based attacks on implementations of MAC-Keccak, AES, and a few other cryptographic algorithms [Eldib et al. 2015]. In these experiments, we ran all software code on a 32-bit Microblaze processor [Xilinx 2014] built on a Xilinx Spartan-3e FPGA (Fig. 9). To measure the power dissipation of the processor core, we used a Tektronix DPO 3034 oscilloscope and a CT-2 current probe to sample the power dissipation. The side-channel attack shown in Fig. 9 was conducted using the classic differential power analysis, i.e., difference of means [Kocher et al. 1999]. To limit the effect of measurement noise, we collected each *trace* after running the same software code 128 times and using the oscilloscope to calculate the average. Thus, a trace refers to a set of samples taken during the execution of the software code.

We used differential power analysis (DPA) to determine if a key guess was correct. Recall that DPA relies on the observation that power dissipation variations correlate to the values of the sensitive bits being manipulated. Using the same input vector stream of plaintext as in the measured traces, we computed the value of the sensitive variable assuming that the secret key was one of the key guesses. For an n-bit key, there would be 2^n key guesses. For each key guess, we divided the set of measurement traces into two bins, one for all the sensitive values of logic 0, and one for all the sensitive values of logic 1. Then, we computed the difference of means between those two bins for each key guess, and selected the key guess that result in the maximum difference.

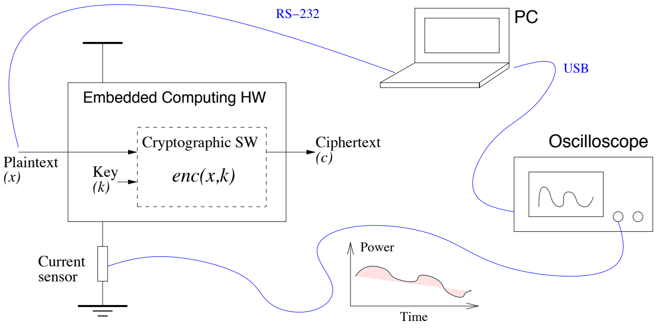


Fig. 9. The power side-channel attack system setup.

Fig. 10 shows our results on the SHA3 benchmark. The x-axis denotes the QMS value as defined in Section 3, while the y-axis (in logarithmic scale) denotes the number of traces needed to determine the secret key. In addition to the measured data, which are the stars in the figure, we plotted an empirical approximation rule (dotted curve) generated by hit-and-trial to estimate the measured data. We can see that when the QMS approaches 1.0, the number of traces needed to determine the secret key approaches infinity. However, when the QMS deviates from 1.0 slightly, the number of traces needed to determine the secret key drops quickly. Overall, the side-channel resistance as measured by the number of traces needed to determine the secret key is dependent on QMS. Fig. 11 shows our results on the AES benchmark.

In both cases, the approximate empirical formula computed to estimate the number of required DPA traces has the following relation with the QMS value:

$$N_{trace} = \frac{1}{(1 - \textsc{qms})^c} \ ,$$

where $c \approx 2.0$. Note that we obtained this equation without prior knowledge of what the relation should look like. Later, we discovered that it matches the theoretical analysis result in the literature [Mangard 2004], which says that c should be precisely 2.0 as opposed to ≈ 2.0, since $(1 - \textsc{qms})$ represents the standard deviation of power analysis measurements.

6. FUTURE DIRECTIONS

The next step is to generalize the verification and program synthesis techniques to handle other types of side channels and software systems. We envision a comprehensive framework (Fig. 12) whose input is the source code of some security-critical software, together with a set of sensitive variables (keys, passwords, etc.) tagged in the source code. To support modeling of various types of side-channel leakage, it also accepts a set of architecture parameters and leakage models. The output is a transformed

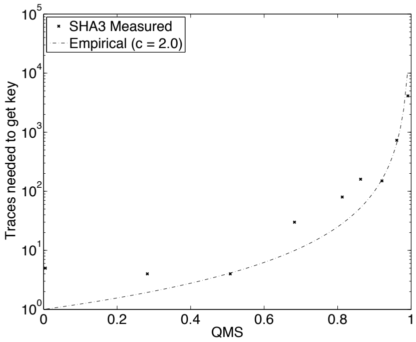


Fig. 10. DPA attacks on MAC-Keccak: number of traces needed versus the QMS.

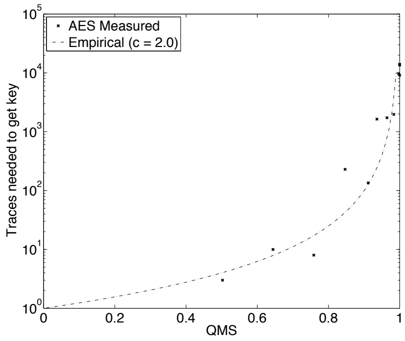


Fig. 11. DPA attacks on AES: the number of traces needed versus the QMS value.

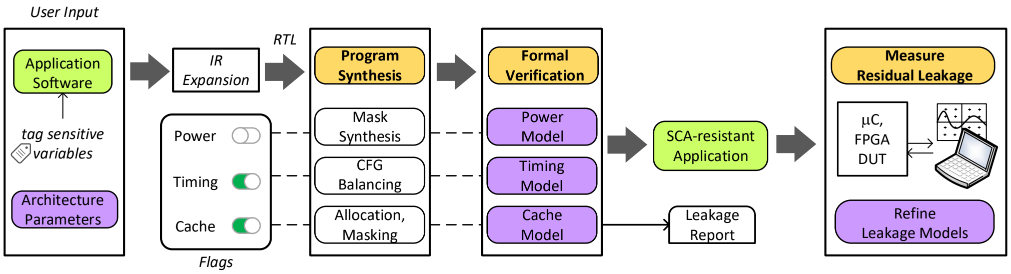


Fig. 12. Framework for synthesizing, verifying, and validating side-channel countermeasures. The input includes the source code of an application, a list of sensitive variables, and the parametric architecture definition. The compiler-like tool can (a) insert countermeasures through inductive synthesis and (b) statically detect remaining side-channel leakage in either synthesized or manually programmed countermeasures.

application for which the dependency between side-channel leakage and sensitive variables is removed.

When a programmer develops an application, for example, he or she will indicate one or more types of side-channel leakage, including power dissipation, instruction time, and cache-memory timing behavior. The framework will examine the software code to check for the presence of side-channel leakage. In the presence of side-channel leakage, the framework will leverage inductive synthesis to transform the software code into an implementation that eliminates the side-channel leakage. The framework also assesses the quality of the countermeasure. In addition, by measuring and analyzing the actual leakage of driver applications using a hardware prototype, we will refine the architecture parameters and leakage models, thus improving verification and countermeasure synthesis.

Advantages over Alternative Approaches. There are significant efforts on eliminating physical emissions of sensitive equipments and electronic systems, e.g. in the legendary TEMPEST project [TEMPEST 1972]. There are also techniques for reducing physical emissions of hardware (microcontrollers, FPGAs, ASICS, and CPUs) – although these techniques are theoretically feasible, they are not economical. In contrast, our approach does not aim to eliminate physical emissions of the computing devices; instead, it transforms the software running on these devices to make the computation *leak-resistant*. Therefore, our approach is fundamentally more economical and thus more widely applicable.

Another alternative is the use of side-channel resistant software libraries developed by experts. While libraries could help for selected cases of reusable functionality on some widely deployed platforms, it is not scalable in general, for several reasons. First, since side-channel leakage is platform-specific, side-channel resistant libraries must also be platform-specific and thus non-portable. Second, side-channel resistant techniques incur performance penalty, which means an expert has to decide which sources of side-channel leakage to address and what level of residual leakage should be tolerated. Therefore, a *universal* countermeasure library is not meaningful; in practice, the application context is crucial to decide on what makes sense and what not.

7. RELATED WORK

Formal Verification. We started with the notion of *perfect masking* introduced by [Blömer et al. 2004] and developed *SC-Sniffer* [Eldib et al. 2014a; 2014b], the first automated tool for formally verifying that a software program is perfectly masked. In comparison, the *Sleuth* tool developed by [Bayrak et al. 2013] can only check if sensitive data are masked by some random variables (a logical property), but cannot check if the masking is perfect (a statistical property). We also extended the notion of perfect masking to quantify the amount of residual leakage in software that are not perfectly masked [Eldib et al. 2014c]. The strength of masking may be computed statically on the source code of the software program, and its accuracy as an indicator for side-channel resistance has been validated by DPA attacks on real devices [Eldib et al. 2015].

Countermeasure Synthesis. There is a large body of work on masking countermeasures for cryptographic algorithms [Messerges 2000; Goubin 2001; Oswald et al. 2005; Herbst et al. 2006; Canright and Batina 2008; Moradi et al. 2011; Barthe et al. 2016], but they require manual design and implementation. By leveraging our verification procedure for proving side-channel resistance, we developed *SC-Masker* [Eldib and Wang 2014b], a tool for automatically synthesizing perfectly-masked software code. Although there exist some other compiler-like tools for mitigating side-channel leaks [Bayrak et al. 2011; Moss et al. 2012; Agosta et al. 2012], they rely on *ad hoc* techniques, e.g., matching some code patterns and applying predefined transformations, as opposed to inductive program synthesis techniques. The main advantage of using inductive synthesis is that the tool becomes application-agnostic and it no longer relies on existing patterns or mitigation strategies. Therefore, it can handle unknown and unexpected vulnerabilities.

Other Side Channels. Besides power side channels, there are other types of side channels through which sensitive information may be leaked. They include, for example, instruction timing side channels [Kocher 1996; Köpf and Dürmuth 2009], cache timing side channels [Grabher et al. 2007], string-related side channels [Bang et al. 2016], and fault-related side channels [Biham and Shamir 1997]. In addition to CPUs, side channels have been identified in GPUs [Jiang et al. 2016; Luo et al. 2015]. Techniques for mitigating some of these side-channel leaks are also proposed. For example, Köpf *et al.* developed techniques for quantitative information flow analysis [Köpf et al. 2012; Backes et al. 2009]. Doychev *et al.* [Doychev et al. 2013] developed static analysis techniques for detecting leaks through cache side channels. Barthe *et al.* [Barthe et al. 2014] developed techniques for mitigating concurrent cache attacks.

8. CONCLUSIONS

We have presented an automated approach to comprehensive side-channel resistance for embedded computing applications. It relies on formal verification techniques to detect side-channel leaks or prove that leaks do not exist, and program synthesis techniques to generate secure implementations. It also leverages hardware prototyping to validate the effectiveness of these verification and synthesis techniques. Although we have used power side-channel leaks in cryptographic software as examples, the underlying techniques may be applied to various side channels in a wide range of embedded processing systems, e.g., in phones, cars, and home appliances, as well as industrial, medical, and transportation systems.

ACKNOWLEDGMENTS

This work was primarily supported by NSF under grants CNS-1617203, CNS-1128903 and CNS-1115839. Partial support was provided by ONR under grant N00014-13-1-0527. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

Giovanni Agosta, Alessandro Barenghi, and Gerardo Pelosi. 2012. A code morphing methodology to automate power analysis countermeasures. In *ACM/IEEE Design Automation Conference*. 77–82.

Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *International Conference on Formal Methods in Computer-Aided Design*. 1–17.

Abdulbaki Aydin, Lucas Bang, and Tevfik Bultan. 2015. Automata-based model counting for string constraints. In *International Conference on Computer Aided Verification*. 255–272.

Michael Backes, Boris Köpf, and Andrey Rybalchenko. 2009. Automatic discovery and quantification of information leaks. In *IEEE Symposium on Security and Privacy*. 141–153.

Josep Balasch, Benedikt Gierlichs, Roel Verdult, Lejla Batina, and Ingrid Verbauwhede. 2012. Power analysis of Atmel CryptoMemory - recovering keys from secure EEPROMs. In *CT-RSA: The Cryptographers' Track at the RSA Conference*. 19–34.

Lucas Bang, Abdulbaki Aydin, Quoc-Sang Phan, Corina S. Pasareanu, and Tevfik Bultan. 2016. String analysis for side channels with segmented oracles. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*. 193–204.

Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. 2016. Strong non-interference and type-directed higher-order masking. In *ACM SIGSAC Conference on Computer and Communications Security*. 116–129.

Gilles Barthe, Boris Köpf, Laurent Mauborgne, and Martín Ochoa. 2014. Leakage resilience against concurrent cache attacks. In *International Conference on Principles of Security and Trust*. 140–158.

Ali Bayrak, Francesco Regazzoni, David Novo, and Paolo Ienne. 2013. Sleuth: Automated Verification of Software Power Analysis Countermeasures. In *Workshop on Cryptographic Hardware and Embedded Systems*. 293–310.

Ali Galip Bayrak, Francesco Regazzoni, Philip Brisk, François-Xavier Standaert, and Paolo Ienne. 2011. A first step towards automatic application of power analysis countermeasures. In *Proceedings of the Design Automation Conference*. 230–235.

Guido Bertoni, Joan Daemen, Michael Peeters, Gilles Van Assche, and Ronny Van Keer. 2013. Keccak implementation overview. URL: http://keccak.neokeon.org/Keccak-implementation-3.2.pdf. (2013).

Eli Biham and Adi Shamir. 1997. Differential fault analysis of secret key cryptosystems. In *International Cryptology Conference*. 513–525.

Johannes Blömer, Jorge Guajardo, and Volker Krummel. 2004. Provably secure masking of AES. In *Selected Areas in Cryptography*. 69–83.

Eric Brier, Christophe Clavier, and Francis Olivier. 2004. Correlation power analysis with a leakage model. In *Workshop on Cryptographic Hardware and Embedded Systems*. 16–29.

David Canright and Lejla Batina. 2008. A very compact "perfectly masked" S-Box for AES. In *International Conference on Applied Cryptography and Network Security*. 446–459.

Supratik Chakraborty, Daniel J. Fremont, Kuldeep S. Meel, Sanjit A. Seshia, and Moshe Y. Vardi. 2014. Distribution-aware sampling and weighted model counting for SAT. In *AAAI Conference on Artificial Intelligence*. 1722–1730.

Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. 2013. A scalable approximate model counter. In *International Conference on Principles and Practice of Constraint Programming*. 200–216.

Suresh Chari, Charanjit Jutla, Josyula Rao, and Pankaj Rohatgi. 1999. Towards sound approaches to counteract power-analysis attacks. In *International Cryptology Conference*. 398–412.

E. M. Clarke, O. Grumberg, and D. A. Peled. 1999. *Model Checking*. MIT Press, Cambridge, MA.

Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. 2013. CacheAudit: A tool for the static analysis of cache side channels. In *USENIX Security*. 431–446.

Thomas Eisenbarth, Timo Kasper, Amir Moradi, Christof Paar, Mahmoud Salmasizadeh, and Mohammad T. Manzuri Shalmani. 2008. On the power of power analysis in the real world: A complete break of the KeeLoqCode Hopping scheme. In *International Cryptography Conference*. 203–220.

Hassan Eldib and Chao Wang. 2014a. An SMT based method for optimizing arithmetic computations in embedded software code. *IEEE Trans. on CAD of Integrated Circuits and Systems* 33, 11 (2014), 1611–1622.

Hassan Eldib and Chao Wang. 2014b. Synthesis of masking countermeasures against side channel attacks. In *International Conference on Computer Aided Verification*. 114–130.

Hassan Eldib, Chao Wang, and Patrick Schaumont. 2014a. Formal verification of software countermeasures against side-channel attacks. *ACM Trans. Softw. Eng. Methodol.* 24, 2 (2014), 11:1–24.

Hassan Eldib, Chao Wang, and Patrick Schaumont. 2014b. SMT based verification of software countermeasures against side-channel attacks. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*. 62–77.

Hassan Eldib, Chao Wang, Mostafa Taha, and Patrick Schaumont. 2014c. QMS: Evaluating the side-channel resistance of masked software from source code. In *ACM/IEEE Design Automation Conference*. 209:1–6.

Hassan Eldib, Chao Wang, Mostafa Taha, and Patrick Schaumont. 2015. Quantitative masking strength: Quantifying the side-channel resistance of masked software code. In *IEEE Trans. on CAD of Integrated Circuits and Systems*, Vol. 34. 10:1558–1568.

Hassan Eldib, Meng Wu, and Chao Wang. 2016. Synthesis of Fault-Attack Countermeasures for Cryptographic Circuits. In *International Conference on Computer Aided Verification*. 343–363.

Daniel J. Fremont, Markus N. Rabe, and Sanjit A. Seshia. 2017. Maximum model counting. In *AAAI Conference on Artificial Intelligence*. 3885–3892.

Daniel Genkin, Adi Shamir, and Eran Tromer. 2014. RSA key extraction via low-bandwidth acoustic cryptanalysis. In *International Cryptology Conference*. 444–461.

Louis Goubin. 2001. A sound method for switching between boolean and arithmetic masking. In *Workshop on Cryptographic Hardware and Embedded Systems*. 3–15.

Philipp Grabher, Johann Großschädl, and Dan Page. 2007. Cryptographic side-channels from low-power cache memory. In *International Conference on Cryptography and Coding*. 170–184.

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. 317–330.

William R. Harris and Sumit Gulwani. 2011. Spreadsheet table transformations from examples. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 317–328.

William R. Harris, Somesh Jha, Thomas W. Reps, Jonathan Anderson, and Robert N. M. Watson. 2013. Declarative, temporal, and practical programming with capabilities. In *IEEE Symposium on Security and Privacy*. 18–32.

Christoph Herbst, Elisabeth Oswald, and Stefan Mangard. 2006. An AES smart card implementation resistant to power analysis attacks. In *International Conference on Applied Cryptography and Network Security*. 239–252.

Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *International Conference on Software Engineering*. 215–224.

Zhen Hang Jiang, Yunsi Fei, and David R. Kaeli. 2016. A complete key recovery timing attack on a GPU. In *IEEE International Symposium on High Performance Computer Architecture*. 394–405.

Timo Kasper, David Oswald, and Christof Paar. 2011. Side-channel analysis of cryptographic RFIDs with analog demodulation. In *RFIDSec*. 61–77.

Paul C. Kocher. 1996. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *International Cryptology Conference*. 104–113.

Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential power analysis. In *International Cryptology Conference*. 388–397.

Boris Köpf and Markus Dürmuth. 2009. A provably secure and efficient countermeasure against timing attacks. In *IEEE Symposium on Computer Security Foundations*. 324–335.

Boris Köpf, Laurent Mauborgne, and Martín Ochoa. 2012. Automatic quantification of cache side-channels. In *International Conference on Computer Aided Verification*. 564–580.

Chao Luo, Yunsi Fei, Pei Luo, Saoni Mukherjee, and David R. Kaeli. 2015. Side-channel power analysis of a GPU AES implementation. In *IEEE International Conference on Computer Design*. 281–288.

Stefan Mangard. 2004. Hardware countermeasures against DPA – A statistical analysis of their effectiveness. In *The Cryptographers' Track at the RSA Conference 2004*. 222–235.

K. L. McMillan. 1994. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, MA.

Thomas S. Messerges. 2000. Securing the AES finalists against power analysis attacks. In *Fast Software Encryption*. 150–164.

Amir Moradi, Alessandro Barenghi, Timo Kasper, and Christof Paar. 2011. On the vulnerability of FPGA bitstream encryption against power analysis attacks: Extracting keys from Xilinx Virtex-II FPGAs. In *ACM Conference on Computer and Communications Security*. 111–124.

Amir Moradi, David Oswald, Christof Paar, and Pawel Swierczynski. 2013. Side-channel attacks on the bitstream encryption mechanism of Altera Stratix II: facilitating black-box analysis using software reverse-engineering. In *FPGA*. 91–100.

Amir Moradi, Axel Poschmann, San Ling, Christof Paar, and Huaxiong Wang. 2011. Pushing the limits: A very compact and a threshold implementation of AES. In *EUROCRYPT*. 69–88.

Andrew Moss, Elisabeth Oswald, Dan Page, and Michael Tunstall. 2012. Compiler assisted masking. In *Workshop on Cryptographic Hardware and Embedded Systems*. 58–75.

NIST. 2013. Keccak reference code submission to NIST's SHA-3 competition (Round 3). URL: http://csrc.nist.gov/groups/ST/hash/sha-3/Round3/documents/Keccak_FinalRnd.zip. (2013).

Elisabeth Oswald, Stefan Mangard, Norbert Pramstaller, and Vincent Rijmen. 2005. A side-channel analysis resistant description of the AES S-Box. In *International Workshop on Fast Software Encryption*. 413–423.

Andrei Sabelfeld and Andrew C. Myers. 2003. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications* 21, 1 (2003), 5–19.

Sergei Skorobogatov and Christopher Woods. 2012. Breakthrough silicon scanning discovers backdoor in military chip. In *Workshop on Cryptographic Hardware and Embedded Systems*. 23–40.

Armando Solar-Lezama, Rodric M. Rabbah, Rastislav Bodík, and Kemal Ebcioglu. 2005. Programming by sketching for bit-streaming programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 281–294.

TEMPEST. 1972. A signal problem – the story of the discovery of various compromising radiations from communications and Comsec equipment. *Cryptologic Spectrum, Vol. 2, No. 3, National Security Agency, partially FOAI declassified 2007-09-27* (1972).

Xilinx. 2014. MicroBlaze soft processor core. (2014). URL: http://www.xilinx.com/tools/microblaze.htm.

Conference Reports

JORGE A PÉREZ, University of Groningen, The Netherlands
j.a.perez@rug.nl

This installment of the conference report column includes a report by Bernardo Toninho (Imperial College London) on POPL 2017, which took place in Paris, France on January 15-21, 2017.[1]

POPL is the premier forum on all aspects of programming languages research. In his report, Bernardo describes the conference from the dual perspective of paper author and participant, with a focus on invited and contributed talks on type theory and concurrency. I am most grateful to Bernardo for his detailed report.

As usual, I look forward to receiving your personal impressions and/or reports on conferences and meetings broadly related to SIGLOG. I will also be pleased to hear your ideas and suggestions for future installments of the column.

[1] See http://conf.researchr.org/home/POPL-2017.

Report on POPL 2017

Bernardo Toninho, Imperial College London, United Kingdom

For researchers working on programming languages (or related fields), mid-January always brings the excitement of a new edition of the ACM Symposium on Principles of Programming Languages (POPL). POPL is a chance to learn about the best work that is happening in the field, usually with more emphasis on the theoretical aspects of programming language research, but not exclusively so – which is one of the many things that makes POPL great.

For this particular year, I had the opportunity to experience POPL in person for the first time both as an author and a participant. This year, POPL was located in a particularly cold Paris, at the Jussieu campus of Université Pierre et Marie Curie (Paris 6). It was one of the largest editions ever, with 722 registrants for the entire week (of which 571 registered specifically for POPL).

My coauthors and I arrived at the conference venue on the first day of POPL, not attending any of the workshops earlier in the week. The conference was split between two rooms. A larger auditorium where the invited talks, and one of the two parallel sessions, took place; and a smaller (but still quite large) amphitheatre located in a different building, hosting the second parallel session. Logistically this made it a bit harder than usual to swap between talks mid-session, but overall it worked out nicely (especially given the somewhat unprecedented size of the event).

The day opened with the first of the three invited talks for the week, which was delivered by Stephanie Weirich (University of Pennsylvania) on "The Influence of Dependent Types" in the Haskell GHC compiler. I could not quite secure a seat in the main room, but luckily the invited talks were broadcast to the second room, so I did not lose much. The goal of the talk was to zoom in on the several (type system) extensions to GHC that have been inspired by dependent types, and how these extensions enable (with varying degrees of ease) a form of dependently-typed programming within a general purpose language like Haskell.

The talk itself consisted of a live coding session of a regular expression parser, exploiting the several advanced type features of GHC. While it is a bit of a risk to expect that most of the audience is fluent in the more specialised "dependently-typed" Haskell forms, I think the talk worked very well in both showcasing what one can do with Haskell's rich type system, but also what one cannot do, or at least how the compromises of integrating such advanced type level features in an existing (non-total) language play out in practice. All in all, it was a very interesting exercise in the true spirit of POPL – how programming language theory can and does impact the practice in meaningful and relevant ways, even if it can take a while to get there.

Keeping with the roots of the conference, the first session of the day (in the main auditorium) was devoted to type systems in all their glory, ranging from fundamental problems like type inference with subtyping in ML and parametricity results for module systems to elegantly formal solutions to more practical problems such as representing and reasoning about incomplete pieces of code in editors using types. However, the main reason why I mention this particular session is that it held what I thought was the best talk of the conference (at least of those I attended), which was by Radu Grigore (University of Kent) on the Turing completeness of Java generics. The talk was a clever mix of the history of the formalisation of generics and the intuitions of how to write a Turing machine within Java's type system, including a grand *finale* (with roaring applause from the audience) demo of a Turing machine that checks for palindromes implemented in Java's *type system*.

The rest of the day consisted of the usual healthy mix of theory and practice that POPL is known for, with sessions on abstract interpretation, shared-memory concurrency and separation logic, compiler optimisation, probabilistic programming and logic. The day ended with the social dinner, held at the somewhat surreal venue that was the Musée des Arts Forains (a museum of old amusement park rides and related activities). It was likely the most fun conference dinner I have attended, since the participants were allowed (and encouraged) to engage in the various rides and activities within the venue – featuring a faster than expected bicycle-powered carousel.

The second day of festivities opened with a very different kind of invited talk (technically, it started with the Most Influential Paper and Reynolds Doctoral Dissertation awards), consisting of an interview with Patrick Cousot (New York University / ENS) by Roberto Giacobazzi (University of Verona) on the 40-year history of Abstract Interpretation. The session was an informal journey through the origins of abstract interpretation, dating back to the seminal 1977 POPL paper by Patrick and the late Radhia Cousot, covering the many challenges that were faced early on and the many successes that followed.

The rest of the morning had a very interesting type systems session (yet again), showing the more theoretical side of POPL with very good talks on equivalence in λ-calculi, a generalisation of the notion of type isomorphism to account for effects in a pleasing way, and on a polymorphic self-interpreter for an F_ω-style language (all of this happened in parallel with a session on program analysis that I could not attend). The afternoon focused on (more) shared-memory concurrency – as a side-note, its quite the sight to see so many groups working on formal reasoning and verification of concurrency, including the body of work on weak memory models (C++11) – effectful functional programming and semantics and concluded with the POPL business meeting. It is always amazing to see the herculean amount of work that PC chairs (and the PC itself) put into the organisation of a conference like POPL, and this year was no exception in the amount of careful work that ultimately resulted in yet another successful edition of the conference. The audience was also introduced to the city of Los Angeles where the 2018 edition of POPL will happen, and on some very much needed updates on the Open Access policies of ACM and SIGPLAN (spoiler alert: OA is a good thing). SIGPLAN is also looking at ways to improve the carbon footprint of conferences (spoiler alert #2: also a good thing).

The final day of the conference started with a very nice invited talk by Aaron Turon (Mozilla) on how it is indeed possible to realise the kind of programming language research ideas seen at conferences like POPL into a production-level language – Rust. The talk was structured as a form of walk-through of the kinds of problems Rust is trying to solve (safe systems programming) and how the notions of substructural typing and typeclasses made their way into the core of Rust's design. The morning closed off with the final type systems session and a session on verification and synthesis (apologies to the authors of the papers in the latter session, but I opted to attend the former). The types session featured a nice alternation of theory and practice, flowing from intersection type theories to techniques to prove type soundness of languages like (the core of) Scala; and from a computational reading of higher-dimensional type theory (in the HoTT-sense) to a metalanguage for (typed) embedded DSLs exploiting the macro system of Racket. The second half of the day kicked off with a session on gradual typing and one on concurrency (notably, the only non-separation logic/weak memory models concurrency session), and closed with parallel sessions on quantum computing and security. Shortly after I was on a train back to London, and thus my first (but certainly not last) POPL ended.

I will conclude this report with some general semi-random observations: POPL, while being focused on programming language research, features extremely varied work (e.g. sessions on quantum computing, type theory and compiler optimisation) which really showcases how diverse the field has become; student volunteering is a great way to see the "wonders of POPL" and every student everywhere should apply (and also to the student poster competition); and finally, "(good) theory matters!" is – at least to me – the motto of POPL (well, that and "well-typed programs can't go wrong").

From the invited talks to the research talks, the main take-away message is that practical problems can indeed have elegant theoretical solutions, and even the seemingly esoteric or "too abstract" works *do* indeed leave their mark, often in unexpected ways. So let's keep doing what we do in the way we know how to do it (and then submit it to next year's POPL)!

SIGLOG Monthly 191

DANIELA PETRIŞAN, Université Paris Diderot

SIGLOG Monthly 191
April 1, 2017

****
* Past issues of the newsletter are available at
 http://lii.rwth-aachen.de/lics/newsletters/
* Instructions for submitting an announcement to the newsletter
 can be found at
 http://lii.rwth-aachen.de/lics/newsletters/inst.html
****

TABLE OF CONTENTS

AIRIM'17 - Call for Papers
TIME 2017 - Call for Papers
SyGuS-COMP 2017 - Call for Solvers and Benchmarks Submission
NLS 2017 - Second announcement and call for papers
SR 2017 - Preliminary Call for Papers
HaPoC4 - Second Call for Papers
PPDP 2017 - Call for Papers
GandALF 2017 - Preliminary Call for Papers
RERS Challenge 2017 - Call for Papers
RSSRail 2017 - Call for Papers
LSFA 2017 - Second Call for Papers
EPS 2017 - Call for Posters and Encyclopedia Entries
* JOB ANNOUNCEMENTS

THIRTY-SECOND ANNUAL ACM/IEEE SYMPOSIUM ON LOGIC IN COMPUTER SCIENCE (LICS 2017)
18 - 23 June 2017, Reykjavik
Early registration deadline: April 7, 2017
http://lics.rwth-aachen.de/lics17/
http://www.icetcs.ru.is/lics2017-registration.html
* We strongly encourage conference and workshop participants to register, and to make their travel and accommodation arrangements, as soon as possible. Iceland is a very hot holiday destination these days and it becomes fully booked soon, especially during the summer months.
* ACCEPTED PAPERS
http://lics.rwth-aachen.de/lics17/accepted.html
* MENTORING WORKSHOP
Sponsorship application deadline:
31 March 2017
* OTHER WORKSHOPS
INFINITY: 19th International Workshop on Verification of Infinite-State Systems.
LearnAut: Learning and Automata.
LCC: Logic and Computational Complexity.
LMW: Logic Mentoring Workshop.
LOLA: Syntax and Semantics of Low-Level Languages.
METAFINITE model theory and definability and complexity of numeric graph parameters.
WiL: Women in Logic.
http://lics.rwth-aachen.de/lics17/workshops.html

ACM SIGLOG ANNOUNCEMENT
http://siglog.acm.org
* The ACM has recently chartered a Special Interest Group on Logic and Computation (ACM SIGLOG).
* We are pleased to announce the 2016 ACM SIGLOG election results for the term of 1 July 2016 - 30 June 2019. The SIGLOG Chair is Prakash Panangaden and the other officers are Luke Ong (vice-Chair), Amy

Felty (Treasurer) and Alexandra Silva (Secretary).
* The ACM-IEEE Symposium on Logic in Computer Science is the flagship conference of SIGLOG. SIGLOG will also actively seek association agreements with other conferences in the field. A SIGLOG newsletter (SIGLOG News) is also published quarterly in an electronic format with community news, technical columns, members' feedback, conference reports, book reviews and other items of interest to the community.
* One can join SIGLOG by visiting https://campus.acm.org/public/qj/gensigqj/siglist/gensigqj_siglist.cfm It is possible to join SIGLOG without joining ACM (the SIGLOG membership fee is $25 and $15 for students).

DATES
* CALCO 2017
Last Call for Papers
June 13 - 16, 2017
Ljubljana, Slovenia
http://coalg.org/calco17/
Paper submission: April 7, 2017
* CALCO Tools 2017
Call for Contributions
A satellite event of CALCO 2017
June 13, 2017, Ljubljana, Slovenia
http://coalg.org/calco17/tools.html
Paper submission: April 7, 2017
* LearnAut 2017
Call for Papers
LICS 2017 Workshop
June 19, Reykjavik (Iceland)
Website: https://learnaut.wordpress.com/
Submission deadline: April, 1st, 2017
* CCA 2017
Second Call for Papers
http://cca-net.de/cca2017/
July 24-27, 2017, Daejeon, South Korea
Submission deadline: April 3, 2017
* DARe at LPNMR 2017
Call for Papers
Espoo, Finland, 3 July 2017
Deadline: 10 April 2017
https://sites.google.com/view/dare-17
* METAFINITE 2017
First Call for Presentations
Affiliated with LICS 2017
June 19 2017, Reykjavik, Iceland
http://cs.technion.ac.il/~janos/metafinte2017
Submission: Friday 7 April 2017
* MARKTOBERDORF SUMMER SCHOOL
Call for Participation
Logical Methods for Safety and Security of Software Systems

August 2-11 2017
https://asimod.in.tum.de/2017/
Apply online: https://asimod.in.tum.de/2017/participation.shtml
Deadline: April 9

* ITP 2017
Call for Papers
Brasilia, Brazil - 25-29 September 2017
Co-located with TABLEAUX 2017 and FroCoS 2017
http://itp2017.cic.unb.br
Paper submission deadline: April 10, 2017

* FSCD 2017
Second Call for Papers
Second Call for Papers
4 - 7 September 2017, Oxford, UK
http://www.cs.ox.ac.uk/conferences/fscd2017/
Submission Deadline: 14 April 2017

* LACompLing 2017
Call for papers
Stockholm, August 18-19, 2017
http://staff.math.su.se/rloukanova/LACompLing17.html
Submission deadline for regular papers: April 14, 2017

* LORI-VI 2017
Third Call for Papers (extended deadline)
September 11-14, 2017, Hokkaido University, Sapporo, Japan
http://golori.org/lori2017/
Submission Deadline extended to Friday April 14

* CCC 2017
Call for papers
Loria, 26-30 June 2017, Nancy, France
https://members.loria.fr/MHoyrup/CCC/home.html
Deadline: 17 April 2017

* METAFINITE 2017
Second Call for Presentations (Extended deadline)
Affiliated with LICS 2017
June 19 2017, Reykjavik, Iceland
http://cs.technion.ac.il/~janos/metafinte2017
Submission: Friday 18 April 2017

* ESORICS 2017
Call For Papers
Oslo, Norway, September 11-15, 2017
https://www.ntnu.edu/web/esorics2017/
Paper submission deadline: April 19, 2017

* MFCS 2017
Call for Papers
Aalborg, Denmark, August 21-25, 2017
http://mfcs2017.cs.aau.dk/
Paper submission deadline: April 24th, 2017 (AoE)

* VECoS 2017
Call for Papers
August 24-25, 2017 Montreal, Canada
https://vecos.ensta-paristech.fr/2017/
Papers deadline: April 24, 2017 (AoE)

* TABLEAUX 2017
Call for Papers
Brasilia, Brazil - 25-29 September 2017
Co-located with FroCoS 2017 and ITP 2017.
http://tableaux2017.cic.unb.br
Submission deadline: April 25, 2017

* FroCoS 2017
Call for Papers
Brasilia, Brazil - 25-29 September 2017
Co-located with TABLEAUX 2017 and ITP 2017
http://frocos2017.cic.unb.br
Submission Deadlines: 24 April 2017 (abstracts)
and 28th April 2017 (full papers)

* RV 2017
Call for Papers and Tutorials
September 13-16, Seattle, WA, USA
http://rv2017.cs.manchester.ac.uk
Paper and tutorial deadline: May 1, 2017 (Anywhere on Earth)

* CiE 2017
Call for Informal Presentations
Turku, Finland, June 12-16, 2017
http://math.utu.fi/cie2017
Submission deadline: May 1, 2017

* CALCO EI 2017
Call for Contributions
A satellite event of CALCO 2017
June 13-16, 2017
http://coalg.org/calco17/ei.html
Submission of short contributions: May 1, 2017

* LOGIC COLLOQUIUM 2017
First Announcement and Call for Submissions
August 14-20, 2017, Stockholm, Sweden
https://www.lc17.conf.kth.se
Abstract submission for contributed talks: May 5, 2017

* AIRIM'17
Call for Papers
Prague, Czech Republic, 3 - 6 September, 2017
https://www.fedcsis.org/2017/airim
Paper submission (strict deadline): May 10 2017 23:59:59 pm HST

* TIME 2017
Call for Papers
Mons (Belgium), 16-18 October 2017
http://informatique.umons.ac.be/time2017/
Full papers due: May 12

* SyGuS-COMP 2017
Call for Solvers and Benchmarks Submission
July 22, 2017 Heidelberg, Germany (with CAV and SYNT)
http://www.sygus.org/SyGuS-COMP2017.html
Benchmark submission deadline: 15 May 2017

* NLS 2017
Second announcement and call for papers
Stockholm, August 7 - 11, 2017

Department of Mathematics, KrÃď'ftriket Campus, Stockholm University.
https://www.sls17.conf.kth.se
Early registration ends: May 15, 2017
* SR 2017
Preliminary Call for Papers
Liverpool, UK, July 26-27, 2017
http://sr2017.csc.liv.ac.uk/
Submission deadline: May 15, 2017
* HaPoC4
Second Call for Papers
4-7 October 2017, Masaryk University Brno
https://hapoc2017.sciencesconf.org/
Deadline: 15 May 2017
* PPDP 2017
Call for Papers
Namur, Belgium, October 9-11, 2017
(co-located with LOPSTR'17)
http://complogic.cs.mcgill.ca/ppdp2017
Deadline: 12 May (abstracts) / 19 May (papers)
* GandALF 2017
Preliminary Call for Papers
Rome, Italy, 20-22 September 2017
http://gandalf2017.istc.cnr.it
Paper submission deadline: May 26, 2017
* RERS Challenge 2017
Call for Papers
Santa Barbara, USA, July 2017.
Deadline for all submission: 01.07.2017
* RSSRail 2017
Call for Papers
November 14-16, 2017, Pistoia, Italy
https://conferences.ncl.ac.uk/rssrail/
paper submission deadline: June 8, 2017
* LSFA 2017
Second Call for Papers
23-24 September 2017, Brasilia, Brazil
Satellite event of TABLEAUX, FroCoS, and ITP 2017
http://lsfa2017.cic.unb.br/
Submission deadline: 21 June 2017
* EPS 2017
Call for Posters and Encyclopedia Entries
24, 25th of September 2017, Brasilia, Brazil
http://proofsystem.github.io/Encyclopedia/
Submission: August 1

7TH INTERNATIONAL CONFERENCE ON ALGEBRA AND COALGEBRA IN COMPUTER SCIENCE (CALCO 2017)
Last Call for Papers
June 13 - 16, 2017

Ljubljana, Slovenia
http://coalg.org/calco17/

* IMPORTANT DATES
Abstract submission: April 3, 2017
Paper submission: April 7, 2017
Author notification: May 15, 2017
Final version due: May 31, 2017

* SCOPE
CALCO aims to bring together researchers and practitioners with interests in foundational aspects, and both traditional and emerging uses of algebra and coalgebra in computer science.
It is a high-level, bi-annual conference formed by joining the forces and reputations of CMCS (the International Workshop on Coalgebraic Methods in Computer Science), and WADT (the Workshop on Algebraic Development Techniques). Previous CALCO editions took place in Swansea (Wales, 2005), Bergen (Norway, 2007), Udine (Italy, 2009), Winchester (UK, 2011), Warsaw (Poland, 2013) and Nijmegen (the Netherlands, 2015).

* INVITED SPEAKERS
- Nicoletta Sabadini - University of Insubria, IT
- Alex Simpson - University of Ljubljana, SL

Joint Session with MFPS on Metrics, Privacy and Learning:
- James Worrell - University of Oxford, UK (joint with MFPS)

Further Invited Speakers
- Catuscia Palamidessi - Ecole polytechnique, FR (joint with MFPS)
- Vincent Danos - Ecole normale superieure, FR (joint with MFPS)
- Marco Gaboardi - University at Buffalo, USA (joint with MFPS)

* SPECIAL SESSION ON METRICS, PRIVACY AND LEARNING
(joint event with MFPS)

* TOPICS OF INTEREST
- Abstract models and logics
- Specialised models and calculi
- Algebraic and coalgebraic semantics
- System specification and verification
- Corecursion in Programming Languages
- Algebra and Coalgebra in quantum computing
- String Diagrams and Network Theory

* BEST PAPER AND BEST PRESENTATION AWARDS
This edition of CALCO will feature two awards: a Best Paper Award whose recipients will be selected by the PC before the conference and a Best Presentation Award, elected by the participants.

* PC CHAIRS
Filippo Bonchi (ENS Lyon, France, co-chair)
Barbara Koenig (University of Duisburg-Essen, Germany, co-chair)

* PUBLICITY CHAIR
Fabio Zanasi (UCL, UK)

* SATELLITE WORKSHOPS: CALCO EARLY IDEAS AND CALCO TOOLS
The CALCO Early Ideas Workshop is intended to enable presentation of work in progress and original research proposals. PhD students and young researchers are particularly encouraged to contribute.
The CALCO Tools Workshop is dedicated to tools based on algebraic and/or coalgebraic principles.

CALCO 2017 will run together with the CALCO Early Ideas Workshop, with dedicated sessions at the end of each conference day. CALCO Tools will take place on June 13.

CALCO Tools 2017
Call for Contributions
A satellite event of CALCO 2017
June 13, 2017, Ljubljana, Slovenia
http://coalg.org/calco17/tools.html

* IMPORTANT DATES
Abstract submission: April 3, 2017
Paper submission: April 7, 2017
Author notification: May 15, 2017
Final version: May 31, 2017

* SCOPE A special workshop at CALCO 2017 is dedicated to tools based on algebraic and/or coalgebraic principles or that are emerging from the intersection of the two approaches, such as graph grammars or coinductive proof techniques. These include systems/prototypes/tools developed specifically for design, checking, execution, and verification of (co)algebraic specifications, but also tools targeting different application domains while making core or interesting use of (co)algebraic techniques. CALCO-Tools will take place on the same dates as the main CALCO conference, with no overlap between the technical programmes of the two events.

* INVITED SPEAKER
Nate Foster - Cornell University, USA

* PC CHAIR
Till Mossakowski - Universitat Magdeburg, Germany

* SUBMISSION DETAILS
http://coalg.org/calco17/tools.html

LEARNING AND AUTOMATA - LICS 2017 WORKSHOP (LEARNAUT 2017)
Call for Papers
June 19, Reykjavik (Iceland)
Website: https://learnaut.wordpress.com/

* Grammatical Inference (GI) studies machine learning algorithms for classical recursive models of computations like automata and grammars. The expressive power of these models and the complexity of associated computational problems are a major research topic within theoretical computer science (TCS). This workshop aims at offering a favorable place for dialogue and at generating discussions between researchers from these two communities. We invite submissions of recent works, possibly preliminary ones, related to the theme of the workshop. Similarly to how main machine learning conferences and workshops are organized, all accepted abstracts will be part of a poster session held during the workshop. Additionally, the Program Committee will select a subset of the abstracts for oral presentation. At least one author of each

accepted abstract is expected to represent it at the workshop. A
list of topics of interest can be found on the website.
* INVITED SPEAKERS, [TBC]:
Kim G. Larsen (Aalborg),
Mehryar Mohri (NYU & Google),
Alexandra Silva (UCL)
* IMPORTANT DATES:
Submission deadline: April, 1st, 2017
Notification of acceptance: mid-April, 2017

FOURTEENTH INTERNATIONAL CONFERENCE ON COMPUTABILITY AND COMPLEXITY IN ANALYSIS (CCA 2017)
Final Call for Papers
July 24-27, 2017, Daejeon, South Korea
http://cca-net.de/cca2017/
Submission deadline: April 3, 2017
* TOPICS
- Computable analysis
- Complexity on real numbers
- Constructive analysis
- Domain theory and analysis
- Theory of representations
- Computable numbers, subsets and functions
- Randomness and computable measure theory
- Models of computability on real numbers
- Realizability theory and analysis
- Reverse analysis
- Real number algorithms
- Implementation of exact real number arithmetic
* INVITED SPEAKERS
Hee-Kap Ahn (Pohang, Republic of Korea)
Veronica Becher (Buenos Aires, Argentina)
Anders Hansen (Cambridge, UK)
Takayuki Kihara (Berkeley, USA)
Amaury Pouly (Max Planck Institute, Germany)
Linda Brown Westrick (Connecticut, USA)
* FUNDING
Funding opportunities for student members of the Association for
Symbolic Logic (ASL) are available.
* IMPORTANT DATES
Submission deadline: April 3, 2017
Notification of authors: May 1, 2017
Final version: May 29, 2017
* Conference Web Page
http://cca-net.de/cca2017/
* SATELLITE WORKSHOP ON REAL VERIFICATION
A co-located "Workshop on Real Verification" will take place on
Friday, July 28
https://complexity.kaist.edu/CCA2017/workshop.html

THE FOURTH INTERNATIONAL WORKSHOP ON DEFEASIBLE AND AMPLIATIVE REASONING (DARe)

Call for Papers
Espoo, Finland, 3 July 2017
https://sites.google.com/view/dare-17
held at the International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2017)

* Latest News

There will be a special issue of the International Journal of Approximate Reasoning (IJAR) containing selected extended versions of papers that have been accepted at DARe. The call for this special issue is planned for late 2017. More information to follow.

* Workshop Description, Aims and Scope

Classical reasoning is not flexible enough when directly applied to the formalization of certain nuances of human quotidian decision making. These involve different kinds of reasoning such as reasoning with uncertainty, exceptions, similarity, vagueness, incomplete or contradictory information and many others.

DARe welcomes contributions on all aspects of defeasible and ampliative reasoning such as (but not limited to):

- Abductive and inductive reasoning
- Explanation finding, diagnosis and causal reasoning
- Inconsistency handling and exception-tolerant reasoning
- Decision-making under uncertainty and incomplete information
- Default reasoning, non-monotonic reasoning, non-monotonic logics, conditional logics
- Specific instances and variations of ampliative and defeasible reasoning
- Probabilistic and statistical approaches to reasoning
- Vagueness, rough sets, granularity and fuzzy-logics
- Philosophical foundations of defeasibility
- Empirical studies of reasoning
- Relationship with cognition and language
- Contextual reasoning
- Preference-based reasoning
- Analogical reasoning
- Similarity-based reasoning
- Belief dynamics and merging
- Argumentation theory, negotiation and conflict resolution
- Heuristic and approximate reasoning
- Defeasible normative systems
- Reasoning about actions and change
- Reasoning about knowledge and belief, epistemic and doxastic logics
- Ampliative and defeasible temporal and spatial reasoning
- Computational aspects of reasoning with uncertainty
- Implementations and systems
- Applications of uncertainty in reasoning

* IMPORTANT DATES

- Submission deadline: 10 April 2017
- Notification: 1 May 2017
- Camera ready: 22 May 2017

- Workshop date: 3 July 2017

* WORKSHOP CO-CHAIRS

- Richard Booth, Cardiff University, UK
- Giovanni Casini, University of Luxembourg
- Ivan Varzinczak, CRIL, Univ. Artois & CNRS, France

WORKSHOP ON METAFINITE MODEL THEORY AND DEFINABILITY AND COMPLEXITY OF NUMERIC GRAPH PARAMETERS (METAFINITE 2017)

First Call for Presentations
Affiliated with LICS 2017
June 19 2017, Reykjavik, Iceland
http://cs.technion.ac.il/~janos/metafinte2017

* AIM:

The workshop will bring together three strands of investigation dealing with the model theory and complexity of numeric graph parameters and their generalization to other first order structures.
(A) Gurevich and Graedel in 1998 initiated the study of metafinite model theory to study descriptive complexity of numeric parameters. Metafinite model theory found most of its applications in databases and abstract state machines (ASM), but was not widely studied in connection to numeric combinatorial parameters.
(B) Courcelle, Makowsky and Rotics initiated a definability theory for graph polynomials in 2000 and proved metatheorems for graph polynomials and numeric structural parameters.
(C) Kotek, Makowsky and Ravve questioned wether the Turing model of computation was the right choice to discuss the complexity of numeric graph parameters and proposed alternatives using the Blum-Shub-Smale model of computation.
The aim of the workshop is to bring together researchers of these three strands in order to further explore and elaborate on the appropriate framework for the study of numeric structural parameters and polynomials and to investigate further metatheorems.
For more background see http://cs.technion.ac.il/~janos/metafinte2017

* ORGANIZERS and PC:

A. Goodall (Charles University, Prague)
J.A. Makowsky (Technion, Haifa)
E.V. Ravve (ORT-Braude, Karmiel)

* CONFIRMED SPEAKERS:

E. Graedel (RWTH, Aachen)
K. Meer (BTU, Cottbus)
M. Ziegler (KAIST, Southkorea)
T. Kotek (TU, Vienna)

* IMPORTANT DATES (AoE):

Submission: Friday 7 April 2017
Notification: Monday 24 April 2017
Final version: Monday 15 May 2017
Workshop: Monday June 19 2017

* CONTACT:

For further questions write to Dr. Elena Ravve at cselena@braude.ac.il

MARKTOBERDORF SUMMER SCHOOL
Call for Participation
Logical Methods for Safety and Security of Software Systems
August 2-11 2017
https://asimod.in.tum.de/2017/

* Apply online: https://asimod.in.tum.de/2017/participation.shtml
Deadline: April 9
* The "Marktoberdorf Summer School" is an 11-day event for young computer scientists and mathematicians, typically doctoral and post-doctoral researchers. It provides mini-courses on state-of-the-art topics in "Logical Methods for Safety and Security of Software Systems" and leaves ample room for interaction between participants and speakers.
* SPEAKERS AND COURSES:
CHRISTEL BAIER:
Probabilistic Model Checking
GILLES BARTHE:
Relational Verification for Differential Privacy and Cryptography
NICOLAJ BJORNER:
Satisfiability Modulo Theories
CEDRIC FOURNET:
Security Verification in F*
ORNA GRUMBERG:
Program Repair
JOOST-PIETER KATOEN:
Foundations of Probabilistic Programming
DANIEL KROENING:
Static Analysers for Black Hats and White Hats
ORNA KUPFERMANN:
Automated Synthesis of Temporal-Logic Specifications
MAGNUS MYREEN:
Verification of an ML Compiler
TOBIAS NIPKOW:
Verified Analysis of Functional Data Structures
LARRY PAULSON:
Proof Support for Hybrid System Analysis
ANDRE PLATZER:
Dynamic Logic for Dynamical Systems

8th INTERNATIONAL CONFERENCE ON INTERACTIVE THEOREM PROVING (ITP 2017)
Call for Papers
Brasilia, Brazil - 25-29 September 2017
Co-located with TABLEAUX 2017 and FroCoS 2017
http://itp2017.cic.unb.br

* The ITP conference series is concerned with all topics related to interactive theorem proving, ranging from theoretical foundations to implementation aspects and applications in program verification, security, and formalization of mathematics. ITP is the evolution of the TPHOLs conference series to the broad field of interactive theorem proving. TPHOLs meetings took place every year from 1988

until 2009.
* PROGRAM CHAIRS
Mauricio Ayala-Rincon, University of Brasilia
Cesar Munoz, NASA
* ORGANISATION
University of Brasilia
Federal University of Rio Grande do Norte
* IMPORTANT DATES
Abstract submission deadline: April 3, 2017
Full paper submission deadline: April 10, 2017
Author notification: June 2, 2017
Camera-ready papers: June 30, 2017

SECOND INTERNATIONAL CONFERENCE ON FORMAL STRUCTURES FOR COMPUTATION AND DEDUCTION (FSCD'17)
Second Call for Papers
4 - 7 September 2017, Oxford, UK
in-cooperation with the ACM SIGLOG and SIGPLAN and co-located with ICFP 2017
http://www.cs.ox.ac.uk/conferences/fscd2017/
* FSCD (http://fscdconference.org/) covers all aspects of formal structures for computation and deduction from theoretical foundations to applications. Building on two communities, RTA (Rewriting Techniques and Applications) and TLCA (Typed Lambda Calculi and Applications), FSCD embraces their core topics and broadens their scope to closely related areas in logics, proof theory and new emerging models of computation such as quantum computing or homotopy type theory.
* IMPORTANT DATES.
All deadlines are midnight anywhere-on-earth (AoE) and are firm; late submissions will not be considered.
Abstract Deadline: 7 April 2017
Submission Deadline: 14 April 2017
Rebuttal: 29--31 May 2017
Notification: 14 June 2017
Camera-Ready: 7 July 2017
* TOPICS. Suggested, but not exclusive, list of topics for submission are:
1. Calculi: Lambda calculus * Concurrent calculi * Logics * Rewriting systems * Proof theory * Type theory and logical frameworks
2. Methods in Computation and Deduction: Type systems * Induction and coinduction * Matching, unification, completion, and orderings * Strategies * Tree automata * Model checking * Proof search and theorem proving * Constraint solving and decision procedures
3. Semantics: Operational semantics * Abstract machines * Game Semantics * Domain theory and categorical models * Quantitative models
4. Algorithmic Analysis and Transformations of Formal Systems: Type Inference and type checking * Abstract Interpretation * Complexity analysis and implicit computational complexity * Checking termination, confluence, derivational complexity and related properties * Symbolic computation

5. Tools and Applications: Programming and proof environments * Verification tools * Libraries for proof assistants and interactive theorem provers * Case studies in proof assistants and interactive theorem provers * Certification

* BEST PAPER AWARD BY JUNIOR RESEARCHERS The program committee will consider declaring this award to a paper in which all authors are junior researchers: a junior researcher is a person who is either a student or whose PhD award date is less than three years from the first day of the meeting.

* PROGRAM COMMITTEE CHAIR
Dale Miller, Inria Saclay & LIX <fscd17@easychair.org>

WORKSHOP ON LOGIC AND ALGORITHMS IN COMPUTATIONAL LINGUISTICS (LACOMPLING 2017)
Call for papers
Stockholm, August 18-19, 2017
http://staff.math.su.se/rloukanova/LACompLing17.html

* Affiliated with the 26th Annual EACSL Conference on Computer Science Logic CSL'2017 (Stockholm, 20--26 August 2017)
https://www.csl17.conf.kth.se/

* Co-located with Logic in Stockholm 2017:
https://www.lis17.conf.kth.se/

* DESCRIPTION
Computational linguistics studies natural language in its various manifestations from a computational point of view, both on the theoretical level (modeling grammar modules dealing with natural language form and meaning, and the relation between these two) and on the practical level (developing applications for language and speech technology). Right from the start in the 1950ties, there have been strong links with computer science and logic - one can think of Chomsky's contributions to the theory of formal languages and automata, or Lambek's logical modeling of natural language syntax. The workshop assesses the place of computer science logic in present day computational linguistics. It intends to be a forum for presenting new results as well as work in progress.

* SCOPE The workshop focuses on logical approaches to the computational processing of natural language, and on the applicability of methods and techniques from the study of artificial languages (programming/logic) in computational linguistics.

* TOPICS. The topics of LACompLing2017 include, but are not limited to:
- Computational theories of human language
- Computational syntax
- Computational semantics
- Computational syntax-semantics interface
- Interfaces between morphology, lexicon, syntax, semantics, speech, text, pragmatics
- Computational grammar
- Logic and reasoning systems for linguistics
- Type theories for linguistics
- Models of computation and algorithms for linguistics

- Language processing
- Parsing algorithms
- Generation of language from semantic representations
- Large-scale grammars of natural languages
- Multilingual processing
- Data science in language processing
- Machine learning of language
- Interdisciplinary methods
- Integration of formal, computational, model theoretic, graphical, diagrammatic, statistical, and other related methods
- Logic for information extraction or expression in written and spoken language
- Language theories based on biological fundamentals of information and languages
- Computational neuroscience of language

* IMPORTANT DATES

Submission deadline for regular papers: April 14, 2017
Notification of paper acceptance: May 31, 2017
Deadline for abstracts of short presentations: June 4, 2017
Notifications for short presentations: June 12, 2017
Deadline for final submissions: June 25, 2017
Workshop: August 18-19, 2017

* CONTACT

Roussanka Loukanova (rloukanova@gmail.com)
Valeria de Paiva (valeria.depaiva@gmail.com)

THE SIXTH INTERNATIONAL CONFERENCE ON LOGIC, RATIONALITY AND INTERACTION (LORI-VI 2017)

Third Call for Papers (extended deadline)
September 11-14, 2017, Hokkaido University, Sapporo, Japan
http://golori.org/lori2017/

* OVERVIEW. The International Conference on Logic, Rationality and Interaction (LORI) conference series aims at bringing together researchers working on a wide variety of logic-related topics that concern the understanding of rationality and interaction. The series aims at fostering a view of Logic as an interdisciplinary endeavour, and supports the creation of an East-Asian community of interdisciplinary researchers.

* NEWS. Submission Deadline extended to Friday April 14

* WEBSITE. For detailed conference information and registration, please visit http://golori.org/lori2017/ .

* INVITED SPEAKERS

Mike Dunn: Indiana University, U.S.A.
Alan Hajek: Australian National University, Australia
Nina Gierasimczuk: Technical University of Denmark, Denmark
Willemien Kets: Northwestern University, U.S.A
Sara Negri: University of Helsinki, Finland
Hiroakira Ono: JAIST, Japan

* PC CHAIRS

Alexandru Baltag: University of Amsterdam, The Netherlands
Jeremy Seligman: University of Auckland, New Zealand

CONTINUITY, COMPUTABILITY, CONSTRUCTIVITY FROM LOGIC TO ALGORITHMS 2017 (CCC 2017)

Call for papers
Loria, 26-30 June 2017, Nancy, France
https://members.loria.fr/MHoyrup/CCC/home.html

* OVERVIEW. CCC is a workshop series bringing together researchers from exact real number computation, computable analysis, effective descriptive set theory, constructive analysis, and related areas. The overall aim is to apply logical methods in these disciplines to provide a sound foundation for obtaining exact and provably correct algorithms for computations with real numbers and related analytical data, which are of increasing importance in safety critical applications and scientific computation. *

* SCOPE. The workshop specifically invites contributions in the areas

- Exact real number computation,
- Correctness of algorithms on infinite data,
- Computable analysis,
- Complexity of real numbers, real-valued functions, etc.
- Effective descriptive set theory
- Scott's domain theory,
- Constructive analysis,
- Category-theoretic approaches to computation on infinite data,
- Weihrauch degrees,
- And related areas.

* INVITED SPEAKERS.

Matthew de Brecht (Kyoto, Japan)
Bernhard Reus (Brighton, UK)
Matthias Schroder (Darmstadt, Germany)
Alex Simpson (Ljubljana, Slovenia)

* IMPORTANT DATES

Deadline: 17 April 2017

* PROGRAMME COMMITTEE CHAIRS

Mathieu Hoyrup (Nancy) (co-chair)
Dieter Spreen (Siegen) (co-chair)

WORKSHOP ON METAFINITE MODEL THEORY AND DEFINABILITY AND COMPLEXITY OF NUMERIC GRAPH PARAMETERS (METAFINITE 2017)

Affiliated with LICS 2017
June 19 2017, Reykjavik, Iceland
http://cs.technion.ac.il/~janos/metafinte2017

* AIM: The workshop will bring together three strands of investigation dealing with the model theory and complexity of numeric graph parameters and their generalization to other first order structures.

(A) Gurevich and Graedel in 1998 initiated the study of metafinite model theory to study descriptive complexity of numeric parameters. Metafinite model theory found most of its applications in databases and abstract state machines (ASM), but was not widely studied in connection to numeric combinatorial parameters.

(B) Courcelle, Makowsky and Rotics initiated a definability theory for graph polynomials in 2000 and proved metatheorems for graph

polynomials and numeric structural parameters.
(C) Kotek, Makowsky and Ravve questioned wether the Turing model of computation was the right choice to discuss the complexity of numeric graph parameters and proposed alternatives using the Blum-Shub-Smale model of computation.
(D) Nesetril and Ossona de Mendez introduced key notions in the theory of sparse graphs, such as graph families of bounded expansion or polynomial expansion and the (surprisingly robust) nowhere dense versus somewhere dense dichotomy. Their 2012 book 'Sparsity - Graphs, Structures, and Algorithms' combines model theory, analysis and combinatorics and gives a comprehensive overview. Recently Nesetril, along with Ossona de Mendez and Goodall, used finite model theory, and particularly interpretation schemes, to provide a general construction of polynomial graph invariants. This is an alternative approach to graph invariants related to the framework introduced by Makowsky and Zilber in 2005 and is best expressed in the framework of Metafinite model theory.

* The aim of the workshop is to bring together researchers of these four strands in order to further explore and elaborate on the appropriate framework for the study of numeric structural parameters and polynomials and to investigate further metatheorems.

* ORGANIZERS and PC:
A. Goodall (Charles University, Prague)
J.A. Makowsky (Technion, Haifa)
E.V. Ravve (ORT-Braude, Karmiel)

* KEYNOTE SPEAKERS: (* still to be confirmed)
Y. Gurevich (Microsoft, Redmond)
E. Graedel (RWTH, Aachen)
J. Nesetril (Charles University, Prague)

* CONFIRMED SPEAKERS:
K. Meer (BTU, Cottbus)
M. Ziegler (KAIST, Southkorea)
T. Kotek (TU, Vienna)
N. Labai (TU, Vienna)
A. Manuel (CMI, Chennai)
T. Colcombet (Paris VII, Paris)
J. Hubicka* (Charles University, Prague)
Introductory lecture will be given by the organizers.

* IMPORTANT DATES (AoE):
Submission: Friday 18 April 2017
Notification: Monday 1 May 2017
Final version: Monday 15 May 2017
Workshop: Monday June 19 2017

TWENTY-SECOND EUROPEAN SYMPOSIUM ON RESEARCH IN COMPUTER SECURITY (ESORICS 2017)

Call for Papers
Oslo, Norway, September 11-15, 2017
https://www.ntnu.edu/web/esorics2017/

* OVERVIEW
ESORICS is the annual European research event in Computer Security.

The Symposium started in 1990 and has been held in several European countries, attracting a wide international audience from both the academic and industrial communities. Papers offering novel research contributions in computer security are solicited for submission to the Symposium. The primary focus is on original, high quality, unpublished research and implementation experiences. We encourage submissions of papers discussing industrial research and development.

* IMPORTANT DATES

Paper submission deadline: April 19, 2017
Notification to authors: June 16, 2016
Camera ready due: July 26, 2016

* WORKSHOP CHAIR

Sokratis Katsikas, Norwegian University of Science and Technology (NTNU), Norway.

* PROGRAM COMMITTEE CHAIRS:

Dieter Gollman, Technische Universitat Hamburg-Harburg, Germany
Simon Foley, IMT Atlantique, France

42ND INTERNATIONAL SYMPOSIUM ON MATHEMATICAL FOUNDATIONS OF COMPUTER SCIENCE (MFCS 2017)

Call for Papers
Aalborg, Denmark, August 21-25, 2017
http://mfcs2017.cs.aau.dk/

* BACKGROUND:

MFCS conference series is organized since 1972. MFCS is a high-quality venue for original research in all branches of theoretical computer science. The broad scope of the conference encourages interactions between researchers who might not meet at more specialized venues. MFCS 2017 consists of invited lectures and contributed talks, selected by an international program committee of researchers focusing on diverse areas of theoretical computer science. The conference will be accompanied by workshops.

* We encourage submission of original research papers in all areas of theoretical computer science, including (but not limited to) the following (alphabetically ordered):

- algebraic and co-algebraic methods in computer science
- algorithms and data structures
- automata and formal languages
- bioinformatics
- combinatorics on words, trees, and other structures
- computational complexity (structural and model-related)
- computational geometry
- computer-assisted reasoning
- concurrency theory
- cryptography and security
- databases and knowledge-based systems
- formal specifications and program development
- foundations of computing
- logics in computer science
- mobile computing

- models of computation
- networks (incl. wireless, sensor, ad-hoc networks)
- parallel and distributed computing
- quantum computing
- semantics and verification of programs
- theoretical issues in artificial intelligence
- types in computer science

* IMPORTANT DATES:

Abstract submission deadline: April 20th, 2017 (AoE)
Paper submission deadline: April 24th, 2017 (AoE)
Notification of authors: June 12th, 2017 (AoE)
Camera-ready copies due: June 22nd, 2017 (AoE)
Early registration deadline: June 23rd, 2017 (AoE)
Late registration deadline: August 7th, 2017 (AoE; afterward, only on-site registration)
Conference dates: August 21 - 25, 2017

* PROGRAM CHAIRS:

Kim G. Larsen - PC chair (Aalborg University, Denmark)
Hans L. Bodlaender - co-chair (Eindhoven University of Technology, Netherlands)
Jean-Francois Raskin - co-chair (Universite Libre de Bruxelles, Belgium)

11TH INTERNATIONAL CONFERENCE ON VERIFICATION AND EVALUATION OF COMPUTER AND COMMUNICATION SYSTEMS (VECoS 2017)

Call for Papers
August 24-25, 2017 Montreal, Canada
https://vecos.ensta-paristech.fr/2017/

* The VECoS conference series is interested in the analysis of computer and communication systems in which functional and extra-functional properties are inter-related. VECoS encourages the cross-fertilization between the various formal verification and evaluation approaches, methods and techniques, and especially those developed for concurrent and distributed hardware/software systems.

* Topics of interest to the conference include, but are not limited to:

- Abstraction techniques
- Certification standards for real-time systems
- Compositional verification
- Correct-by-construction design
- Dependability assessment techniques
- Equivalence checking
- Model-checking
- Parameterized verification
- Performance and robustness evaluation
- Probabilistic verification
- QoS evaluation, planning and deployment
- RAMS (Reliability Availability Maintainability Safety) assessment
- Rigorous system design
- Security protocols verification
- Simulation techniques of discrete-event and hybrid systems
- Supervisory control

- Verification & validation of IoT
- Verification & validation of safety-critical systems
- Worst-case execution time analysis

* IMPORTANT DATES
Abstract deadline: April 10, 2017
Papers deadline: April 24, 2017 (Anywhere on Earth)
Paper notification: May 29, 2017
Camera-ready deadline: June 12, 2017
Conference: August 24-25, 2017

* INVITED SPEAKERS
Mourad Debbabi, Concordia University, Montreal, Canada
Michel Dagenais, Polytechnique Montreal, Canada
Mengchu Zhou, NJIT, Newark, NJ, USA

* PROGRAM CHAIRS
Kamel Barkaoui, CNAM, Paris, France
Hanifa Boucheneb, Polytechnique Montreal, Canada

26th INTERNATIONAL CONFERENCE ON AUTOMATED REASONING WITH ANALYTIC TABLEAUX AND RELATED METHODS (TABLEAUX 2017)

Call for Papers
Brasilia, Brazil - 25-29 September 2017
Co-located with FroCoS 2017 and ITP 2017.
http://tableaux2017.cic.unb.br

* TABLEAUX is the main international conference at which research on all aspects, theoretical foundations, implementation techniques, systems development and applications, of the mechanization of tableau-based reasoning and related methods is presented.

* Tableau methods offer a convenient and flexible set of tools for automated reasoning in classical logic, extensions of classical logic, and a large number of non-classical logics. For large groups of logics, tableau methods can be generated automatically. Areas of application include verification of software and computer systems, deductive databases, knowledge representation and its required inference engines, teaching, and system diagnosis.

* PROGRAM CHAIRS
Claudia Nalon, University of Brasilia, Brazil
Renate Schmidt, The University of Manchester, UK

* IMPORTANT DATES
Abstract deadline: April 18, 2017
Submission deadline: April 25, 2017
Notifications: June 8, 2017
Camera-Ready deadline: July 3, 2017.

* AWARDS
The TABLEAUX 2017 Best Paper Award will be presented to the best submission nominated and chosen by the Program Committee among the accepted papers.

11th INTERNATIONAL SYMPOSIUM ON FRONTIERS OF COMBINING SYSTEMS (FroCoS 2017)

Call for Papers

Brasilia, Brazil - 25-29 September 2017
Co-located with TABLEAUX 2017 and ITP 2017
http://frocos2017.cic.unb.br

* The main goal of the symposium is to disseminate and promote progress in research areas related to the development of techniques for the integration, combination, and modularization of formal systems together with their analysis.

* SCOPE OF CONFERENCE. In various areas of computer science, such as logic, computation, program development and verification, artificial intelligence, knowledge representation, and automated reasoning, there is an obvious need for using specialized formalisms and inference systems for selected tasks. To be usable in practice, these specialized systems must be combined with each other and integrated into general purpose systems. This has led---in many research areas---to the development of techniques and methods for the combination and integration of dedicated formal systems, as well as for their modularization and analysis.

* INVITED SPEAKERS
Katalin Bimbe (University of Alberta, Canada) (joint with TABLEAUX and ITP)
Jasmin Blanchette (Inria and LORIA, Nancy, France) (joint with TABLEAUX and ITP)
Cezary Kaliszyk (University of Innsbruck, Austria) (joint with TABLEAUX and ITP)
Cesare Tinelli (University of Iowa, USA)
Renata Wassermann (University of Sao Paulo, Brazil)

* PROGRAM COMMITTEE CHAIRS
Clare Dixon, University of Liverpool, UK
Marcelo Finger, Universidade de Sao Paulo, Brazil

* ORGANISATION
University of Brasilia
Federal University of Rio Grande do Norte

* WORKSHOPS AND TUTORIALS
Proposals for Workshops and Tutorial sessions have been solicited in a separate call, which can be found at http://frocos2017.cic.unb.br/#cfw.
There are co-located events, described at http://frocos2017.cic.unb.br/#colocated .

* IMPORTANT DATES
Abstract deadline: April 24, 2017
Submission deadline: April 28, 2017
Notifications: June 9, 2017
Camera-Ready deadline: June 23, 2017.

THE 17TH INTERNATIONAL CONFERENCE ON RUNTIME VERIFICATION (RV 2017)
Call for Papers and Tutorials
September 13-16, Seattle, WA, USA
http://rv2017.cs.manchester.ac.uk
rv2017@easychair.org

* Runtime verification is concerned with the monitoring and analysis of the runtime behaviour of software and hardware systems. Runtime

verification techniques are crucial for system correctness, reliability, and robustness; they provide an additional level of rigor and effectiveness compared to conventional testing, and are generally more practical than exhaustive formal verification. Runtime verification can be used prior to deployment, for testing, verification, and debugging purposes, and after deployment for ensuring reliability, safety, and security and for providing fault containment and recovery as well as online system repair.

* Topics of interest to the conference include, but are not limited to:

- specification languages
- monitor construction techniques
- program instrumentation
- logging, recording, and replay
- combination of static and dynamic analysis
- specification mining and machine learning over runtime traces
- monitoring techniques for concurrent and distributed systems
- runtime checking of privacy and security policies
- statistical model checking
- metrics and statistical information gathering
- program/system execution visualization
- fault localization, containment, recovery and repair
- integrated vehicle health management (IVHM)
- Application areas of runtime verification include cyber-physical
- systems, safety/mission-critical systems, enterprise and systems
- software, autonomous and reactive control systems, health management
- and diagnosis systems, and system security and privacy.

* We welcome contributions exploring the combination of runtime verification techniques with machine learning and static analysis. Whilst these are highlight topics, papers falling into these categories will not be treated differently from other contributions.

* IMPORTANT DATES.
Abstract deadline: April 24, 2017 (Anywhere on Earth)
Paper and tutorial deadline: May 1, 2017 (Anywhere on Earth)
Tutorial notification: May 21, 2017
Paper notification: June 26, 2017
Camera-ready deadline: July 24, 2017
Conference: September 13-16, 2017

* INVITED SPEAKERS
Rodrigo Fonseca, Brown University, USA
Vlad Levin and Jakob Lichtenberg, Microsoft Research, USA
Andreas Zeller, Saarland University, Germany

* PROGRAM CHAIRS
Shuvendu Lahiri, Microsoft Research, USA
Giles Reger, University of Manchester, UK

UNVEILING DYNAMICS AND COMPLEXITY - 13TH COMPUTABILITY IN EUROPE CONFERENCE (CIE 2017)
Call for Informal Presentations
Turku, Finland, June 12-16, 2017
http://math.utu.fi/cie2017
Submission deadline: May 1, 2017

* OVERVIEW: Continuing the tradition of past CiE conferences, in addition to the formal presentations based on the LNCS proceedings volume, we invite researchers to present informal presentations, that are prepared shortly before the conference and inform the participants about current research and work in progress (in the style of mathematics conferences).
* IMPORTANT DATES
  Submission deadline: May 1, 2017
  Notification of acceptance: Within two weeks of submission
* Please send us a brief description of your talk (one page) by the submission deadline May 1st.

CALCO Early Ideas 2017 WORKSHOP (CALCO EI 2017)
Call for Contributions
A satellite event of CALCO 2017
June 13-16, 2017
http://coalg.org/mfps-calco2017/cfp-early-ideas-workshop.html
Submission of short contributions: May 1, 2017

* SCOPE. The programme of CALCO 2017 will include special sessions reserved for the CALCO Early Ideas Workshop, featuring presentations of work in progress and original research proposals. PhD students and young researchers are particularly encouraged to contribute. Attendance at the workshop is open to all conference participants. The CALCO Early Ideas Workshop invites submissions on the same topics as the CALCO conference: reporting results of theoretical work, the way these results can support methods and techniques for software development, as well as experience with the transfer of the resulting technologies into industrial practice.
* IMPORTANT DATES
  Submission of short contributions: May 1, 2017
  Author notification: May 19, 2017
  Final version: May 31, 2017
* PC CHAIR
  Daniela Petrisan - Universite Paris 7, France

LOGIC COLLOQUIUM 2017 (LC 2017)
First Announcement and Call for Submissions
August 14-20, 2017, Stockholm, Sweden
https://www.lc17.conf.kth.se

* The Logic Colloquium 2017 (LC2017) is the 2017 Annual European summer meeting of the Association for Symbolic Logic (ASL) and will be held during August 14-20, 2017 at the main campus of Stockholm University. The Logic Colloquium 2017 is organised and hosted jointly by the Departments of Mathematics and Philosophy at Stockholm University, and is also supported by the KTH Royal Institute of Technology.
* LC2017 will be co-located with two other logic-related events, all taking place at Stockholm University:
  - the 3rd Nordic Logic Summer School, NLS2017, August 7-12
  - the 26th EACSL Annual Conference on Computer Science Logic,

CSL2017, August 20-24.
* There will be a joint session of CSL2017 and LC2017 in the morning of August 20. Further information about all events can be found at: https://www.lis17.conf.kth.se
* The programme of LC2017 will also include special sessions, which will be announced later.
* INVITED SPEAKERS
* Plenary speakers:
 - David Aspero (University of East Anglia)
 - Alessandro Berarducci (Pisa)
 - Elisabeth Bouscaren (Paris 11)
 - Christina Brech (Sao Paolo)
 - Sakae Fuchino (Kobe University)
 - Denis Hirschfeldt (University of Chicago)
 - Wilfrid Hodges (British Academy)
 - Emil Jerabek (Prague)
 - Per Martin-LÃűf (Stockholm University)
 - Dag Prawitz (Stockholm University)
 - Sonja Smets (University of Amsterdam)
* Tutorial speakers:
 - Patricia Bouyer-Decitre (LSV ENS Cachan)
 - Mai Gehrke (Paris 7)
* LC2017 invited highlight speakers for the joint LC-CSL session:
 - Veronica Becher (Buenos Aires)
 - Pierre Simon (UC Berkeley)
* IMPORTANT DATES
 Abstract submission for contributed talks: May 5, 2017
 Notification: TBA
* PC CHAIR
 Mirna Dzamonja (PC chair, University of East Anglia)
* CONTACTS AND ENQUIRIES
 For enquiries on scientific and programme issues, send email to: Mirna Dzamonja (M.Dzamonja@uea.ac.uk) For enquiries on organising matters, send email to: lc2017 at philosophy.su.se

2ND INTERNATIONAL WORKSHOP ON AI ASPECTS OF REASONING, INFORMATION, AND MEMORY 2017 (AIRIM'17)
 Call for Papers
 Prague, Czech Republic, 3 - 6 September, 2017
 https://www.fedcsis.org/2017/airim
* SCOPE:
 There is general realization that computational models of languages and reasoning can be improved by integration of heterogeneous resources of information, e.g., multidimensional diagrams, images, language, syntax, semantics, quantitative data, memory. While the event targets promotion of integrated computational approaches, we invite contributions from any individual areas related to information, language, memory, reasoning.
* IMPORTANT DATES
 Paper submission (strict deadline): May 10 2017 23:59:59 pm HST

Position paper submission: May 31, 2017
Authors notification: June 14, 2017
Final paper submission and registration: June 28, 2017
Final deadline for discounted fee: August 01, 2017
Conference dates: September 3-6, 2017

* ORGANIZERS
Roussanka Loukanova, Stockholm University, Sweden
M. Dolores Jimenez-Lopez, Universitat Rovira i Virgili, Spain
Henning Christiansen, Roskilde University, Denmark

* CONTACT INFORMATION
M. Dolores Jimenez-Lopez (mariadolores.jimenez@urv.cat)
Roussanka Loukanova (rloukanova@gmail.com)

24th INTERNATIONAL SYMPOSIUM ON TEMPORAL REPRESENTATION AND REASONING (TIME 2017)

Call for Papers
Mons (Belgium), 16-18 October 2017
http://informatique.umons.ac.be/time2017/

* TIME 2017 aims to bring together researchers interested in reasoning about temporal aspects of information in any area of Computer Science. The symposium, currently in its 24th edition, has a wide remit and intends to cater to both theoretical aspects and well-founded applications. One of the key aspects of the symposium is its interdisciplinarity, with attendees from distinct areas such as artificial intelligence, database management, logic and verification, and beyond. The symposium will encompass three tracks on temporal representation and reasoning in (1) Artificial Intelligence, (2) Databases and (3) Logic and Verification. Submissions of high-quality papers describing research results are solicited. See the webpage for a detailed list of topics of interest.

* PROGRAM COMMITTEE CHAIRS: Sven Schewe (University of Liverpool); Thomas Schneider (University of Bremen); Jef Wijsen (University of Mons)

* INVITED SPEAKERS:
Alessandro Artale (Free University of Bozen-Bolzano);
Javier Esparza (Technical University of Munich);
Sheila McIlraith (University of Toronto)

* Authors of selected papers will be invited to submit an extended version of their contribution to a special issue of the journal Theoretical Computer Science.

* IMPORTANT DATES:
Abstracts due: May 8, 2017;
Full papers due: May 12, 2017;
Notification: June 27, 2017;
Final version due: July 14, 2017;
Symposium: October 16-18, 2017

SYGUS-COMP 2017 4TH ANNUAL SYNTAX GUIDED SYNTHESIS COMPETITION (SyGuS-COMP 2017)

Call for Solvers and Benchmarks Submission

July 22, 2017 Heidelberg, Germany (with CAV and SYNT)
http://www.sygus.org/SyGuS-COMP2017.html

* The SyGuS Competition is an annual competition for solvers of the syntax-guided synthesis problem. This problem asks to find a program meeting a given logical formulae augmented with a grammar restricting the set of allowed implementations. These are formulated in SyGuS-IF, a logical formalism built on top of SMT-LIB.
* Benchmarks and Solvers submission is now open.
* IMPORTANT DATES:

Benchmark submission deadline:	15 May 2017
Deadline for first version of solvers:	7 June 2017

* Detailed information can be found on the webpage.

THIRD NORDIC LOGIC SUMMER SCHOOL (NLS 2017)
Second announcement and call for papers
Stockholm, August 7 - 11, 2017
Department of Mathematics, KrÃď'ftriket Campus, Stockholm University.
https://www.sls17.conf.kth.se

* The third Nordic Logic Summer School is arranged under the auspices of the Scandinavian Logic Society (http://scandinavianlogic.org/). The two previous schools were organized in Nordfjordeid, Norway (2013) and Helsinki (2015). The intended audience is advanced master students, PhD-students, postdocs and experienced researchers wishing to learn the state of the art in a particular subject. The school is co-located with Logic Colloquium 2017 (August 14-20) and Computer Science Logic 2017 (August 21-24).
* The school will consist of 10 five-hour courses, running in two qparallel streams. In addition, there will be short student presentations and poster sessions.
* The lectures start on:
Monday August 7, 9:00, and end Friday August 11, 16:15
* LECTURERS AND COURSES
The following lecturers and course topics are confirmed.
- Mirna Dzamonja (University of East Anglia)
Set Theory
- Martin Escardo (Birmingham)
Topological and Constructive Aspects of Higher-Order Computation
- Henrik Forssell (Oslo)
Categorical Logic
- Volker Halbach (Oxford)
Truth & Paradox
- Larry Moss (Indiana University, Bloomington)
Natural Logic
- Anca Muscholl (LaBRI, UniversitÃľ Bordeaux)
Logic in Computer Science - Control and Synthesis, from a Distributed Perspective
- Eric Pacuit (University of Maryland)
Logic and Rationality
- Peter Pagin and Dag WesterstÃěhl (Stockholm University)

Compositionality
- Sara L. Uckelman (Durham)
Medieval Logic
- Andreas Weiermann (Ghent)
Proof Theory

* IMPORTANT DATES
Registration:
Registration opens: March 6, 2017
Early registration ends: May 15, 2017
Late registration ends: August 4, 2017
Submission of abstracts for presentations and posters:
Opening: March 6, 2017
Closing: May 2, 2017
Notification of acceptance: May 9, 2017

* FURTHER INFORMATION
Further information about submissions, registration and accommodation possibilities will (in due time) be available on the NLS webpage:
General enquiries: nls2017 [at] philosophy.su.se
Accommodation enquiries: logic2017-accommodation [at] math.su.se

* PROGRAM CHAIR
Erik Palmgren (chair, Stockholm U)

5TH INTERNATIONAL WORKSHOP ON STRATEGIC REASONING (SR 2017)
Preliminary Call for Papers
Liverpool, UK, July 26-27, 2017
http://sr2017.csc.liv.ac.uk/

* OVERVIEW: Strategic reasoning is one of the most active research areas in the multi-agent system domain. The literature in this field is extensive and provides a plethora of logics for modelling strategic ability. Theoretical results are now being used in many exciting domains, including software tools for information system security, robot teams with sophisticated adaptive strategies, and automatic players capable of beating expert human adversaries, just to cite a few. All these examples share the challenge of developing novel theories and tools for agent-based reasoning that take into account the likely behaviour of adversaries. The international workshop on strategic reasoning aims to bring together researchers working on different aspects of strategic reasoning in computer science, both from a theoretical and a practical point of view. SR 2017 will be co-located with TARK 2017, which will be held in Liverpool on July 24-26, 2017.

* LIST OF TOPICS
The topics covered by the workshop include, but are not limited to, the following:
Logics for reasoning about strategic abilities;
Logics for multi-agent mechanism design, verification, and synthesis;
Logical foundations of decision theory for multi-agent systems;
Strategic reasoning in formal verification;
Automata theory for strategy synthesis;
Strategic reasoning under perfect and imperfect information;
Applications and tools for cooperative and adversarial reasoning;

Robust planning and optimisation in multi-agent systems;
Risk and uncertainty in multi-agent systems;
Quantitative aspects in strategic reasonings.

* IMPORTANT DATES
Abstract deadline: May 8, 2017
Submission deadline: May 15, 2017
Acceptance notification: June 16, 2017
Camera-ready deadline: June 30, 2017
Workshop: July 26-27, 2017

* WORKSHOP CO-CHAIRS
Wiebe van der Hoek, University of Liverpool
Bastien Maubert, University of Naples "Federico II"
Aniello Murano, University of Naples "Federico II"
Sasha Rubin, University of Naples "Federico II"

4TH INTERNATIONAL CONFERENCE ON HISTORY AND PHILOSOPHY OF COMPUTING (HaPoC4 2017)
Second Call for Papers
4-7 October 2017, Masaryk University Brno
https://hapoc2017.sciencesconf.org/

* HaPoC4 2017 will be held under the auspices of the DHST/DLMPS Commission for the History and Philosophy of Computing (HaPoC) www.hapoc.org

* OVERVIEW. HaPoC conferences aim to bring together researchers exploring the various aspects of the computer from historical or philosophical standpoint. The series aims at an interdisciplinary focus on computing, rooted in historical and philosophical viewpoints. The conference brings together researchers interested in the historical developments of computing, as well as those reflecting on the sociological and philosophical issues springing from the rise and ubiquity of computing machines in the contemporary landscape. For HaPoC 2017 we welcome contributions from logicians, philosophers and historians of computing as well as from philosophically aware computer scientists and mathematicians. We also invite contributions on the use of computers in art. As HaPoC conferences aim to provide a platform for interdisciplinary discussions among researchers, contributions stimulating such discussions are preferable.

* IMPORTANT DATES
Deadline for abstracts and extended abstracts: 15 May 2017
Notifications of acceptance: July 2016

19TH INTERNATIONAL SYMPOSIUM ON PRINCIPLES AND PRACTICE OF DECLARATIVE PROGRAMMING (PPDP 2017)
Call for Papers
Namur, Belgium, October 9-11, 2017
(co-located with LOPSTR’17)
http://complogic.cs.mcgill.ca/ppdp2017

* PPDP 2017 is a forum that brings together researchers from the declarative programming communities, including those working in the

functional, logic, answer-set, and constraint programming paradigms. The goal is to stimulate research in the use of logical formalisms and methods for analyzing, performing, specifying, and reasoning about computations, including mechanisms for concurrency, security, static analysis, and verification.

* This year the conference will be co-located with the 27th Int'l Symp. on Logic-Based Program Synthesis and Transformation (LOPSTR 2017).
* Submissions are invited on all topics from principles to practice, from foundations to applications. Topics of interest include, but are not limited to
  - Language Design: domain-specific languages; interoperability; concurrency, parallelism, and distribution; modules; probabilistic languages; reactive languages; database languages; knowledge representation languages; languages with objects; language extensions for tabulation; metaprogramming. -- Implementations: abstract machines; interpreters; compilation; compile-time and run-time optimization; garbage collection; memory management.
  - Foundations: type systems; type classes; dependent types; logical frameworks; monads; resource analysis; cost models; continuations; control; state; effects; semantics.
  - Analysis and Transformation: partial evaluation; abstract interpretation; control flow; data flow; information flow; termination analysis; resource analysis; type inference and type checking; verification; validation; debugging; testing.
  - Tools and Applications: programming and proof environments; verification tools; case studies in proof assistants or interactive theorem provers; certification; novel applications of declarative programming inside and outside of CS; declarative programming pearls; practical experience reports and industrial application; education.
* IMPORTANT DATES:
  Abstract Submission: 12 May 2017
  Paper Submission: 19 May 2017
  Paper Rebuttal: 10 July 2017
  Notification: 20 July 2017
  Final Version: 15 Aug 2017
* SUBMISSION CATEGORIES:
  Submissions can be made in three categories: regular Research Papers, System Descriptions, and Experience Reports.
* PROGRAM COMMITTEE CHAIR
  Brigitte Pientka (McGill University)

THE EIGHTH INTERNATIONAL SYMPOSIUM ON GAMES, AUTOMATA, LOGICS, AND FORMAL VERIFICATION (GandALF 2017)
Rome, Italy, 20-22 September 2017
http://gandalf2017.istc.cnr.it

* AIM.
  The aim of GandALF 2017 symposium is to bring together researchers from academia and industry which are actively working in the fields of Games, Automata, Logics, and Formal Verification. The

idea is to cover an ample spectrum of themes, ranging from theory to applications, and stimulate cross-fertilization.

* TOPICS

Papers focused on formal methods are especially welcome. Authors are invited to submit original research or tool papers on all relevant topics in these areas. Papers discussing new ideas that are at an early stage of development are also welcome. The topics covered by the conference include, but are not limited to, the following:

- Automata Theory
- Automated Deduction
- Computational aspects of Game Theory
- Concurrency and Distributed computation
- Decision Procedures
- Deductive, Compositional, and Abstraction Techniques for Verification
- Finite Model Theory
- First-order and Higher-order Logics
- Formal Languages
- Formal Methods for Systems Biology, Hybrid, Embedded, and Mobile Systems
- Games and Automata for Verification
- Game Semantics
- Logical aspects of Computational Complexity
- Logics of Programs
- Modal and Temporal Logics
- Model Checking
- Models of Reactive and Real-Time Systems
- Program Analysis and Software Verification
- Run-time Verification and Testing
- Specification and Verification of Finite and Infinite-state Systems
- Synthesis

* IMPORTANT DATES

Abstract submission: May 19, 2017
Paper submission: May 26, 2017
Notification: July 7, 2017
Camera-ready: July 31, 2017
Conference: Sept. 20-22, 2017

* PROGRAM CHAIRS

Patricia Bouyer-Decitre LSV, CNRS & ENS de Cachan, France
Pierluigi San Pietro, Politecnico di Milano, Italy

7TH INTERNATIONAL CHALLENGE ON THE RIGOROUS EXAMINATION OF REACTIVE SYSTEMS (RERS 2017)

Call for Papers
Santa Barbara, USA, July 2017.
co-located with ISSTA/SPIN 2017.

* RERS is designed to encourage software developers and researchers to apply and combine their tools and approaches in a free style manner to answer evaluation questions for reachability and LTL formulas on specifically designed benchmarks. The goal of this challenge is to

provide a basis for the comparison of verification techniques and available tools. The benchmarks are automatically synthesized to exhibit chosen properties and then enhanced to include dedicated dimensions of difficulty, ranging from conceptual complexity of the properties (e.g. reachability, full safety, liveness), over size of the reactive systems (a few hundred lines to tens of thousands of them), to exploited language features (arrays and index arithmetics). They are therefore especially suited for community-overlapping tool comparisons. What distinguishes RERS from other challenges is that the challenge problems can be approached in a free-style manner: it is highly encouraged to combine and exploit all known (even unusual) approaches to software verification. In particular, participants are not constrained to their own tools. To clearly separate RERS from other challenges, this year the LTL analysis is separated from the reachability of labels. RERS is then the only challenge with a special track for LTL analysis on synthesized benchmarks.

* The main aims of RERS 2017 are to : - encourage the combination of usually different research fields for better software verification results; - provide a comparison foundation based on differently tailored benchmarks that reveals the strengths and weaknesses of specific approaches; - initiate a discussion for better benchmark generation reaching out across the usual community barriers to provide benchmarks useful for testing and comparing a wide variety of tools.
* There will be a 1 day workshop where the results will be presented, the generation methodology will be explained, and the modalities for the RERS 2018 challenge, which will be part of ISoLA 2018 will be discussed. There is still a lot of time to get engaged, and collecting RERS achievements is a lot of fun!
* SCHEDULE:
* SEQUENTIAL PROBLEMS
  The sequential challenge just started. Its entire setup in online since a few days. Thus you can start right away. At least if you are a RERS newcomer, we would strongly recommend you to start with the training problems:
  (http://www.rers-challenge.org/2017/index.php?page=trainingphase)
  They are an ideal starting point for the challenge: They are smaller in size than the challenge problems but otherwise structurally equivalent. Moreover, an automatic checker (available on the same page) allows you to evaluate your own solutions. After having tackled the training problems it should be easy to move on to attack the challenge problems.
* PARALLEL PROBLEMS: 01.03.2017:
  The training problems for the parallel challenge wil be online
  01.05.2017: The setup for the parallel challlenge will be online.
* DEADLINE for all submission: 01.07.2017
  Please note that we want to specifically encourage also solutions from participants that work with tools developed by others.
* FURTHER INFORMATION
  More detailed information on the challenge can be found in the participants section of www.rers-challenge.org/2017. Looking forward

to seeing you in Santa Barbara! Best regards Bernhard, Falk, Jaco, and Markus

INTERNATIONAL CONFERENCE ON RELIABILITY, SAFETY AND SECURITY OF RAILWAY SYSTEMS: MODELLING, ANALYSIS, VERIFICATION AND CERTIFICATION - (RSSRAIL 2017)

Call for Papers
November 14-16, 2017, Pistoia, Italy
https://conferences.ncl.ac.uk/rssrail/

* AIMS.
The railway industry is facing an increasing pressure to improve system safety, to decrease the production cost and time to market, to reduce the carbon emission and running cost, and to improve the system capacity. Railway systems are now being integrated into larger multi-transport networks. Such systems require an even higher degree of automation at all levels of operation. These trends dramatically increase the complexity of railway applications and pose new challenges in developing novel methods of modelling, analysis, verification and validation to ensure their reliability, safety and security, as well as in supporting novel mechanisms and procedures to help argue that the development processes are meeting the standards. Following the success of RSSRail 2016 held in Paris on June 28-30, 2016, this conference will contribute to a range of key objectives. There is a pressing demand to bring together researchers and developers working on railway system reliability, security and safety to discuss how these requirements can be met in an integrated way.

* The conference aims to bring together researchers and engineers interested in building critical railway applications and systems. This will be a working conference in which research advances will be discussed and evaluated by both researchers and engineers focusing on their potential to be deployed in industrial settings.

* We are interested in the submissions of three types:

- Research papers
- Industrial experience reports
- PhD student papers.

* IMPORTANT DATES:
Abstract submission: June 1, 2017
Paper submission deadline: June 8, 2017
Notification: July 8, 2017
camera-read papers submitted: August 18, 2017

* CONFERENCE CHAIRS:
Alessandro Fantechi, University of Firenze, Italy
Thierry Lecomte, ClearSy, France
Alexander Romanovsky, Newcastle University, UK

12TH WORKSHOP ON LOGICAL AND SEMANTIC FRAMEWORKS, WITH APPLICATIONS (LSFA 2017)

Second Call for Papers

23-24 September 2017, Brasilia, Brazil
Satellite event of TABLEAUX, FroCoS, and ITP 2017
http://lsfa2017.cic.unb.br/

* OVERVIEW

Logical and semantic frameworks are formal languages used to represent logics, languages and systems. These frameworks provide foundations for the formal specification of systems and programming languages, supporting tool development and reasoning.

LSFA 2017 aims to be a forum for presenting and discussing work in progress, and therefore to provide feedback to authors on their preliminary research. The proceedings are produced after the meeting, so that authors can incorporate this feedback in the published papers.

* TOPICS OF INTEREST

Topics of interest to this forum include, but are not limited to:

- Automated deduction
- Applications of logical and/or semantic frameworks
- Computational and logical properties of semantic frameworks
- Formal semantics of languages and systems
- Implementation of logical and/or semantic frameworks
- Lambda and combinatory calculi
- Logical aspects of computational complexity
- Logical frameworks
- Process calculi
- Proof theory
- Semantic frameworks
- Specification languages and meta -languages
- Type theory

* IMPORTANT DATES

Submission: 21 June 2017
Notification: 21 July 2017
Final pre-proceedings version due: 11 August 2017
LSFA 2017 23-24 September 2017

* PROGRAMME COMMITTEE CHAIRS

Sandra Alves, University of Porto - co-chair
Renata Wassermann, University of Sao Paulo - co-chair

ENCYCLOPEDIA OF PROOF SYSTEMS - POSTER SESSION & TASK-FORCE (EPS 2017)

Call for Posters and Encyclopedia Entries
24, 25th of September 2017, Brasilia, Brazil
http://proofsystem.github.io/Encyclopedia/

* DESCRIPTION. The Encyclopedia of Proof Systems was created in 2014 with the goal of being a quick reference for the various proof systems used by logicians. Since then, it has collected 64 entries on the most various logics and calculi. This was only possible due to the collaboration of many members of the logic community. This event aims to promote the encyclopedia and attract more contributions and collaborators. It consists of:

- a poster session in the afternoon of September 24th, 2017, during which submitted entries will be displayed as posters;
- an interactive hands-on meeting in the morning of September 25th, 2017, for those who would like to contribute to the continuous

improvement of the encyclopedia.

* Submissions and instructions are available in the website:
  http://proofsystem.github.io/Encyclopedia/

* IMPORTANT DATES
  Submission: 1 August 2017
  Notification:	15 August 2017